國立交通大學

電機學院 IC 設計產業研發碩士班

碩士論文

應用於H.264/AVC 1080HD的高產量背景適應性二元算術編解碼器

A High Throughput Context Adaptive Binary Arithmetic Codec for H.264/AVC 1080HD Application

研究生：林秉昌

指導教授：李鎮宜　教授

中華民國九十六年一月

應用於H.264/AVC 1080HD的高產量背景適應性二元算術編解碼器

A High Throughput Context Adaptive Binary Arithmetic Codec for H.264/AVC 1080HD Application

研究生：林秉昌　　Student：Ping-Chang Lin

指導教授：李鎮宜 教授　　Advisor：Dr. Chen-Yi Lee

國立交通大學

電機學院 IC 設計產業研發碩士班

碩士論文

A Thesis
Submitted to College of Electrical and Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Industrial Technology R & D Master Program on
IC Design

January 2007
Hsinchu, Taiwan, Republic of China

中華民國九十六年一月

應用於H.264/AVC 1080HD的高產量背景適應性二元算術編解碼器

學生：林秉昌　　　　指導教授：李鎮宜 教授

國立交通大學

電機學院產業研發碩士班

摘要

在本論文中，我們對算數編碼器提出三個提升生產率的方法以提高整個背景適應性二元算數(CABAC)編碼器的產率，此外我們也對CABAC編解碼器提出硬體共享的方法以降低其硬體成本。關於我們的解碼器部份，我們沿用我們在2006年8月份所發表的高產率CABAC解碼器 [2]。我們所提出的提高產率的方法主要是以所統計的資料分布特性來設計，這可以改善資料相依特性所產生的產率受限制情形。我們所提出的架構可以達到平均每秒處裡245760個巨方塊，在編碼時可以滿足層次4.1，而在解碼時可以滿足層次4.0。因此，這足以對1080HD格式每秒三十張畫面的影像做即時編解碼。基於0.18微米聯華電子互補式金氧半導體製程，我們的CABAC編解碼器設計需要173303個邏輯閘(含SRAM)，其操作時脈為110MHz。

A High Throughput Context Adaptive Binary Arithmetic Codec for H.264/AVC 1080HD Application

Student：Ping-Chang Lin　　　　Advisor：Prof. Chen-Yi Lee

Industrial Technology R & D Master Program of
Electrical and Computer Engineering College
National Chiao Tung University

ABSTRACT

In this thesis, we propose three high throughput methods for arithmetic encoder to promote the throughput of CABAC encoder and some hardware sharing methods for CABAC codec to lower its cost. About the decoder of our CABAC codec, we continue using the high throughput CABAC decoder [2] which we presented in August 2006. The proposed high throughput methods are based on the results of data statistic, they can improve the restriction in throughput caused by data dependence. Our proposed architecture can achieve 245760 macroblocks per second in average for level 4.1 in encoding and for level 4.0 in decoding. Therefore, it is sufficient to support 1080HD for real-time encoding/decoding at 30fps. Based on 0.18 μm UMC CMOS Process, our CABAC codec design needs 173303 gates with SRAM and clock rate at 110MHz.

誌　　謝

首先要感謝恩師 李鎮宜教授，在老師的細心指導下，不但習得做研究的方法，且逐漸建立了老師一直強調的系統層級設計考量的設計觀念。這使我在微觀的設計同時，也漸漸可以用巨觀的角度審視自己的設計是否滿足系統的需求。在每次會議報告中老師給予的寶貴意見及 SI2 實驗室提供了優良的積體電路設計環境，使我在研究上能更順利的完成。

再來是感謝 SI2 實驗室的學長：毅宏、康正、俊彥、陳元，及同學們：皆賢、岳琪、大嘉、堉棋、名威、文平、義閔的指導幫助，其中黃毅宏 學長更是細心指導我在硬體設計上正確觀念的建立。在交大研究所這兩年，大家一起熬過艱難日子所培養出的革命情感，使我衷心希望大家都有著美好的未來。

最後要感謝我的 父母的關心與協助，使我能無後顧之憂得專心在做研究上。

謹將此論文獻給關心及愛護我的家人與朋友們。

Contents

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Organization of thesis 2

Chapter 2 Algorithm of CABAC for H.264/AVC 4

2.1 Overview of CABAC encoding/decoding flow 5

2.2 Binarization process 8

2.2.1 Unary (U) binarization process 8

2.2.2 Truncated unary (TU) binarization process 10

2.2.3 Fixed-length (FL) binarization process 12

2.2.4 Unary/k-th order Exp_Golomb (UEGk) binarization process 13

2.2.5 Special binarization process 15

2.3 Algorithm Of basic binary arithmetic coding 19

2.3.1 Basic binary arithmetic encoding algorithm 20

2.3.2 Basic binary arithmetic decoding algorithm 22

2.4 Advanced binary arithmetic coding for H.264/AVC 24

2.4.1 Advanced binary arithmetic encoding algorithm 24

2.4.2 Advanced binary arithmetic decoding algorithm 31

2.5 Context model organization 35

2.6 Syntax elements for the neighbor blocks 40

2.7 Paper survey for state-of –the-art CABAC designs 43

Chapter 3 Binary Arithmetic Encoding Engine 47

3.1 Overview of CABAC 48

3.2 Three throughput promoting methods 51

3.2.1 Multi-symbol architecture 52

3.2.2 Pipeline organization 60

3.2.3 Case Efficiency Architecture 63

3.2.3.1 Throughput bottleneck of arithmetic encoding engine 64

3.2.3.2 Throughput efficiency design 66

3.2.3.3 Cost issue in output interface 79

3.2.3.4 Cost efficiency design 80

Chapter 4 CABAC Codec System Architecture 85

4.1 Hardware sharing methods 85

4.2 Memory requirement 89

Chapter 5 Simulation and Implementation Result for Digital TV Application 91

5.1 Specification of different levels 92

5.2 Simulation and inplementation result 94

Chapter 6 Conclusion and Future Work 100

6.1 Conclusion 100

6.2 Future work 101

6.3 Discussion from H.264/AVC system view 101

Bibliography 104

About The Author 106

List of Figures

Figure 1 H.264/AVC encoder/decoder system block diagram................................ 5

Figure 2 CABAC encoder flow chart [4].. 6

Figure 3 CABAC decoder flow chart 7

Figure 4 (a) Definition of MPS and LPS 20

Figure 4 (b) Sub-divided interval of MPS.. 20

Figure 4 (c) Sub-divided interval of LPS 20

Figure 5 (a) Result of MPS subdivision.. 23

Figure 5 (b) Result of LPS subdivision... 23

Figure 6 Flowchart of the normal encoding flow [1] 25

Figure 7 Flowchart of renormalization in the encoder [1] 26

Figure 8 Flowchart of PutBit(B) [1] 27

Figure 9 Flowchart of encoding bypass [1] 28

Figure 10 Flowchart of the terminal encoding flow [1] 29

Figure 11 Flowchart of flushing at termination[1]... 30

Figure 12 Flowchart of the normal decoding flow [1] 32

Figure 13 Flowchart of renormalization in the decoder [1] 33

Figure 14 Flowchart of the bypass decoding flow [1] 34

Figure 15 Flowchart of the terminal decoding flow [1] 34

Figure 16 Illustration of the neighbor location in macroblock level 41

Figure 17 Illustration of the neighbor location for sub-macroblock level 42

Figure 18 Arithmetic coding interval update. (a) Complete iteration. [11] 44

Figure 18 Arithmetic coding interval update. (b) range updating. [11] 44

Figure 18 Arithmetic coding interval update. (c) Pre-computations for low updating. [11] 44

Figure 19 CABAC & MZ pseudo-code description [9]....................................... 45

Figure 20 Sample update of codlLow [12] 46

Figure 21 System architecture of CABAC encoder 48

Figure 22 System architecture of CABAC decoder 50

Figure 23 Percentage of the usage of the three encoding modes in AE 51

Figure 24 Percentage of the number of the concatenate bypass encoding per bypass demand under executing CIF frames for four typical video sequences... 54

Figure 25 Percentage of the number of the concatenate bypass encoding per bypass demand under executing 1080HD frames for two typical video sequences.. 55

Figure 26 Bypass encoding unit 57

Figure 27 Architecture of the multi-symbol bypass encoding............................. 58

Figure 28 Timing diagram of the pipeline comparison 60

Figure 29 Timing diagram of bypass encoding just after normal encoding 61

Figure 30 Timing diagram after taking the method shown in Section 3.2.3....... 62

Figure 31 Statistic of bitsOutstanding for the low complexity and slow motion video sequence 66

Figure 32 Statistic of bitsOutstanding for the high complexity and fast motion video sequence 67

Figure 33 Statistic of bitsOutstanding for 1080HD video sequence 68

Figure 34 Tree map of the renormalization process........ 69

Figure 35 Illustration of the shifting left of codlLow 74

Figure 36 Probabilities of the different containing range of remainder bitsOutstaning based on Table 17 76

Figure 37 Case Efficiency Architecture for renormalization process........ 77

Figure 38 The proposed bit-stream output interface 81

Figure 39 Three main parts of CABAC encoder........ 87

Figure 40 Illustration of hardware sharing for CABAC encoder and decoder.. 87

Figure 41 Architecture of the hardware sharing CABAC codec system........ 88

Figure 42 The required memory size of our CABAC codec system 89

Figure 43 The characteristic curves of station video sequence (QP: 42 ~ 16) 94

Figure 44 The characteristic curves of riverbed video sequence (QP: 42 ~ 16) . 95

Figure 45 System block diagram of H.264/AVC decoding for main profile 102

Figure 46 Bit-stream structure of H.264/AVC 103

List of Tables

Table 1 *bin* string of the unary binarization [1].. 9

Table 2 *bin* string of the fixed-length code 12

Table 3 Binarization for macroblock types in I slice.. 16

Table 4 Binarization for macroblock types in P, SP, and B slices 18

Table 5 Binarization for sub-macroblock types in P, SP, and B slices................ 19

Table 6 Value of *ctxIdxOffset* definition [1].. 37

Table 7 Assignment of *ctxBlockCat* due to coefficient type [1]............................ 37

Table 8 Assignment of ctxIdxBlockCatOffset due to ctxBlockCat and syntax elements of the residual data [1] 38

Table 9 Definition of the *ctxIdxInc* value for context model index [1]................. 38

Table 10 Required syntax elements of the left and top neighbor blocks and the computation for *ctxIdxInc*... 39

Table 11 Assignment of *ctxIdx* for syntax element *mbType*.................................. 39

Table 12 The cases corresponding to its accumulating number of bitsOutstanding=7 (It is under last remainder bitsOutstanding=0) (4 cases) 70

Table 13 The cases corresponding to its accumulating number of bitsOutstanding=6 (It is under last remainder bitsOutstanding=0) (12 cases) 71

Table 14 The cases corresponding to its accumulating number of bitsOutstanding=5 (It is under last remainder bitsOutstanding=1) (32 cases) 71

Table 15 The number of the corresponding cases for the accumulating number of bitsOutstanding of one symbol........ 73

Table 16 The utility rate of each case for one symbol 75

Table 17 The number of cases and the probabilities of the different containing range of remainder bitsOutstanding 76

Table 18 The relationship between shift and codIRange 79

Table 19 Probability of bitsOutstanding being greater than 7........ 80

Table 20 Required buffer slot for the different length bit-stream 82

Table 21 The states of FSM which are similar procedures between binarization and debinarization 86

Table 22 Maximum frame rates (fps) for some different frame types [1]........ 92

Table 23 level limits [1] 93

Table 24 The proposed arithmetic encoder comparing with the existing designs 96

Table 25 The proposed CABAC codec comparing with the existing CABAC designs 96

Table 26 Percentage of the cycle reduction for the proposed three throughput promoting methods 98

Chapter 1
Introduction

1.1 Motivation

H.264/AVC is the state-of-the-art video coding standard developed by ITU-T Video Coding Experts Group and ISO/IEC Moving Picture Experts Group (MPEG). The new standard provides gains in compression efficiency of up to 50% over a wide range of bit rates and video resolutions compared with the former standards such as H.263 and MPEG-4 by employing many innovative technologies such as multiple reference frame, variable block size motion estimation, in-loop de-blocking filter and context-based adaptive binary arithmetic coding (CABAC). Because of its outstanding performance in quality and compression gain, the H.264/AVC is adopted to be video standard in more and more consumer application products such as digital video recorder / player, portable video device…etc.

H.264/AVC contains two alternative entropy coding schemes which are context-based adaptive variable length coding (CAVLC) and context-based adaptive binary arithmetic coding (CABAC). The simpler entropy coding method is CAVLC for simple profile. It can save about 10% for the execution time under increasing about 7% bit-rate compared with CABAC. Because of bit-rate saving, CABAC is the superior scheme for massive capacity demand of the newest video application.

The compression efficiency improvement in CABAC is obtained at the cost of an inevitable complexity overhead. The results of the software-based complexity analysis are presented in [3], which claims that switching from CAVLC to CABAC usually leads to complexity increasing by 25% ~ 30% for encoding and 12% for decoding, in terms of access frequency (total number of memory transfers per second); therefore, both the coding acceleration and the cost efficiency promoting of CABAC are required.

We propose three throughput promoting methods to make CABAC encoder achieve the specification of 1080HD in level 4.1 stipulated in H.264/AVC standard [1]. For CABAC decoder, we continue using the high throughput CABAC decoder [2] which we presented in August 2006. It can achieve the specification of 1080HD in level 4.0. So our CABAC codec is sufficient to support 1080 HD video for real-time encoding and decoding at 30 fps. Besides, we also introduces some low cost methods such as finite state machine sharing, table reuse…etc to make the CABAC codec be better cost efficiency.

1.2 Organization of this thesis

This thesis is organized as follows. In Chapter 2, we present the algorithm of CABAC. It contains two levels coding procedure. For encoding, the first level is binarization engine, and the second level is arithmetic encoder. For decoding, the order of the two levels procedure is just opposite to encoding. Chapter 3 focuses on throughput promoting. We introduce three arithmetic encoding modes, and showing the proposed three high throughput methods. Chapter 4 focuses on cost efficiency designing. We present the proposed low cost methods and the memory requirement. In the end of this chapter, we show the proposed CABAC codec system architecture which has adopted the proposed low cost methods. At the final, the results of simulation and chip implementation will be shown in Chapter 5. We make a brief conclusion and future work in the last chapter.

Chapter 2

Algorithm of CABAC for H.264/AVC

In this chapter, we introduce the algorithm of CABAC encoding and decoding respectively. Both CABAC encoding and CABAC decoding are composed of three parts: the binarization process, the arithmetic coding process and the context model. For CABAC encoding, the binarization process reads syntax elements (SE), then computing the *bin* to offer the arithmetic encoding process for encoding the corresponding bit-streams. For CABAC decoding, the arithmetic decoding process reads the input bit-streams generated by H.264 encoder, and computing the *bin* to offer the binarization process for decoding the suitable SE. Both arithmetic encoding and arithmetic decoding have to look up context model which records the historical probability to compute the corresponding bit-streams in encoding and the *bin* value in decoding.

About the description of CABAC algorithm in this chapter, it is based on the content of [1] and [5], where the latter only focuses on decoding aspect, and the encoding aspect is introduced in addition in this chapter.

This chapter is organized as follows. In Section 2.1, we present the overviews of the CABAC encoding and decoding flow respectively, and show the two levels coding processe. In Section 2.2, we introduce all kinds of the binarization process such as the unary, the truncated unary, the fixed-length, Exp-Golomb and the defined code organization. In Section 2.3, the algorithm of basic binary arithmetic coding will be introduced briefly. We introduce it in terms of encoding and decoding respectively in the section 2.3.1 and the section 2.3.2. In Section 2.4, we present the advanced binary arithmetic coding for H.264/AVC, and relating it to arithmetic coding process. Section 2.5 shows the context model related to the different SEs. In final section, what we show is that how to get the neighbor SE to index the suitable context model allocation.

2.1 *Overview of CABAC encoding/decoding flow*

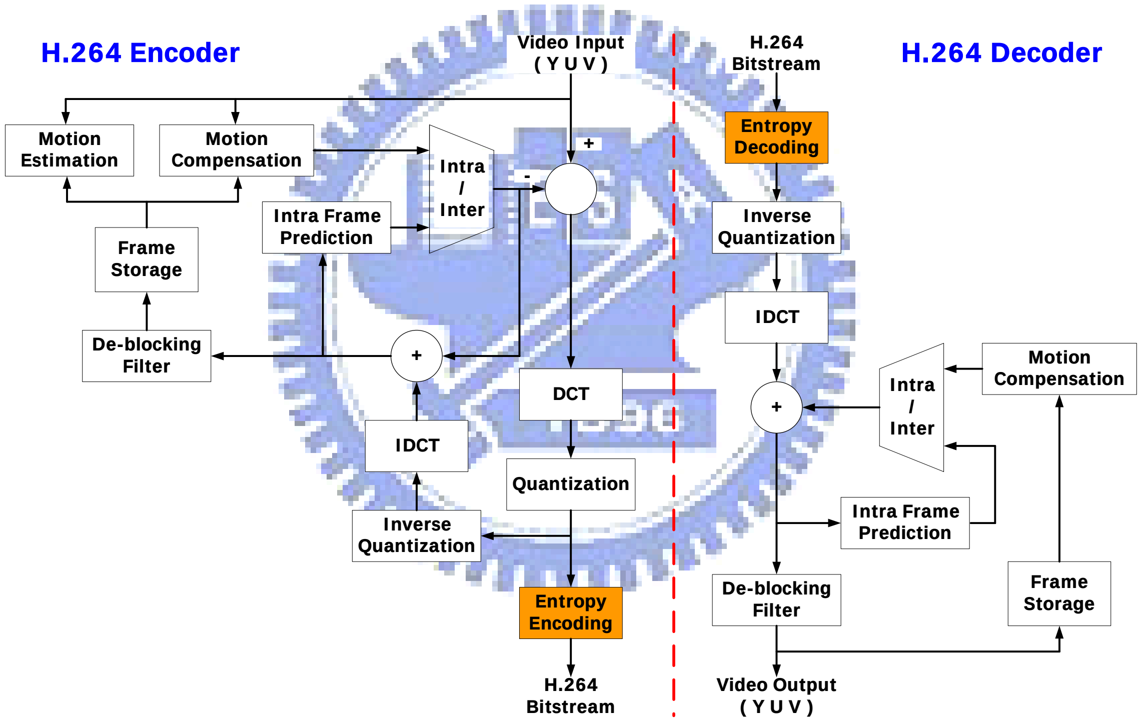


Figure 1 H.264/AVC encoder/decoder system block diagram

Figure 1 shows the system block diagram of H.264/AVC encoder and decoder. Both entropy encoder and entropy decoder contain three entropy coding strategies such as universal variable length coding (UVLC), context-based adaptive variable length coding (CAVLC) and context adaptive binary arithmetic coding (CABAC).

For H.264/AVC baseline profile, it only adopts UVLC and CAVLC two variable length coding (VLC) strategies to code the macroblock (MB) information and the pixels coefficients. UVLC is one of VLC in baseline profile, it codes not only the MB information such as the *mb_type*, *coded_block_pattern*, *intra_prediction_mode*, and so on, but also the MB coefficient such as *mvd*. Because the residual data coding occupies over 50% of the entire execution time, the residual coefficients are computed by the CAVLC architecture for more efficiency.

For H.264/AVC main profile, it has an advance choice except VLC. CABAC can be used in place of UVLC and CAVLC. Thus, H.264 system just needs CABAC to code all MB information and pixel data if entropy coding flag is assigned to CABAC.

In this section, we introduce the block diagram of CABAC encoder and decoder. Then, the execution flow of them will be introduced respectively meanwhile.

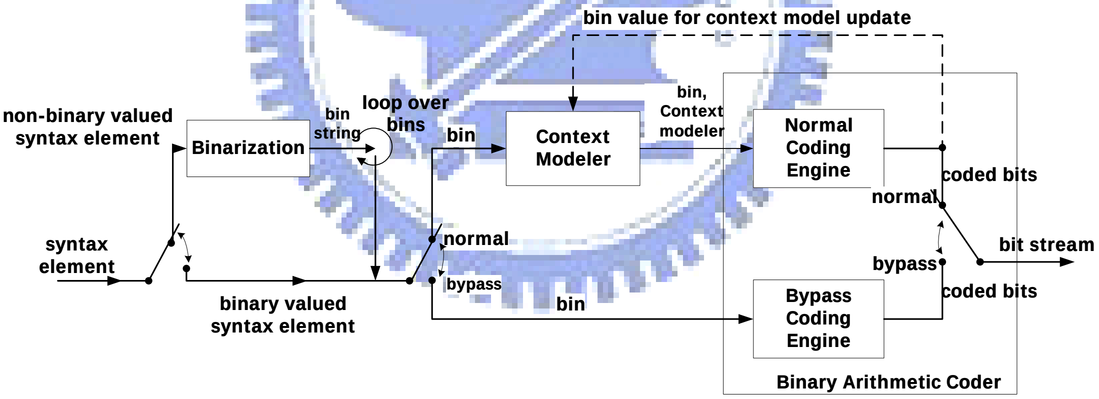


Figure 2 CABAC encoder flow chart [4]

Figure 2 shows the block diagram of CABAC encoder. We first see the left side of this figure. All syntax elements (SEs) of the H.264/AVC will be transferred into the binary code "*bin*" when entering the CABAC encoding process. Besides the SE of fixed-length coding type, all SEs have to be encoded by the binarization process which will be defined in Section 2.2. The transferred *bin* string is encoded to be the bit-stream by the binary arithmetic encoder. The binary arithmetic encoder has three different encoding types such as normal, bypass and terminal encoding processes. The terminal encoding process is seldom applied in CABAC system, which is only executed one time per macroblock (MB) encoding flow when the current MB is complete. So we ignore its influence due to its seldom applying opportunities. The normal and bypass encoding process are two main binary arithmetic coding modes. If it performs the bypass encoding process, there is no need to refer to the context model because the probability of bit-stream value is equal (probability = 0.5) between logical "1" and "0". If it applies the normal encoding process, it has to refer the associated context model depending on the SE type and the *bin* index.

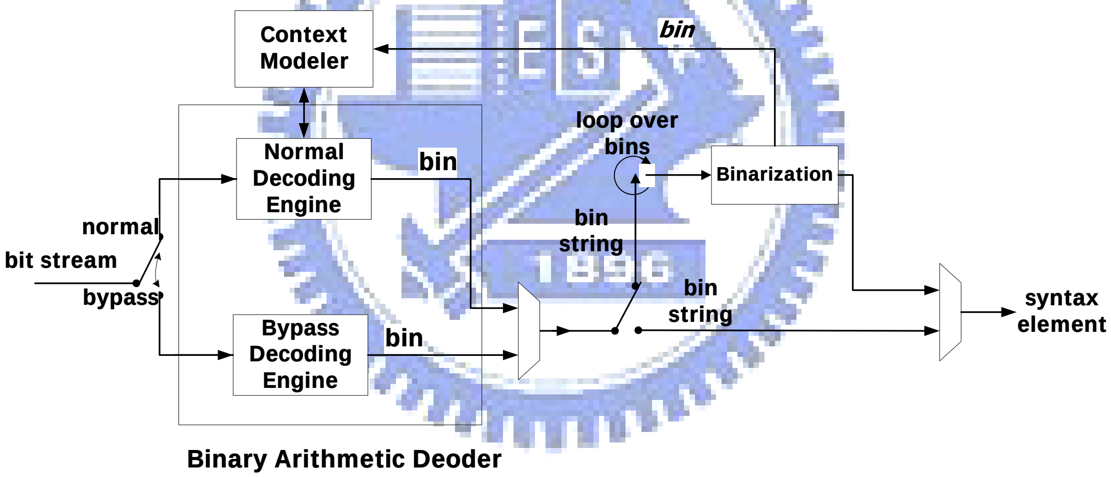


Figure 3 CABAC decoder flow chart

Figure 3 shows the block diagram of CABAC decoder. In H.264/AVC decoder, the decoding flow is contrary to CABAC encoder. At first, the binary arithmetic decoder reads the bit-stream and transfers it to be *bin* string. The binarization process reads the *bin* string and decodes it to be SE by five kinds of decoding flows. The execution sequences between CABAC encoder and decoder are just reverse. But the context modeler is still determined by binarization and SE for both CABAC encoder and CABAC decoder.

2.2 *Binarization process*

In Section 2.2, we focus on the binarization process. In CABAC encoder, it reads the syntax element and transfers it to be *bin* string. In CABAC decoder, it reads the *bin* string to look up the suitable syntax element. For H.264/AVC, both CABAC encoder and CABAC decoder adopt five kinds of the binarization methods to encode/decode the syntax element/*bin* string respectively. This section is organized as follows. In Section 2.2.1, the flow of the unary code is shown at first. The unary code is the basic coding method. Section 2.2.2 shows the truncated unary code which is the advanced unary coding method. In Section 2.2.3, we introduce the fixed-length coding flow. It is the typical binary integer method. Section 2.2.4 is the Exp-Golomb coding flow. The Exp-Golomb coding flow is only used for the residual data and the motion vector difference (*mvd*). Section 2.2.5 is the special definition which is by means of the table method. Specifically, we focus on the binary tree of the macroblock type (*mb_type*) and the sub-macroblock type (*sub_mb_type*).

2.2.1 Unary (U) binarization process

Input to this process is a request for a U binarization for a syntax element. Output of this process is the U binarization of the syntax element.

The bin string of a syntax element having value *synElVal* is a bit string of length *synElVal* + 1 indexed by *binIdx*. The *bin*s for *binIdx* less than *synElVal* are equal to “1”. The bin with *binIdx* equal to *synElVal* is equal to “0”.

Table 1 illustrates the *bin* strings of the unary binarization for a syntax element. If the syntax element is equal to “0”, the *bin* outputs single bit “0”. Except the syntax element “0”, the *bin* string sends *numSE* “1” and one “0” in the end of the binary value. The value of *numSE* is equal to the syntax element. Therefore, we find that the bin string length of current syntax element is *numSE* + 1.

Table 1 *bin* string of the unary binarization [1]

Syntax element	*bin* string					
0	0					
1	1	0				
2	1	1	0			
3	1	1	1	0		
4	1	1	1	1	0	
5	1	1	1	1	1	0
…	……					
binIdx	0	1	2	3	4	5

According to the unary *bin* string format shown above, we arrange the encoding and decoding algorithm in Eq. 1 and Eq. 2. These two equations represent the pseudo code of the unary encoding/decoding flow.

For unary encoding, it gets the *binIdx* which is the index of the bin string in Table 1 from the value of the input syntax element (*SEVal*). The for loop in Eq. 1 generates the *bin* string according to the *binIdx*. When finishing the for loop, it will generate one bit “0” as the end bit of the corresponding *bin* string. Namely, the unary binarization process arrives at the end of encoding step.

For unary decoding, it sets *binIdx* to zero at the initial step. The while loop in Eq. 2 checks the current bin assigned by *binIdx* from the bin string and *binIdx* counts if the current *bin* is equal to “1”. If the current *bin* is equal to “0”, the unary binarization process arrives at the end of decoding step. *binIdx* sends to *SEVal* which is defined as the value of the syntax element.

Start unary (U) encoding process

```
binIdx = SEVal;
for (i == 0; i < binIdx; i ++ ){
    bin = 1;
}
bin = 0;                                    (Eq. 1)
```

Start unary (U) decoding process

```
binIdx = 0;
while (bin[binIdx] == 1){
binIdx = binIdx + 1;
}
SEVal = binIdx;                             (Eq. 2)
```

2.2.2 Truncated unary (TU) binarization process

Input to this process is a request for a TU binarization for a syntax element and *cMax*. Output of this process is the TU binarization of the syntax element.

For syntax element values less than *cMax*, the U binarization process mentioned in Section 2.2.1 is invoked. For the syntax element value equal to *cMax* the *bin* string is a bit string of length *cMax* with all *bin*s being equal to “1”. TU binarization is always invoked with a *cMax* value equal to the largest possible value of the syntax element.

The truncated unary binarization process is based on the unary one and has an additional factor of *cMax* which is defined as the maximum length of the current *bin* string. If the value of syntax element (*SEVal*) is less than *cMax*, the truncated unary and the typical unary binarization process are the same. If *SEVal* is equal to *cMax*, the number “1” of the *bin* string is equal to *cMax* and there is no “0” bit in the end of the current string. For example, *SEVal*(=“4”) is assumed. If the value of *cMax* is “5”, the result of *bin* string is equal to “11110”. If the value of *cMax* is also “4”, the result of the *bin* string is equal to “1111” where the end bit of “0” is truncated in this case.

Eq. 3 is the truncated unary encoding flow which is modified from Eq. 1. Besides checking the value of syntax element (*SEVal*), it also takes *cMax* into consideration. If *SEVal* is less than *cMax*, it works as the unary encoding process. If *binIdx* isn’t less than *cMax*, it doesn’t generate “0” in the end of *bin* string when completing the encoding of current syntax element.

Start truncated unary (TU) encoding process

```
binIdx = SEVal;
for (i == 0; i < binIdx; i ++ ){
        bin = 1;
}
If (SEVal < cMax)   bin = 0;                                        (Eq. 3)
```

Eq. 4 is the truncated unary decoding flow which is modified from Eq. 2. Besides checking the *bin* value, it takes *cMax* as a factor, additionally. It works as the unary decoding process when *binsIdx* is less than *cMax*. If *binIdx* isn’t less than *cMax*, it doesn’t complete the decoding action until reading the end bit “0” in the end of the *bin* string.

Start truncated unary (TU) decoding process

```
binIdx = 0;
while (bin[binIdx] == 1 && (binIdx < cMax)){
binIdx = binIdx + 1;
}
SEVal = binIdx;                                                     (Eq. 4)
```

2.2.3 Fixed-length (FL) binarization process

Input to this process is a request for a FL binarization for a syntax element and *cMax*. Output of this process is the FL binarization of the syntax element.

FL binarization is constructed by using an fixedLength-bit unsigned integer *bin* string of the syntax element value. The indexing of *bin*s for the FL binarization is such that the *binIdx* = 0 relates to the least significant bit with increasing values of *binIdx* towards the most significant bit.

The fixed-length code is the simple-defined format of the binarization coding process which is defined as the typical unsigned integer. The coding rule is represented by means of the typical binary number. For example, the value of “5_{10}” is equal to “101_2”. The value of “5_{10}” is defined as the decimal style and the value of “101_2” is the binary format which is the required fixed-length code. Table 2 shows the fixed-length code definition.

Table 2 *bin* string of the fixed-length code

Syntax element	*bin* string					
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
…						
binIdx	0	1	2	3	4	5

For fixed-length encoding process, the value of input syntax element defines the *binIdx*. The value of (*binIdx* + 1) is just the required bit numbers for the corresponding bin string of the current syntax element.

For fixed-length decoding process, it has to refer to the value of *cMax* which defines the number size of the current syntax element. In Table 2, the *cMax* equals five because the maximum value of *binIdx* is five. All syntax elements which are decoded by the fixed-length format are always represented with six binary bits.

2.2.4 Unary/k-th order Exp_Golomb (UEGk) binarization process

Input to this process is a request for a UEGk binarization for a syntax element, *signedValFlag* and *uCoff*. Output of this process is the UEGk binarization of the syntax element.

A UEGk bin string is a concatenation of a prefix bit string and a suffix bit string. The prefix of the binarization is specified by invoking the TU binarization process for the prefix part [Min(*uCoff*, Abs(*synElVal*))] of a syntax element value *synElVal* as specified in Section 2.2.2 with *cMax* = *uCoff*, where *uCoff* > 0. Namely, the prefix part is dominated by *cMax*. The suffix part of this code doesn't always apply because it isn't adopted by two cases.

The UEGk bin string is derived as follows:

- If one of the following is true, the *bin* string of a syntax element having value *synElVal* consists only of a prefix bit string. Namely, the UEGk doesn't enter the suffix coding step.
    1. If *signedValFlag* is equal to 1 and the prefix *bin* contains only one 0 bit, the value of syntax element is just decided by prefix bin string with truncated unary (TU) code.
    2. If *signedValFlag* is equal to 0 and the prefix *bin* string isn't equal to the bit string which is composed of the string length *cMax* of bit 1.

- Otherwise, the bin string of the UEGk suffix part of a syntax element value synElVal is specified by a process equivalent to the following pseudo-code:

```
If( Abs( synE1Val) >= uCoff){
    sufS = Abs( synElVal) – uCoff
    stopLoop = 0
    do{
        if( sufS >= ( 1<<k)){
            put( 1 )
            sufS = sufS – ( 1<<k)
            k++
        }else{
            put( 0 )
            while( k--)
                put(( sufS >> k) & 0x01)
            stopLoop = 1
        }
    } while( !stopLoop)
}
If( signedValFlag && synElVal != 0)
    If( synElVal > 0)
        put( 0 )
    else
        put( 1 )                                                    (Eq. 5)
```

The initial value of k is defined as the order of the unary Exp-Golomb coding which are named as UEGk. In the binarization decoding of CABAC, it only applies two decoding flows such as UEG0 and UEG3. UEG0 is used by the residual data decoding process and UEG3 is used by the motion vector difference one.

2.2.5 Special binarization process

Input to this process is a request for a binarization for syntax element mb_type or sub_mb_type. Output of this process is the binarization of these two syntax elements.

All formats of the binarization coding process are introduced above. But there is still a special coding flow which we don't describe yet. In order to perform the higher video quality, the macroblock and sub-macroblock are divided into many kinds of types such as I, P, B, and SI slices. In the four basic types, they are also sorted by variable block sizes. These two syntax elements are difficult to define by means of the aforementioned coding flows. In H.264/AVC, it adopts the table-based method to define the macro and sub-macroblock types. The binarization engine reads the *bin* string and checks if the *bin* string is mapped the specified location in these tables. If the assigned *bin* string is found in these tables, it can look up the current macroblock type.

The binarization scheme for coding of macroblock type in I slice is specified in Table 3 [1]. For example, if the value of *bin* string is equal to "1001011" in I slice, the mapped macroblock type is equal to "8" by look up Table 3. We observe that the probability of the macroblock type appearance is large and its corresponding *bin* string is shorter.

For macroblock types in SI slices, the binarization consists of *bin* strings specified as a concatenation of a prefix and a suffix bit string as follows.

The prefix bit string consists of a single bit, which is specified by $b_0 = ((\text{mb\_type} = = \text{SI}) ? \ 0 : 1)$. For the syntax element value for which b_0 is equal to 1, the binarization is given by concatenating the prefix b_0 and the suffix bit string as specified in Table 3 for macroblock type in I slices indexed by subtracting 1 from the value of mb_type in SI slices.

Table 3 Binarization for macroblock types in I slice

Value (name) of mb_type	Bin string						
0 (I_4x4)	0						
1 (I_16x16_0_0_0)	1	0	0	0	0	0	
2 (I_16x16_1_0_0)	1	0	0	0	0	1	
3 (I_16x16_2_0_0)	1	0	0	0	1	0	
4 (I_16x16_3_0_0)	1	0	0	0	1	1	
5 (I_16x16_0_1_0)	1	0	0	1	0	0	0
6 (I_16x16_1_1_0)	1	0	0	1	0	0	1
7 (I_16x16_2_1_0)	1	0	0	1	0	1	0
8 (I_16x16_3_1_0)	1	0	0	1	0	1	1
9 (I_16x16_0_2_0)	1	0	0	1	1	0	0
10 (I_16x16_1_2_0)	1	0	0	1	1	0	1
11 (I_16x16_2_2_0)	1	0	0	1	1	1	0
12 (I_16x16_3_2_0)	1	0	0	1	1	1	1
13 (I_16x16_0_0_1)	1	0	1	0	0	0	
14 (I_16x16_1_0_1)	1	0	1	0	0	1	
15 (I_16x16_2_0_1)	1	0	1	0	1	0	
16 (I_16x16_3_0_1)	1	0	1	0	1	1	
17 (I_16x16_0_1_1)	1	0	1	1	0	0	0
18 (I_16x16_1_1_1)	1	0	1	1	0	0	1
19 (I_16x16_2_1_1)	1	0	1	1	0	1	0
20 (I_16x16_3_1_1)	1	0	1	1	0	1	1
21 (I_16x16_0_2_1)	1	0	1	1	1	0	0
22 (I_16x16_1_2_1)	1	0	1	1	1	0	1
23 (I_16x16_2_2_1)	1	0	1	1	1	1	0
24 (I_16x16_3_2_1)	1	0	1	1	1	1	1
25 (I_PCM)	1	1					
binIdx	0	1	2	3	4	5	6

In other words, the macroblock in SI slice is the enhanced format of the macroblock type in I slice. The *bin* string of SI slice macroblock type is composed of two parts: prefix bit and the suffix part. If the prefix bit is equal to 0, it doesn't need the suffix part and the syntax element is equal to 0. If the prefix bit is equal to 1, the suffix part is defined in Table 3 and the mapped value of macroblock types has to be added by 1 in SI slice.

Besides the macroblock type in SI slice, there are still two cases that they generate the *bin* value through the suffix process. The binarization schemes for P macroblock types in P and SP slices and B macroblocks in B slices are specified in Table 4 [1]. Table 4 is the prefix definitions of mb_type. The suffix parts are decided by the formats in Table 3 because these fields are the intra macroblock type in P, SP, and B slice. But the value of the macroblock type in the last fields in P and B slice have to be added by offset value.

The *bin* string for I macroblock types in P and SP slices corresponding to mb_type value 5 to 30 consists of a concatenation of a prefix, which consists of a single bit with value equal to 1 as specified in Table 4 and a suffix as specified in Table 3, indexed by subtracting 5 from the value of my_type. Namely in the P and SP slices, the value of mb_type is equal to the summation of the offset value "5" and the corresponding value of mb_type for current *bin* string in Table 3 if the prefix bit of the current *bin* string is equal to "1".

The bin string for I macroblock types in B slices (mb_type values 23 to 48) the binarization consists of *bin* strings specified as a concatenation of a prefis bit string as specified in Table 4 and the suffix bit strings as specified in Table 3, indexed by subtracting 23 from the value of mb_type. Namely in the B slices, the value of mb_type is equal to the summation of the offset value "23" and the corresponding value of mb_type for current *bin* string in Table 3 if the prefix bits of the current *bin* string is equal to "111101".

Table 4 Binarization for macroblock types in P, SP, and B slices

Slice type	Value (name) of mb_type	Bin string						
P, SP slice	0 (P_L0_16x16)	0	0	0				
	1 (P_L0_L0_16x8)	0	1	1				
	2 (P_L0_L0_8x16)	0	1	0				
	3 (P_8x8)	0	0	1				
	4 (P_8x8ref0)	na						
	5 to 30 (Intra, prefix only)	1						
B slice	0 (B_Direct_16x16)	0						
	1 (B_L0_16x16)	1	0	0				
	2 (B_L1_16x16)	1	0	1				
	3 (B_Bi_16x16)	1	1	0	0	0	0	
	4 (B_L0_L0_16x8)	1	1	0	0	0	1	
	5 (B_L0_L0_8x16)	1	1	0	0	1	0	
	6 (B_L1_L1_16x8)	1	1	0	0	1	1	
	7 (B_L1_L1_8x16)	1	1	0	1	0	0	
	8 (B_L0_L1_16x8)	1	1	0	1	0	1	
	9 (B_L0_L1_8x16)	1	1	0	1	1	0	
	10 (B_L1_L0_16x8)	1	1	0	1	1	1	
	11 (B_L1_L0_8x16)	1	1	1	1	1	0	
	12 (B_L0_Bi_16x8)	1	1	1	0	0	0	0
	13 (B_L0_Bi_8x16)	1	1	1	0	0	0	1
	14 (B_L1_Bi_16x8)	1	1	1	0	0	1	0
	15 (B_L1_Bi_8x16)	1	1	1	0	0	1	1
	16 (B_Bi_L0_16x8)	1	1	1	0	1	0	0
	17 (B_Bi_L0_8x16)	1	1	1	0	1	0	1
	18 (B_Bi_L1_16x8)	1	1	1	0	1	1	0
	19 (B_Bi_L1_8x16)	1	1	1	0	1	1	1
	20 (B_Bi_Bi_16x8)	1	1	1	1	0	0	0
	21 (B_Bi_Bi_8x16)	1	1	1	1	0	0	1
	22 (B_8x8)	1	1	1	1	1	1	
	23 to 48 (Intra, prefix only)	1	1	1	1	0	1	
binIdx		0	1	2	3	4	5	6

Table 5 Binarization for sub-macroblock types in P, SP, and B slices

Slice type	Value (name) of sub_mb_type	Bin string					
P, SP slice	0 (P_L0_8x8)	1					
	1 (P_L0_8x4)	0	0				
	2 (P_L0_4x8)	0	1	1			
	3 (P_L0_4x4)	0	1	0			
B slice	0 (B_Direct_8x8)	0					
	1 (B_L0_8x8)	1	0	0			
	2 (B_L1_8x8)	1	0	1			
	3 (B_Bi_8x8)	1	1	0	0	0	
	4 (B_L0_8x4)	1	1	0	0	1	
	5 (B_L0_4x8)	1	1	0	1	0	
	6 (B_L1_8x4)	1	1	0	1	1	
	7 (B_L1_4x8)	1	1	1	0	0	0
	8 (B_Bi_8x4)	1	1	1	0	0	1
	9 (B_Bi_4x8)	1	1	1	0	1	0
	10 (B_L0_4x4)	1	1	1	0	1	1
	11 (B_L1_4x4)	1	1	1	1	0	
	12 (B_Bi_4x4)	1	1	1	1	1	
binIdx		0	1	2	3	4	5

For P, SP, and B slices the specification of the binarization for sub_mb_type is given in Table 5. Instead of looking up the prefix part in one table and looking up the suffix part in other table, it only takes Table 5 to process the binarization of sub-macroblock types in P, SP, and B slices.

2.3 *Algorithm of basic binary arithmetic coding*

In this section, we introduce the basic arithmetic encoding and decoding algorithm to understand the organization of the arithmetic code. It is the basic concept to the advanced binary arithmetic algorithm in Section 2.4.

2.3.1 *Basic binary arithmetic encoding algorithm*

This section introduces the basic arithmetic encoding algorithm to understand the binary arithmetic coding algorithm and know how to decode the bit-stream which is generated by encoder. According to the probability, the binary arithmetic encode defines two sub-intervals in the current range. The two sub-intervals are named as MPS (Most Probable Symbol) and LPS (Least Probable Symbol). Figure 4(a) shows the definition of the sub-intervals. The lower part is MPS and the upper one is LPS. The range value of MPS is defined as *rMPS* and the one of LPS is defined as *rLPS*. The ranges of the current MPS and LPS are defined in Eq. 6. In this equation, ρ_{MPS} and ρ_{LPS} are the probability of MPS and LPS. The summation of LPS and MPS is equal to one because the probability of the current interval is one.

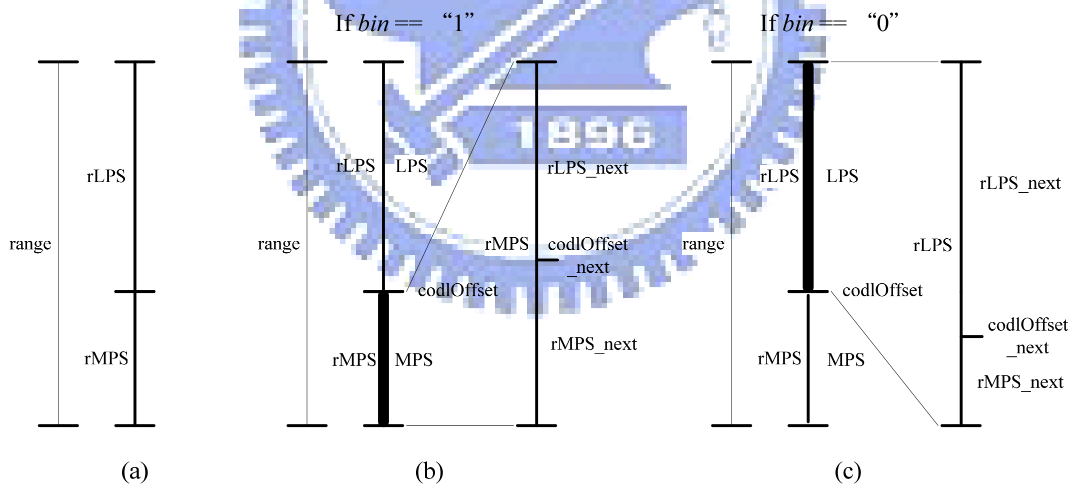


Figure 4 (a) Definition of MPS and LPS and

(b) Sub-divided interval of MPS and

(c) Sub-divided interval of LPS

$rMPS = range \times \rho_{MPS}$

$rLPS = range \times \rho_{LPS} = range \times (1 - \rho_{MPS})$

ρ_{MPS} : The probability of MPS

ρ_{LPS} : The probability of LPS (Eq. 6)

Depending on the *bin* decision, it identifies as either MPS or LPS. If *bin* is equal to “1”, the next interval belongs to MPS. Figure 4(b) shows the MPS sub-interval condition and the lower part of the current interval is the next one. The range of the next interval is re-defined as *rMPS* and ρ_{MPS} is increased. On the contrary, the next current interval belongs to LPS when *bin* is equal to “0”. Figure 4(c) shows the LPS sub-interval condition and the upper part of the current interval is the next one. The range of the next interval is re-defined as *rLPS* and ρ_{MPS} is decreased.

We arrange the algorithm in the following Eq. 7 and Eq. 8.

Most probable symbol (MPS) condition:

The MPS probability of the next interval: $\rho_{MPS\_NEXT} = \rho_{MPS} + \rho_{Inc}$

The range of the next interval: $range_{NEXT} = rMPS$

The value of the next interval: $codlOffset_{NEXT} = rMPS \times \rho_{MPS\_NEXT}$

ρ_{Inc} : The increment of ρ_{MPS} . (Eq. 7)

Least probable symbol (LPS) condition:

The MPS probability of the next interval: $\rho_{MPS\_NEXT} = \rho_{MPS} - \rho_{Dec}$

The range of the next interval: $range_{NEXT} = rLPS$

The value of the next interval: $codlOffset_{NEXT} = codlOffset + rLPS \times \rho_{MPS\_NEXT}$

ρ_{Dec} : The decrement of ρ_{MPS} . (Eq. 8) *codlOffset* is allocated at the intersection between the current MPS and LPS range. Depending on *codlOffset*, the arithmetic encoder produces the bit-stream in order to achieve the compression effect.

2.3.2 Basic binary arithmetic decoding algorithm

In the binary arithmetic decoder, it decompresses the bit-stream to the *bin* value which offers the binarization to restore the syntax elements. The decoding process is similar to the encoding one. Both of them are executed by means of the recursive interval subdivision. But they still have some different coding flow, which is described as follows.

It is needed to define the initial range and the MPS probability ρ_{MPS} when starting the binary arithmetic decode. The value of *codlOffset* is composed of the bit-stream and compared with *rMPS*. The MPS and LPS conditions are unlike the definitions of the encoder. Figure 5 illustrates the subdivision of the MPS and LPS condition. If *codlOffset* is less than *rMPS*, the condition belongs to MPS. The range of the next interval is equal to *rMPS*. The probability of MPS (ρ_{MPS}) is increased and the *bin* value outputs "1". The next value of *codlOffset* remains the current one. Figure 5(a) illustrates the MPS condition. If *codlOffset* is great than or equal to *rMPS*, the next interval turns into LPS. The range of the next interval is defined as *rLPS*. The probability of MPS (ρ_{MPS}) is decreased and the *bin* value outputs "0". The meaning of the next value of *codlOffset* is to subtract the *rMPS* from the current *codlOffset*. Figure 5(b) illustrates the MPS condition.

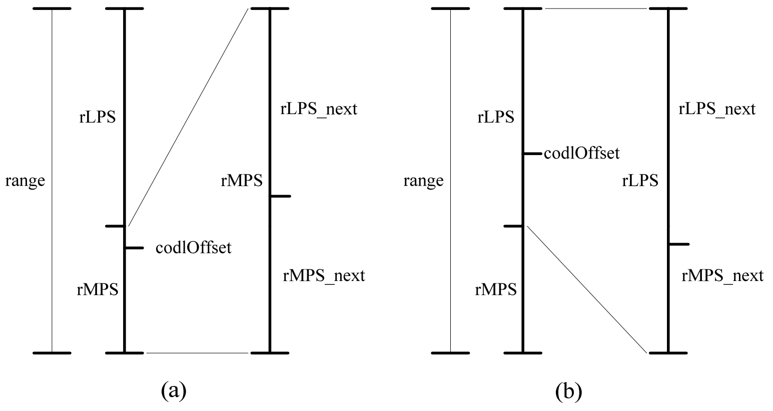


Figure 5 (a) Result of MPS subdivision and

(b) Result of LPS subdivision

We arrange the algorithm in the following Eq. 9 and Eq. 10.

Most probable symbol (MPS) condition: If (*codlOffset* < *rMPS*)

The *bin* Value = “1”

The value of the next *codlOffset*: $codlOffset_{NEXT} = codlOffset$

The MPS probability of the next interval: $\rho_{MPS\_NEXT} = \rho_{MPS} + \rho_{Inc}$

The range of the next interval: $range_{NEXT} = rMPS$

ρ_{Inc} : The increment of ρ_{MPS} . (Eq. 9)

Least probable symbol (LPS) condition: If(*codlOffset* >= *rMPS*)

The value of the next *codlOffset*: $codlOffset_{NEXT} = codlOffset - rMPS$

The MPS probability of the next interval: $\rho_{MPS\_NEXT} = \rho_{MPS} - \rho_{Dec}$

The range of the next interval: $range_{NEXT} = rLPS$

ρ_{Dec} : The decrement of ρ_{MPS} . (Eq. 10)

2.4 *Advanced binary arithmetic coding for H.264/AVC*

According to the H.264/AVC standard [1], we introduce the advanced binary arithmetic algorithm adopted by CABAC in this section. It makes more efficient with the integer operation for range computation by means of multiplication-free and table-based probability architecture.

2.4.1 Advanced binary arithmetic encoding algorithm

In Section 2.3.1, we introduce the basic algorithm of the binary arithmetic encode. Although it can achieve the high compression gain, the algorithm works under the floating-point operation. The hardware complexity becomes the problem when we implement the binary arithmetic encoder. In Eq. 6, it has to compute the values of *rMPS* and *rLPS* with two multipliers and processes the next value of *codlOffset*, *range*, and the probability by means of the floating adders and comparators. It consumes the lots hardware cost because the multipliers and floating operations make the complex circuit. According to H.264/AVC standard [1], we adopt the low complexity algorithm to implement the CABAD circuit.

In order to improve the coding efficiency, there are three kinds of the binary arithmetic decoders in H.264/AVC system such as the normal, bypass, and termination encoding flow. We will show whole algorithms as follows.

The first algorithm is the normal encoding process which is shown in Figure 6. There are two main factors to dominate the hardware efficiency. One is the multiplier of $range \times \rho_{LPS}$ defined as *rLPS* and the other is the probability calculation defined as ρ_{LPS} . In Eq. 6, it applies one multiplier to find the range of LPS (*rLPS*). According to the H.264/AVC standard, the table-based method is used in place of the multiplication operation. In the normal decoding flowchart, *codlRangeLPS* looks up the table, rangeTabLPS, depending on two indexes such as *pStateIdx* and *qCodlRangeIdx*. *pStateIdx* is defined as the probability of MPS (ρ_{MPS}) which gets from the context model. *qCodlRangeIdx* is the quantized value of the current range (*codlRange*) which is separated to four parts in this table. The second factor of the improved method is to estimate the value of ρ_{MPS} . In Section 2.3.1, we know that the value of ρ_{MPS} is increased when MPS condition happened and is decreased when LPS condition happened. But we can't find the rule how much the value has to be increased or decreased.

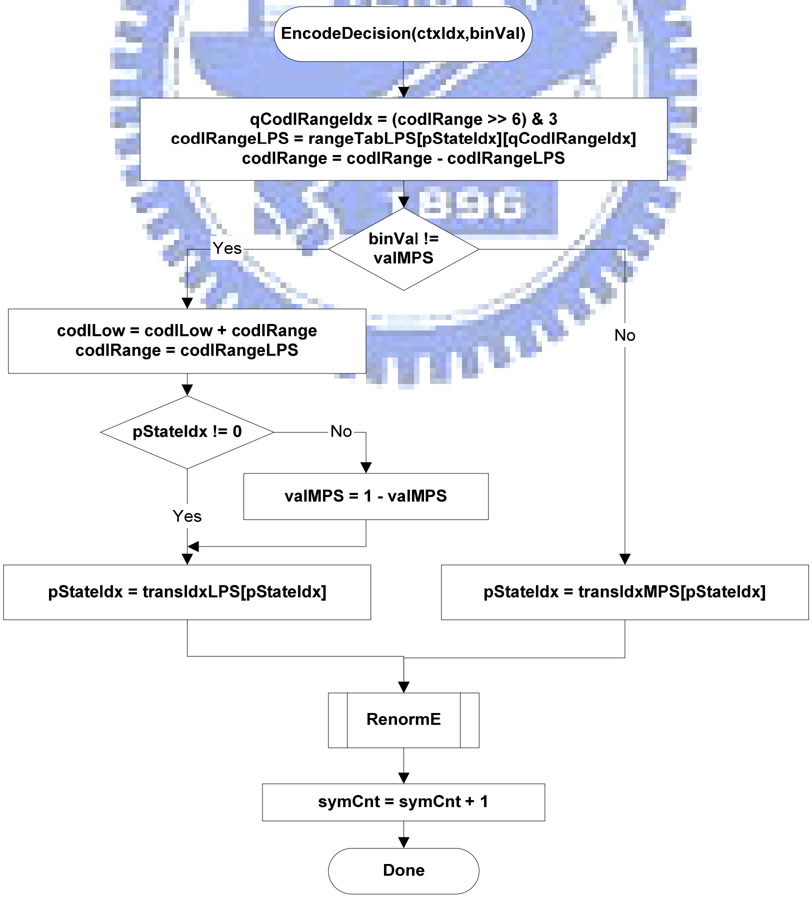

EncodeDecision(ctxIdx,binVal)
qCodlRangeIdx = (codlRange >> 6) & 3
codlRangeLPS = rangeTabLPS[pStateIdx][qCodlRangeIdx]
codlRange = codlRange - codlRangeLPS
binVal != valMPS
Yes
No
codlLow = codlLow + codlRange
codlRange = codlRangeLPS
pStateIdx != 0
No
valMPS = 1 - valMPS
Yes
pStateIdx = transIdxLPS[pStateIdx]
pStateIdx = transIdxMPS[pStateIdx]
RenormE
symCnt = symCnt + 1
Done

Figure 6 Flowchart of the normal encoding flow [1]

The flowchart of Figure 6 also shows the table-based method to process the probability estimation. It divides into two sub-intervals such as MPS and LPS conditions. Depending on the sub-interval, it computes the next probability by the *transIdxLPS* table when the interval division is LPS and by the *transIdxMPS* table when the interval is MPS. These two probability tables are approximated by sixty-four quantized values indexed by the probability of the current interval.

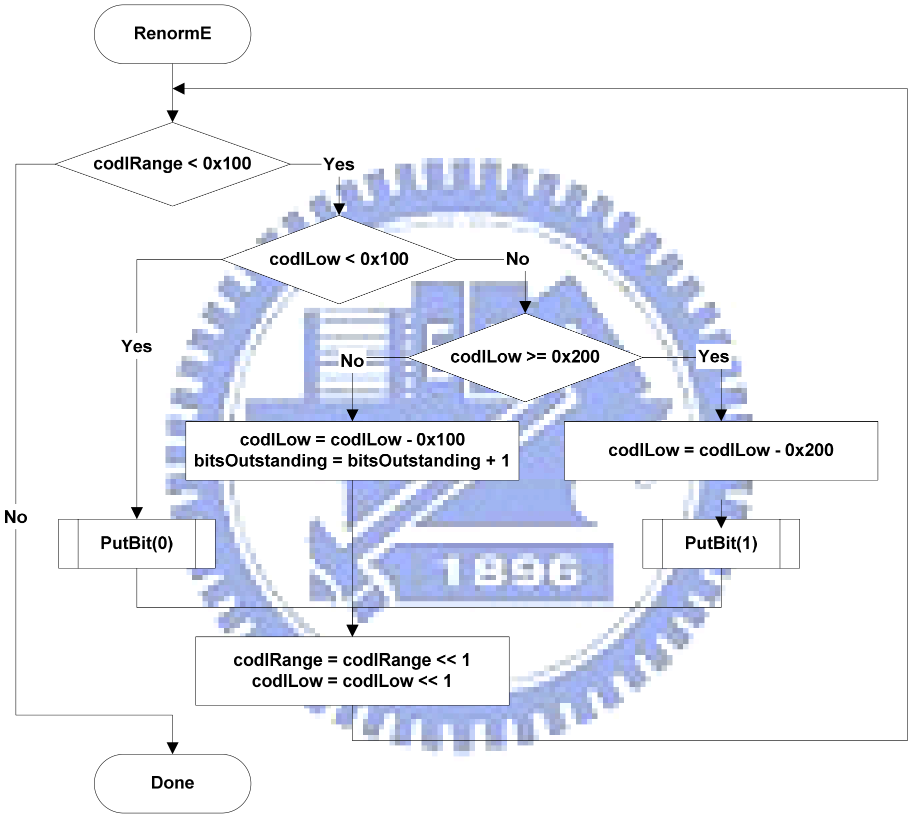


Figure 7 Flowchart of renormalization in the encoder [1]

In the basic binary arithmetic encoding, the interval subdivision is operated under the floating-point operation. In practical implementation, this method causes the complexity of the circuit to be increased. The advanced algorithm adopts the integer operation for H.264/AVC. The value of the next range becomes smaller than the current interval. So we use the renormalization method to keep the scales of *codlRange* and *codlLow*. Figure 7 shows the flowchart of renormalization in the encoder. The MSB of *codlRange* always keeps "1" in order to realize the integer operation. If the MSB of *codlRagne* is equal to "0", the value of *codlRagne* has to be shifted left until the current bit is equal to "1". The *codlLow* follows the *codlRange* to shift left synchronously. Depending on the shifted number of *codlRagne*, it dominates the number of iteration for *codlLow* compute directly and affects the PutBit procedure indirectly.

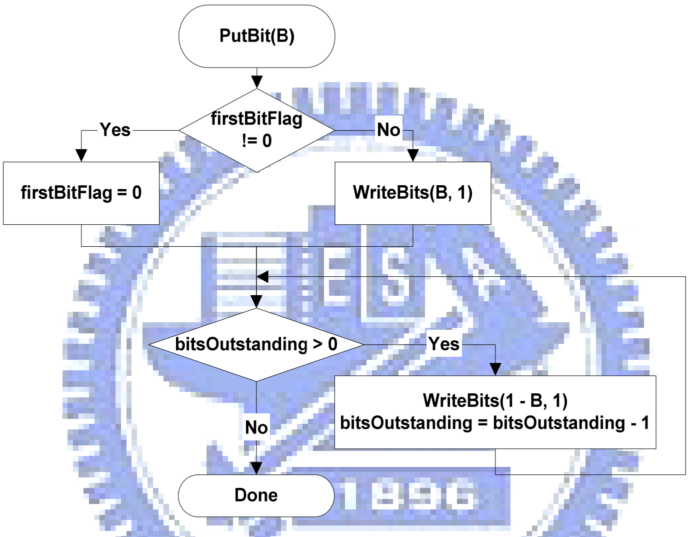


Figure 8　　Flowchart of PutBit(B) [1]

Figure 8 shows the flowchart of PutBit procedure. It is dominated by the *codlLow* decision branch and the bitsOutstanding accumulated in renormalization procedure. The *codlLow* decision branch dominates its leading bit of one piece bit-stream for current *codlLow*. Besides dominating the leading bit, the *codlLow* decision branch also controls if the bitsOutstanding should be accumulated. The accumulated bitsOutstanding dominates the number of bit will be generated in one piece bit-stream for current *codlLow*. For example, if the current *codlLow* is 01_xxxx_xxxx, and the current accumulated bitsOutstanding is 5, the bit-stream will not be generated. Instead of generating bit-stream, the bitsOutstanding will be added 1. So the next bitsOutstanding is 6. If the current *codlLow* is 11_xxxx_xxxx, and the current accumulated bitsOutstanding is 5, the generated bit-stream in PutBit procedure is 10_0000. The length of the generated bit-stream is variable decided by the accumulated value of bitsOutstanding. The accumulation of bitsOutstanding restricts the throughput of the whole arithmetic encoder. The proposed method of lowering its affection in throughput will be introduced in Chapter 3. Besides, the accumulation of bitsOutstanding also leads to the large range in the length of generated variable bit-stream. It causes the cost issue in output interface design. The proposed method of cost efficiency in output interface will also be introduced in Chapter 3.

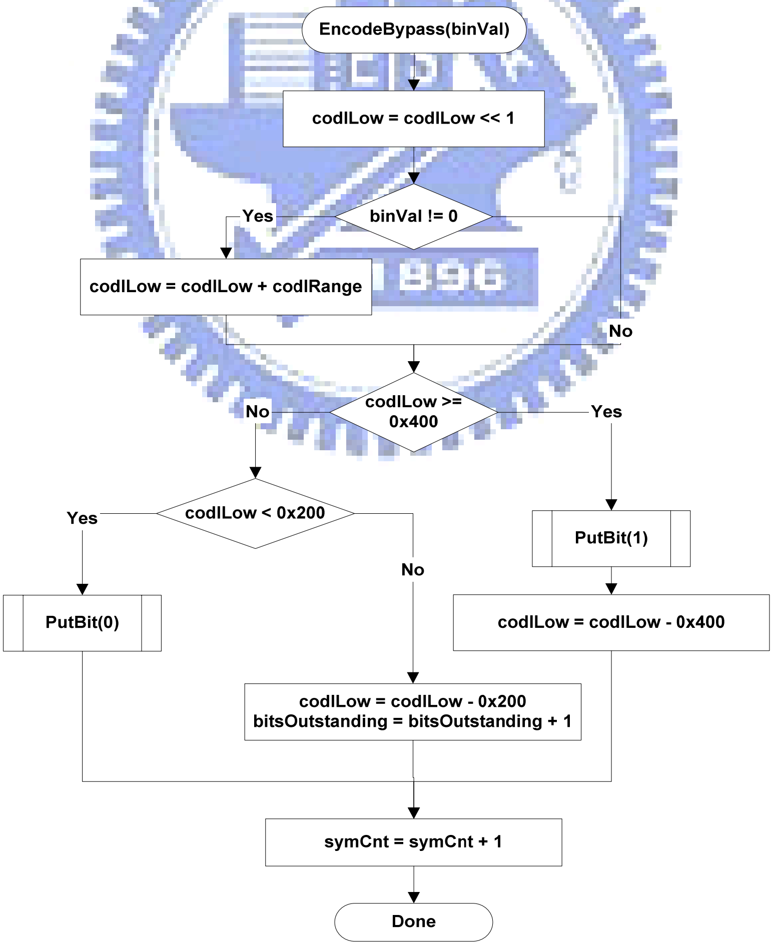


Figure 9 Flowchart of encoding bypass [1]

The second algorithm is the bypass encoding process which is applied by the specified syntax elements such as *abs_mvd*, *significant_coeff_flag*, *last_significant_coeff_flag*, and *coeff_abs_level_minus1*. The probabilities of MPS and LPS are fair; therefore, both probabilities are 0.5. It is unnecessary to refer to the context model during decoding. Figure 9 shows the flowchart of the bypass encoding flow. Compared with Figure 6, the bypass encoding process doesn't estimate the probability of the next interval. So we don't see the probability computation in the bypass encoding. The computed *codlRange* doesn't change which means that it has no renormalization in the bypass decoding. But the *codlLow* decision branch in bypass encoding is similar to the one in renormalization, so we can modify it to be hardware sharing with renormalization in hardware implementation. Besides, it just uses one adder and one no iteration renormalization to implement the encoding process. So we will adopt the multi-symbol architecture based on the result of statistic in several image test sequences to speed up the throughput of the CABAC system due to its simple process. The proposed multi-symbol architecture will be shown in Chapter 3.

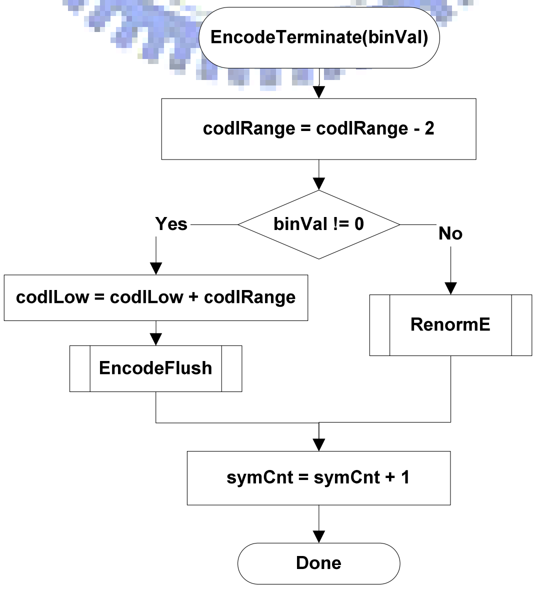


Figure 10　　Flowchart of the terminal encoding flow [1]

The third algorithm is the termination encoding process. Figure 10 shows the flowchart of the terminal encoding flow. The terminal encoding process is simple as well, but it has the more encoding procedure compared to the bypass encoding process. It doesn't need the context model to refer to the probability. No matter the subdivision condition belongs to MPS or LPS, the value of the next *codlRange* is always to subtract two from the current *codlRange* at first. When the subdivision condition is LPS, the *EncodeFlush* process shown at figure 11 has to be executed. The purpose of *EncodeFlush* is stuffing several bits to divide the bit-stream of current macroblock and the bit-stream of next macroblock. In the *EncodeFlush* process, the *codlRange* is always assigned to be the constant value which is two at first. For whole termination process, both MPS process and LPS process have to execute renormalization process. In the MPS process, it only executes the renormalization process. In the LPS process, it has to perform two steps process. The first step is to update the *codlLow* value by taking original *codlLow* to add the initial *codlRange* which has been subtracted two.

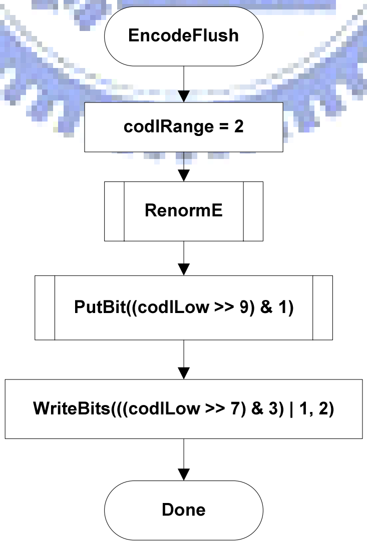

EncodeFlush
codlRange = 2
RenormE
PutBit((codlLow >> 9) & 1)
WriteBits(((codlLow >> 7) & 3) | 1, 2)
Done

Figure 11 Flowchart of flushing at termination [1]

The termination process occurs one time per macroblock encoding process, it is seldom used during all encoding processes. Therefore, it affects slightly in the throughput of whole CABAC encoding system. So we will focus on the first two algorithms in this section.

2.4.2 Advanced binary arithmetic decoding algorithm

In Section 2.3.2, we introduce the basic algorithm of the binary arithmetic decode. The drawbacks of it is the same as the basic algorithm of the binary arithmetic encode, it also leads to more complexity in hardware implementation due to the multipliers and floating-point operations. As arithmetic algorithm used in CABAC encoding, the arithmetic decoding also adopts three kinds of decoding modes such as the normal, bypass, and termination decoding flow to lower its complexity in circuit implementation. We will show these three decoding algorithms as follows.

The first algorithm is the normal decoding process which is shown in Figure 12. As advanced binary arithmetic encoding algorithm, there are also two main factors to dominate the hardware efficiency for the advanced binary arithmetic decoding algorithm. One is the multiplier of $range \times \rho_{LPS}$ defined as *rLPS* and the other is the probability calculation defined as ρ_{LPS}. In Eq. 6, it applies one multiplier to find the range of LPS (*rLPS*). According to the H.264/AVC standard, the table-based method is used in place of the multiplication operation. In the normal decoding flowchart, *codlRangeLPS* looks up the table, rangeTabLPS, depending on two indexes such as *pStateIdx* and *qCodlRangeIdx*. *pStateIdx* is defined as the probability of MPS (ρ_{MPS}) which gets from the context model. *qCodlRangeIdx* is the quantized value of the current range (*codlRange*) which is separated to four parts in this table. The second factor of the improved method is to estimate the value of ρ_{MPS}. In Section 2.3.2, we know that the value of ρ_{MPS} is increased when MPS condition happened and is decreased when LPS condition happened. But we can’t find the rule how much the value has to be increased or decreased. The flowchart of Figure 12 also shows the table-based method to process the probability estimation. It divides into two sub-intervals such as MPS and LPS conditions. Depending on the sub-interval, it computes the next probability by the *transIdxLPS* table when the interval division is LPS and by the *transIdxMPS* table when the interval is MPS. These two probability tables are approximated by sixty-four quantized values indexed by the probability of the current interval.

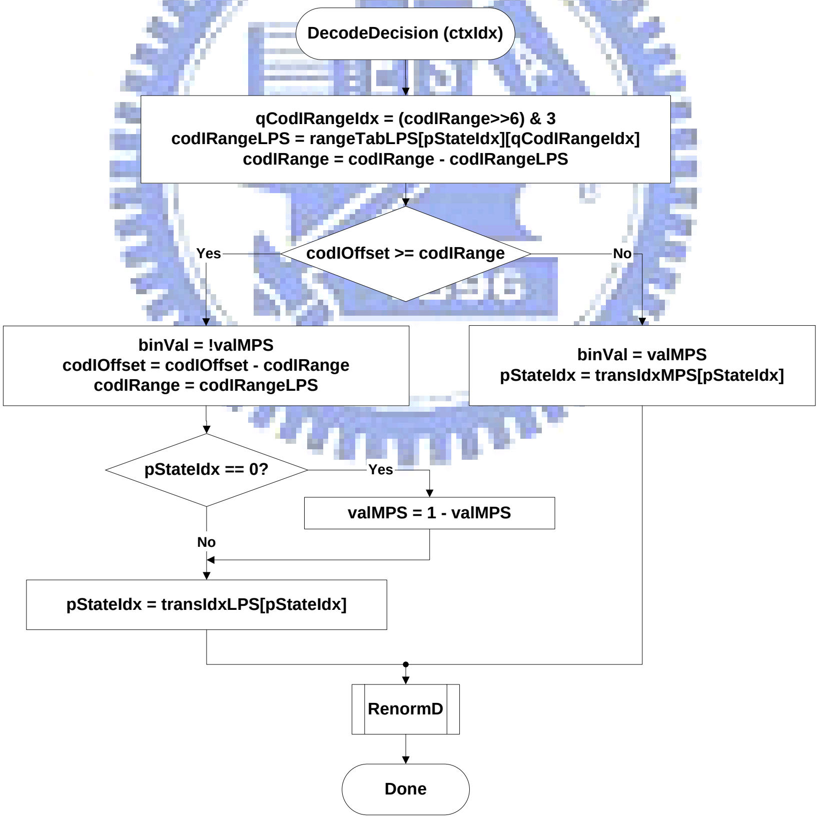

DecodeDecision (ctxIdx)
qCodIRangeIdx = (codIRange>>6) & 3
codIRangeLPS = rangeTabLPS[pStateIdx][qCodIRangeIdx]
codIRange = codIRange - codIRangeLPS
codIOffset >= codIRange
Yes
No
binVal = !valMPS
codIOffset = codIOffset - codIRange
codIRange = codIRangeLPS
binVal = valMPS
pStateIdx = transIdxMPS[pStateIdx]
pStateIdx == 0?
Yes
valMPS = 1 - valMPS
No
pStateIdx = transIdxLPS[pStateIdx]
RenormD
Done

Figure 12 Flowchart of the normal decoding flow [1]

In the basic binary arithmetic decoder, the interval subdivision is operated under the floating-point operation. In practical implementation, this method causes the complexity of the circuit to be increased. The advanced algorithm adopts the integer operation for H.264/AVC. The value of the next range becomes smaller than the current interval. So we use the renormalization method to keep the scales of *codlRange* and *codlOffset*. Figure 13 shows the flowchart of renormalization. The MSB of *codlRange* always keeps "1" in order to realize the integer operation. If the MSB of *codlRagne* is equal to "0", the value of *codlRagne* has to be shifted left until the current bit is equal to "1". Depending on the shifted number of *codlRagne, codlOffset* fill the bit-stream in the LSB.

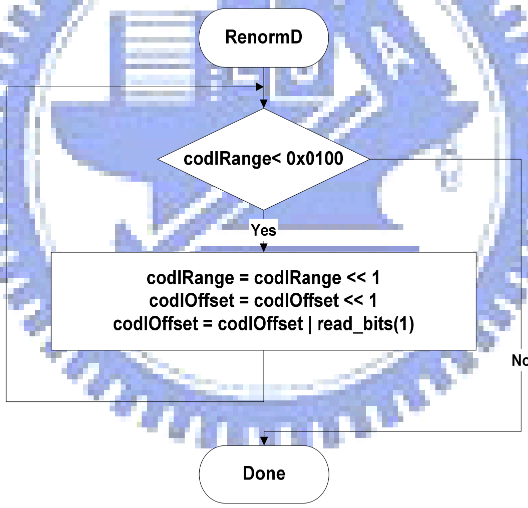

RenormD
codlRange< 0x0100
Yes
codlRange = codlRange << 1
codlOffset = codlOffset << 1
codlOffset = codlOffset | read_bits(1)
No
Done

Figure 13 Flowchart of renormalization in the decoder [1]

The second algorithm is the bypass decoding process which is applied by the specified syntax elements such as *abs_mvd*, *significant_coeff_flag*, *last_significant_coeff_flag*, and *coeff_abs_level_minus1*. The probabilities of MPS and LPS are fair, that is, both probabilities are 0.5. It is unnecessary to refer to the context model during decoding. Figure 14 shows the flowchart of the bypass decoding flow.

Compared with Figure 12, the bypass decoding process doesn't estimate the probability of the next interval. So we can't see the probability computation in the bypass decoding. The computed *codlRange* doesn't change which means that it has no renormalization in the bypass decoding. It just uses one subtraction to implement this decoding process. This algorithm is very simple, so we will use the architecture to speed up the CABAD system.

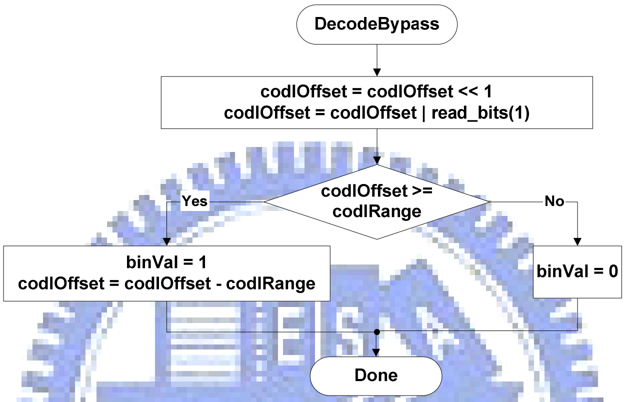

DecodeBypass
codIOffset = codIOffset << 1
codIOffset = codIOffset | read_bits(1)
codIOffset >= codIRange
Yes
No
binVal = 1
codIOffset = codIOffset - codIRange
binVal = 0
Done

Figure 14 Flowchart of the bypass decoding flow [1]

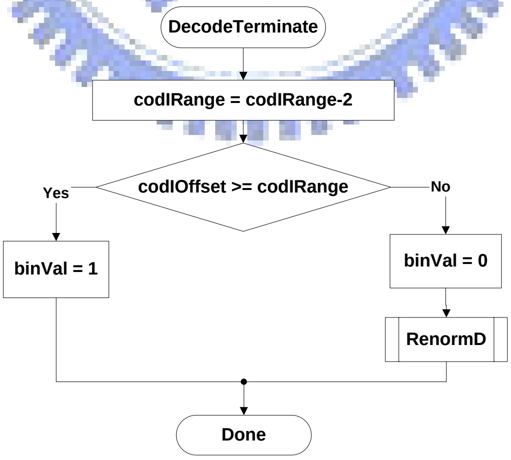


Figure 15 Flowchart of the terminal decoding flow [1]

The third algorithm is the termination decoding process. Figure 15 show the flowchart of the terminal decoding flow. The terminal decoding process is very simple as well, but it has the more decoding procedure compared to the bypass decoding process. It doesn't need the context model to refer to the probability. The value of the next *codlRange* is always to subtract two from the current *codlRange* depending on whether the subdivision condition belongs to MPS or not. The final values of *codlRange* and *codlOffset* are required to renormalize through the *RenormD* in this figure when it branches to the situation that *codlOffset is* smaller than *codlRange* (MPS condition). The architecture of this flowchart is composed of one constant subtraction, one comparator, and one renormalization. The termination decoding process is used to trace if the current slice is ended. It occurs one time per macroblock process which is seldom used during all decoding processes.

2.5 Context model organization

The values of the context model offer the probability value of MPS (*pStateIdx*) and the historical value of *bin* (*MPS*) in order to achieve the adaptive performance. The number of the context model is 399 for baseline profile and 701 for main profile. Here we take the baseline profile context model to introduce the operation of the context model briefly. In the normal encoding/decoding process of the arithmetic encoder/decoder, we have to prepare the 399 locations of the context model to record all encoding/decoding results.

context model index = $ctxIdxOffset+ctxIdxInc$ (Eq. 11)

context model index = $ctxIdxOffset+ ctxIdxBlockCatOffset+ ctxIdxInc$ (Eq. 12)

We divide into two kinds of the context model index methods to allocate the context model. Eq. 11 is one of the index methods. Except residual data encoding/decoding, the context model index is equal to the sum of *ctxIdxOffset* and *ctxIdxInc*. Depending on the syntax element and the slice type, we can find the value of *ctxIdxOffset* in Table 6. The value of *ctxIdxInc* is looked up in Table 9 by referring to the syntax element and *binIdx*. In table 9, the alphabet of "na" denotes the never happened issue and the word of "Terminate" means that the encoding/decoding flow enters the terminal encoding/decoding process. If the generated *bin* is equal to "1", the slice has to be stopped and encodes/decodes the next slice. Table 10 shows the value of *ctxIdxInc* referring to the required neighbor syntax elements of top and left blocks which will be explained in Section 2.6. Table 11 shows the value of *ctxIdxInc* in special *binIdx* when encoding/decoding *mb_type* in Table 9.

Eq. 12 is the index method for the residual data encoding/decoding which focuses on the syntax element of the residual data encoding/decoding flow such as *coded_block_flag*, *significant_coeff_flag*, *last_significant_coeff_flag*, and *coeff_abs_level_minus1*. The value of the context model index is the sum of *ctxIdxOffset*, *ctxIdxBlockCatOffset*, and *ctxIdxInc*. The assignment of *ctxIdxOffset* is also shown in Table 6. The value of *ctxIdxBlockCatOffset* is defined as Table 8 which is dominated by the parameters of syntax elements and *ctxBlockCat*. The value of *ctxBlockCat* is the block categories for the different coefficient presentations. *maxNumCoeff* means the required coefficient number of the current *ctxBlockCat*. *ctxBlockCat* sorts five block categories which are luma_DC for 4x4 blocks, luma_AC for 4x4 blocks, luma_4x4, chroma_DC, and chroma_AC in Table 7. The value of *ctxIdxInc* in residual data is defined as the scanning position that ranges from 0 to "maxNumCoeff – 2" in Table 7. The scanning position of the residual data process has two scanning orders. One is scanned for frame coded blocks with zig-zag scan and the other is scanned for field coded blocks with field scan.

Table 6 Value of *ctxIdxOffset* definition [1]

image layer	syntax element	slice type			
		SI	I	P,SP	B
slice data	*mb_skip_flag*	—	—	11	24
	mb_field_decoding_flag	70	70	70	70
macroblock layer	*mb_type*	3	—	—	—
	mb_type(prefix)	0	—	14	27
	mb_type(suffix)	3	—	17	32
	coded_block_pattern(prefix)	73	73	73	73
	coded_block_pattern(suffix)	77	77	77	77
	mb_qp_delta	60	60	60	60
MB prediction (intra)	*Prev_intra4x4_pre_mode_flag*	68	68	68	68
	rem_intra4x4_pred_mode	69	69	69	69
	Intra_chroma_pred_mode	64	64	64	64
MB prediction and sub-MB prediction (inter)	*ref_idx_l0*	—	—	54	54
	ref_idx_l1	—	—	—	54
	Mvd_l0_x	—	—	40	40
	Mvd_l1_x	—	—	—	40
	Mvd_l0_y	—	—	47	47
	Mvd_l1_y	—	—	—	47
sub-MB prediction	*sub_mb_type*	—	—	21	36
residual data	*coded_block_flag*	85	85	85	85
	Significant_coeff_flag(field)	105	105	105	105
	Significant_coeff_flag(frame)	277	277	277	277
	last_significant_coeff_flag(field)	166	166	166	166
	last_significant_coeff_flag(frame)	338	338	338	338
	Coeff_abs_level_minus1	227	227	227	227

Table 7 Assignment of *ctxBlockCat* due to coefficient type [1]

coefficient type	maxNumCoeff	ctxBlockCat
luma DC	16	0
luma AC	15	1
Luma coefficient	16	2
chroma DC	4	3
chroma AC	15	4

Table 8 Assignment of ctxIdxBlockCatOffset due to ctxBlockCat and syntax elements of the residual data [1]

Syntax element of the residualdata	*ctxBlockCat*				
	0	1	2	3	4
coded_block_flag	0	4	8	12	16
Significant_coeff_flag	0	15	29	44	47
last_significant_coeff_flag	0	15	29	44	47
coeff_abs_level_minus1	0	10	20	30	39

Table 9 Definition of the *ctxIdxInc* value for context model index [1]

Syntax element	binIdx						
	0	1	2	3	4	5	>=6
mbType_SI(prefix)	0,1,2	na	na	na	na	na	na
mbType_SI(suffix) *mbType_I*		Terminate	3	4	5,6	6,7	7
Mb_skip_flag_P		na	na	na	na	na	na
mbType_P(prefix)	0	1	2,3	na	na	na	na
mbType_P(suffix)	0	Terminate	1	2	2,3	3	3
sub_mb_type_P	0	1	2	na	na	na	na
Mb_skip_flag_B	0,1,2	na	na	na	na	na	na
mbType_B(prefix)		3	4,5	5	5	5	5
mbType_B(suffix)	0	Terminate	1	2	2,3	3	3
sub_mbType_B	0	1	2,3	3	3	3	na
mvdl0_x, mvdl1_x	0,1,2	3	4	5	6	6	6
mvdl0_y, mvdl1_y		3	4	5	6	6	6
Ref_idx_l0 , ref_idx_l1	0,1,2,3	4	5	5	5	5	5
Mb_qp_delta	0,1	2	3	3	3	3	3
intra_chroma_pred_mode	0,1,2	3	3	na	na	na	na
prev_intra4x4_pre_mode_fl	0	na	na	na	na	na	na
rem_intra4x4_pred_mode	0	0	0	na	na	na	na
Mb_field_decoding_flag	0,1,2	na	na	na	na	na	na
coded_block_pattern(prefix	0,1,2,3	0,1,2,3	0,1,2,3	0,1,2,3	na	na	na
coded_block_pattern(suffix)		4,5,6,7	na	na	na	na	na
end_of_slice	0	na	na	na	na	na	na

Table 10 Required syntax elements of the left and top neighbor blocks and the computation for *ctxIdxInc*

<table>
<tr><th>Syntax element</th><th>A (left block)</th><th>B (top block)</th><th colspan="2">ctxIdxInc</th></tr>
<tr><td>mbType</td><td>mbType
mb_skip_flag</td><td>mbType
mb_skip_flag</td><td colspan="2">A + B</td></tr>
<tr><td>mb_skip_flag</td><td>Mb_skip_flag</td><td>mb_skip_flag</td><td colspan="2">A + B</td></tr>
<tr><td>Mvdl0_x, mvdl0_y</td><td>Sub_mb_type
mb_skip_flag
ref_idx_l0</td><td>sub_mb_type
mb_skip_flag
ref_idx_l0</td><td colspan="2" rowspan="2">(Eq. 11)</td></tr>
<tr><td>Mvdl1_x, mvdl1_y</td><td>Sub_mb_type
mb_skip_flag
ref_idx_l1</td><td>sub_mb_type
mb_skip_flag
ref_idx_l1</td></tr>
<tr><td>ref_idx_l0 , ref_idx_l1</td><td>Sub_mb_type
mb_skip_flag
ref_idx_l0 ref_idx_l1</td><td>sub_mb_type
mb_skip_flag
ref_idx_l0 ref_idx_l1</td><td colspan="2">{B , A}</td></tr>
<tr><td>mb_qp_delta</td><td>Na</td><td>Na</td><td colspan="2">(mb_qp_delta != 0 && coded_block_pattern != 0)</td></tr>
<tr><td>Intra_chroma_pred_mode</td><td>intra_chroma_pred_mode
mbType</td><td>intra_chroma_pred_mode
mbType</td><td colspan="2">A + B</td></tr>
<tr><td>mb_field_decoding_flag</td><td>Mb_field_decoding_flag</td><td>mb_field_decoding_flag</td><td colspan="2">A + B</td></tr>
<tr><td>coded_block_pattern(prefix)</td><td rowspan="3">coded_block_pattern
mbType</td><td rowspan="3">Coded_block_pattern
mbType</td><td colspan="2">{B , A}</td></tr>
<tr><td rowspan="2">coded_block_pattern(suffix)</td><td>binIdx == 0</td><td>{B , A}</td></tr>
<tr><td>binIdx == 1</td><td>{B , A}+4</td></tr>
<tr><td>coded_block_flag</td><td>coded_block_flag
coded_block_pattern
mbType</td><td>coded_block_flag
coded_block_pattern
mbType</td><td colspan="2">{B , A}</td></tr>
</table>

Table 11 Assignment of *ctxIdx* for syntax element *mbType*

Syntax element	current *binIdx*	index of the read *bin*	value of the read bin	*ctxIdxInc*
mbType_SI(suffix)	4	3	0	6
			1	5
	5	3	0	7
			1	6
mbType__P(prefix)	2	1	0	3
			1	2
mbType__P(suffix)	4	3	0	3
			1	2
mbType__B(prefix)	2	1	0	5
			1	4
mbType__B(suffix)	4	3	0	3
			1	2
sub_mbtype_B	2	1	0	3
			1	2

In addition, *ctxIdxInc* related to the syntax element of *mvd* has the special definition. Eq. 13 shows the *ctxIdxInc* definition of *mvd*. It checks the sum of the absolute *mvd* values in left and top sub-macroblocks. If the summation is less than “3”, the value *ctxIdxInc* is defined as “0”. If the summation is greater than “32”, the value *ctxIdxInc* is defined as “2”. Otherwise, the value *ctxIdxInc* is defined as “1”.

```
sum_A_B = abs(mvd[A])+ abs(mvd[B])
If ( sum_A_B < 3)
    ctxIdxInc = 0 ;
else if( sum_A_B > 32)
    ctxIdxInc = 2 ;
else
    ctxIdxInc = 1 ;                                        (Eq. 13)
```

2.6 Syntax elements for the neighbor blocks

In the previous section, we have explained the methods to compute the context model index to offer the arithmetic decoder to produce the bin value. In both the residual data and the general encoding/decoding, the context model index is dominated by two factors such as ctxIdxOffset and ctxIdxInc. ctxIdxInc is the only one factor related with the syntax elements of the neighbor blocks. In Table 10, we observe the variable syntax elements referring to the left and top blocks to define the ctxIdxInc of the first binIdx such as mb_Type, mb_skip_flag, ref_idx, mb_qp_delta, intra_chroma_pred_mode, mb_field_decoding_flag, and coded_block_pattern. In this section, we introduce how to refer to syntax elements of the left and top neighbor blocks.

In CABAC system, it has two side syntax elements to be required such as the left and top ones. The referred position is based on the current block which can treat as not only the macroblock but also the sub-macroblock. So we have two methods to allocate the required blocks in two levels as follows.

The first method is to get neighbor in macroblock level. Figure 16 illustrates the left and top macroblocks of the current one. “N” denotes the number of the macroblock in the current slice. The black block is the current encoding/decoding macroblock which is the N-th encoded/decoded macroblock coordinated as (MB_x , MB_y) in this slice. The macroblock “N-1” is the left macroblock and “N-w” is the top block where “w” is the width of this frame and means that the frame has “w” macroblocks in every row. In the method of Figure 16, the syntax elements are described for one parameter each macroblock except mvd, ref_idx, and residual data.

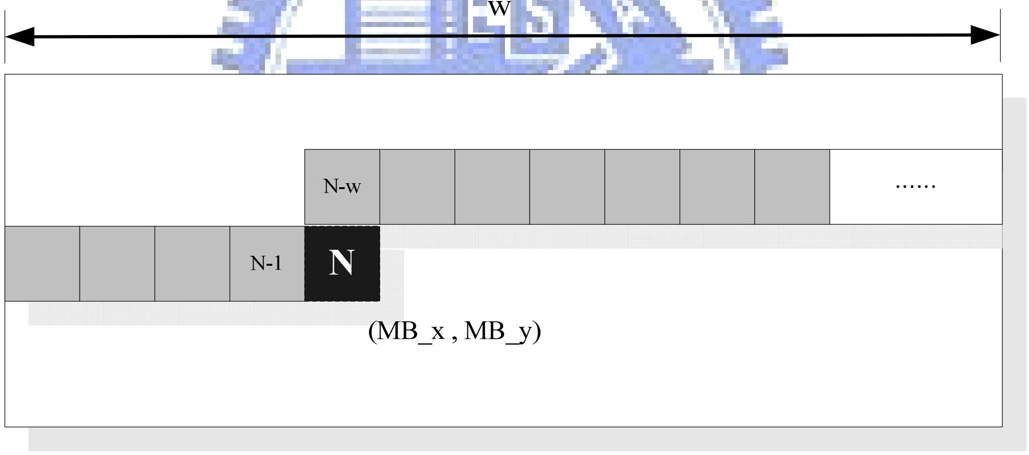


Figure 16 Illustration of the neighbor location in macroblock level

The second method is to get neighbor in sub-macroblock level. Figure 17 illustrates the sub-macroblocks in the current, left and top side macroblocks. The coordinate of the current sub-macroblock is defined as (sub_MB_x , sub_MB_y). The neighbor location is like the allocation in macroblock level. If sub_MB_x is not equal to “0”, the left sub_macroblock is in the left side of the current macroblock. If sub_MB_x is equal to “0”, the left sub_macroblock can’t be found in the current macroblock and has to refer to the left side of the macroblock A. The gray circles in the macroblock A are the required sub-macroblocks which mean the syntax elements of the sub-macroblock 3, 7, 11, 15 have to be stored in order to record the left sub-macroblock. If sub_MB_y is not equal to “0”, the top sub_macroblock is in the upper side of the current macroblock. If sub_MB_y is equal to “0”, the top sub_macroblock can’t be found in the current macroblock and has to refer to the upper side of the macroblock B. The gray circles in the macroblock B are the required sub-macroblocks which mean the syntax elements of the sub-macroblock 12, 13, 14, 15 have to be stored in order to record the top sub-macroblock.

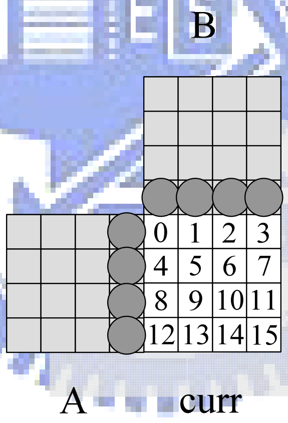


Figure 17 **Illustration of the neighbor location for sub-macroblock level**

2.7 *Paper survey for state-of-the-art CABAC designs*

In this section, we will introduce some of the state-of-the-art CABAC encoding designs which have been published recently (2004~ 2006). Instead of introducing the CABAC encoding and decoding designs, we only introduce the state-of-the-art CABAC encoding designs due to that the proposed three high throughput methods are all for encoder.

The main differences of all of these CABAC encoding are almost in arithmetic encoder due to that the arithmetic encoder is the main dominator of throughput for the whole CABAC system, so we mainly introduce the arithmetic encoders of each state-of the-art designs here. The state-of-the-art CABAC encoder designs are introduced as follows.

1. For the CABAC design of [7] proposed by V. H. Ha, W.–S. Shim, and J.–W. Kim, the initial design without optimization takes 14 clock cycles per bin. The optimization strategies are shown as follows.

(1) Algorithm: optimize computations in each step

(2) Merging: merge multiple steps into one

(3) Pre-fetching: pre-fetch and pre-process data as early as possible

(4) Parallelism: perform independent steps in parallel.

After adopting the four strategies, the processing time is reduced to 5 clock cycles per bin.

2. Figure 18 shows the arithmetic encoder architecture proposed by Roberto R. Osorio [11]. For normal encoding mode of arithmetic coding, it combines two sets of range updating modules and two sets of pre-computing modules for low updating to achieve the purpose of dual-symbol encoding.

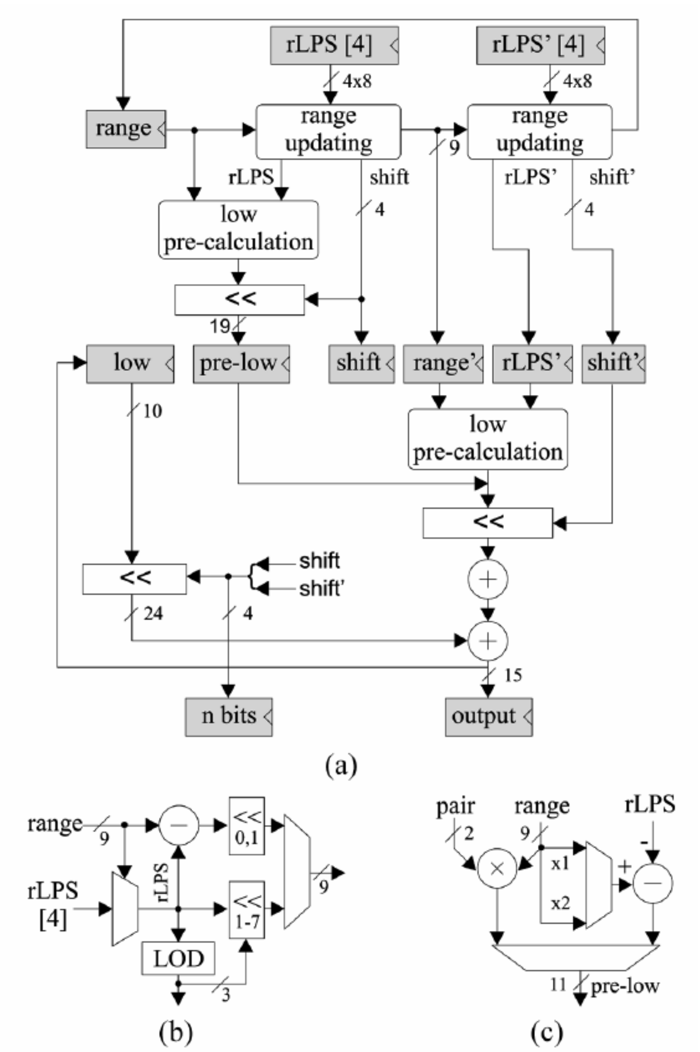
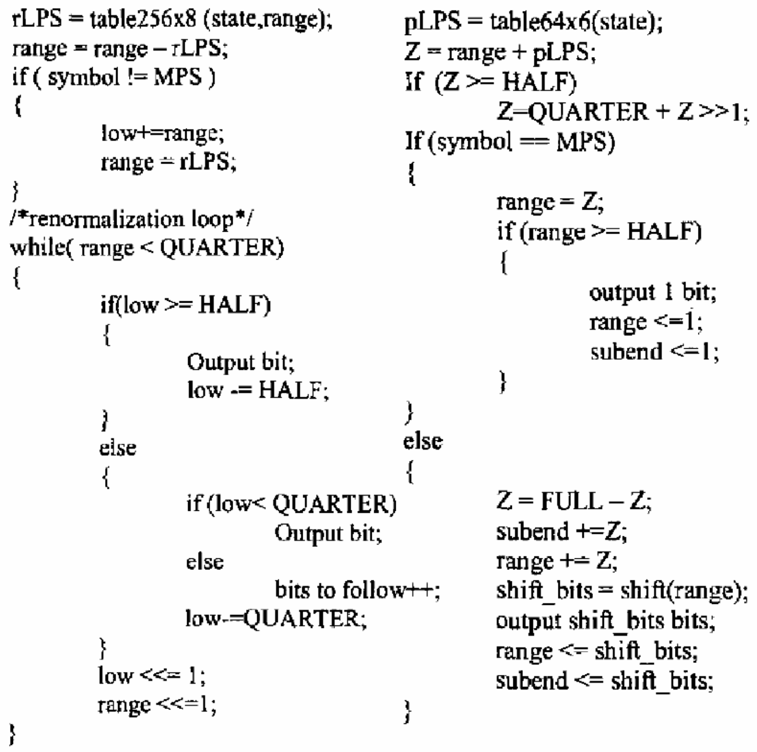


Figure 18 Arithmetic coding interval update. (a) Complete iteration. (b) range updating. (c) Pre-computations for low updating. [11]

3. Figure 19 shows the simplified (pseudo-code) description of the MZ-coder algorithm (right) proposed by J. Núňez and V. Chouliaras [9] and the original CABAC algorithm (left). The MZ-coder evolves from the Z-coder software algorithm presented in [16] as a generalization of the well known GolombiRice coder for lossless coding of bi-level images. The variables of coding state for the CABAC algorithm are range and low, and for the MZ-coder are range and subend. The renormalization process in the MZ-coder does not include internal dependencies. As a result it can be readily accomplished with a single shift left operation. On the other hand the pseudo-code for

the CABAC algorithm shows the internal dependencies of low inside the while loop. This dependency means that a variable number of cycles (from 0 up to a maximum of 6) are required to maintain the state variables in the required range.



```
rLPS = table256x8 (state,range);
range = range – rLPS;
if ( symbol != MPS )
{
        low+=range;
        range = rLPS;
}
/*renormalization loop*/
while( range < QUARTER)
{
        if(low >= HALF)
        {
                Output bit;
                low -= HALF;
        }
        else
        {
                if (low< QUARTER)
                        Output bit;
                else
                        bits to follow++;
                low-=QUARTER;
        }
        low <<= 1;
        range <<=1;
}
```

```
pLPS = table64x6(state);
Z = range + pLPS;
If (Z >= HALF)
        Z=QUARTER + Z >>1;
If (symbol == MPS)
{
        range = Z;
        if (range >= HALF)
        {
                output 1 bit;
                range <=1;
                subend <=1;
        }
}
else
{
        Z = FULL – Z;
        subend +=Z;
        range += Z;
        shift_bits = shift(range);
        output shift_bits bits;
        range <= shift_bits;
        subend <= shift_bits;
}
```

1896

Figure 19 CABAC & MZ pseudo-code description [9]

4. The CABAC encoder design of [12] is proposed by Hassan Shojania and Subramania Sudharsanan. Due to the subtraction operations on codILow in 1 and 1+ branches of renormalization, a simple barrel shifter can not be used directly to mimic the cumulative effect of the iterations for codILow update and output bits generation. However, since each subtraction only affects a single bit of value at positions 8 or 9, an iter-size left shift of codILow still preserves all the necessary information to retrieve the updated codILow value. A few special rules are required to derive the updated value:

(1) The shifted-out bits and the top bit of codILow (bit 9) form an iter+1-bit

parsing area to be interpreted from left to right. Figure 20 shows an example with iter = 5.

(2) Only the leading 1's in the parsing area are proper output bits which won't need further processing. The other 1's are outstanding bits which need to be resolved either by a following zero in the current parsing area or other deterministic 1 or 0 of the subsequent parsing areas resulting from encoding of next symbols.

(3) The first encountered zero bit is to be always ignored.

(4) The updated codILow receives bits 0 to 8 of the shifted codILow. If the shifted-out bits (top iter bits) are all 1, bit 9 is copied over too; otherwise bit 9 must be set to 0.

These rules help constructing a parser using combinational logic to obtain updated codILow and a bit-string of length iter. This combinational logic effectively replaces the 4K entry ROM discussed earlier. If no outstanding bits are present, the string will correspond to output bits. To speed up update of codILow, the combinational circuit is split into two stages corresponding to update of codILow (Low Renormalizer) and generation of the output bits (Parser). Since codILow is updated at the end of the first stage, encoding next symbol can start right after this update.

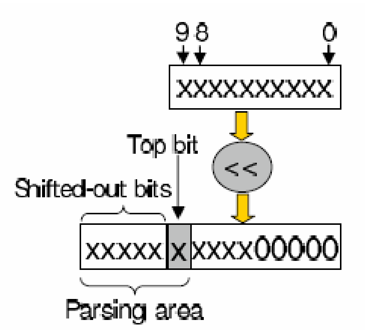


Figure 20 Sample update of codlLow [12]

Chapter 3
Binary Arithmetic Encoding Engine

In this chapter, we propose three methods to promote the throughput of CABAC encoder, and the corresponding architectures of these methods will be also introduced. About the high throughput CABAC decoder, we continue using [2] which we presented in August 2006. So we don't mention it any more here.

The motivation of improving the throughput of binary arithmetic encoder engine is due to the following reasons. The arithmetic encoding engine is the operating center, it dominates the throughput of the whole CABAC encoding system. So the throughput improving methods which focus on it will be more efficient.

There are two main bottlenecks in arithmetic encoding engine, they lead to serious restriction in throughput aspect. One is that the sub-interval algorithm causes the highly data dependence between the current interval and the next ones due to its characteristic of high compression gain. Both the arithmetic encoder and the arithmetic decoder have such a drawback caused by their algorithm intrinsic. The other one is caused by bitsOutstanding accumulation, it appears only in the arithmetic encoder. The detail of bitsOutstanding accumulation will be introduced in Section 3.2.1.

In H.264/AVC system, the entropy coding includes the variable length codes (UVLC and CAVLC) and CABAC. In baseline profile, UVLC and CAVLC are the main coders to compress the macroblock information related to the parameter and the pixel coefficients. In main profile, CABAC substitutes for UVLC and CAVLC to process the video data. CABAC applies two levels hierarchical coding flow. One is the binarization coding flow, it is similar to the process of the variable length coders such as UVLC and CAVLC. Except searching for the context model index, the algorithm of the binarization is easy to realize. The other one is the arithmetic coding flow, it is also the focus of our research.

This chapter is organized as follows. In section 3.1, we present the overview of CABAC for H.264/AVC. In Section 3.2, the proposed three throughput promoting methods will be introduced. We show these three methods in the three sub-sections such as Section 3.2.1, Section 3.2.2, and Section 3.2.3 for more detailed description.

3.1 Overview of CABAC

Figure 21 is the proposed system architecture of CABAC encoder, it consists of three main modules namely the binarization engine, the arithmetic encoder (AE), and the SRAM module. This architecture is based on the design [2] which we proposed in August 2006. The arithmetic encoder is the operating center of the whole CABAC encoding system, so we focus on promoting its efficiency.

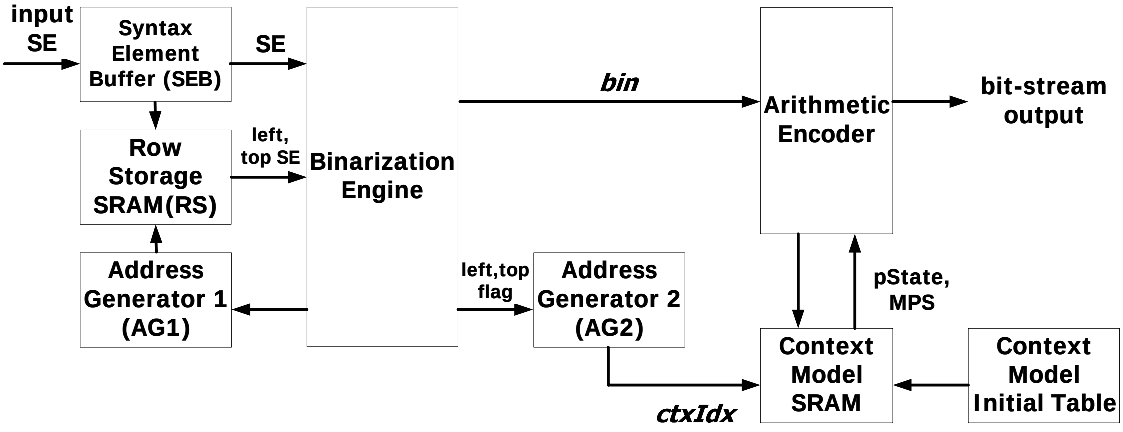


Figure 21 System architecture of CABAC encoder

The entire CABAC encoding procedure is described as follows. When starting to encode, it has to initialize the context model SRAM by looking up the initial table which we implement only by means of the combinational circuit. Besides, the context model also has to be re-initialized when every new slice starts. The adopted context model SRAM is (701x7 bits) dual port SRAM. After the initializing step, the binarization engine has to read the syntax element by syntax element buffer at first. Meanwhile, the binarization engine also controls the row-storage SRAM to backup the essential syntax elements from the syntax element buffer. The adopted row-storage SRAM is (120x208 bits) single port SRAM. The chosen size of the row-storage SRAM is in order to support the encoding of 1080HD video sequence (1920x1088). The detail of memory requirement will be shown in Section 4.2. Besides, the essential syntax elements mean the neighbor information which belongs to the neighbor macroblocks of the current macroblock or the next few macroblocks. The neighbor macroblocks mean the top macroblock and the left macroblock of the current macroblock or the top sub-macroblock and the left sub-macroblock of the current sub-macroblock. The detailed definition of neighbor macroblock and neighbor sub-macroblock are illustrated in Section 2.6. After binarization process, it generates the corresponding *bin* string of the current syntax element to provide for arithmetic encoder. When the arithmetic encoder encoding the *bin* string, it refers to the current probability from the context model SRAM to find the sub-range of MPS and LPS, then it updates the probability to the location of the current context model index (*ctxIdx*). After arithmetic encoding process, the corresponding bit-stream will be generated.

AG1 generates the address of the row-storage SRAM. AG2generates the address of the context model SRAM, its organization has been defined in Section 2.5.

Figure 22 is the system architecture of CABAC decoder. As CABAC encoder, it also consists of three main modules namely the arithmetic encoder (AD), the binarization engine, and the SRAM module. This architecture is what we proposed in August 2006 [2]. The arithmetic decoder also dominates the throughput of the whole CABAC decoding system.

The entire decoding procedure is described as follows. When starting to decode, it also has to initialize the context model SRAM as the encoding initial step. Namely, the entire probabilities of the context model SRAM have to be initialized by the context model initial table. After the initializing step, arithmetic decoder read bit-stream to produce the *bin* value depended on the current range (*codlRange*) and the current value (*codlOffset*). Then, the binarization engine read the bin values to judge if the bin string forms the meaningful data. If negative, the binarization engine requests the arithmetic decoder to decode one *bin* again and re-judges the *bin* string until the value of the current syntax element can be identified. Namely, the binarization engine reads *bin* string until matching the *bin* definition of the standard [1] and transfers it into the mapped syntax element such as the macroblock parameter, *mvd*, residual data, and so on. If completing the current slice, *codlRange* is assigned to "512_{10}" and *codlOffset* is refilled in 9-bit bit-stream from the syntax parser.

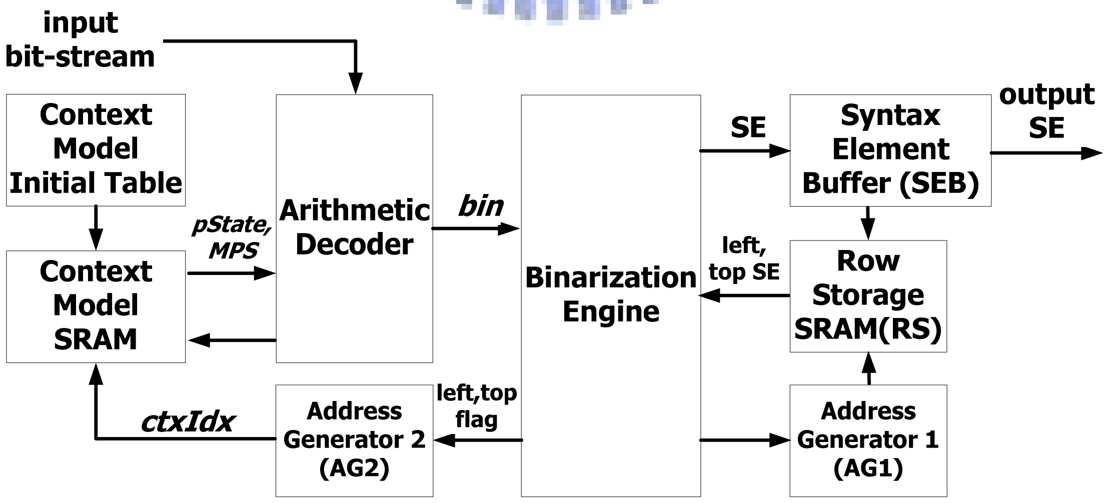


Figure 22 System architecture of CABAC decoder

3.2 *Three throughput promoting methods*

The proposed three throughput promoting methods are all for arithmetic encoder (AE) in CABAC encoding system. This is because the arithmetic encoder is the operating center of the whole CABAC encoding system, and it dominates the throughput of the CABAC encoding system. So the throughput promoting methods focus on the arithmetic encoder will be more efficient.

Besides, about the CABAC decoding system aspect, we continue using the design [2] which we proposed in August 2006, and it has been introduced in detail in [5]. Besides, instead of providing any advanced improving methods for decoder, we only focus on the improving of encoder in the thesis.

The arithmetic encoder is the second level encoding flow. It compresses the *bin* string which is generated by binarization engine (the fist level encoding flow) to be bit-stream. The arithmetic encoder has three kinds of the encoding flows such as normal encoding mode, bypass encoding mode, and terminal encoding mode.

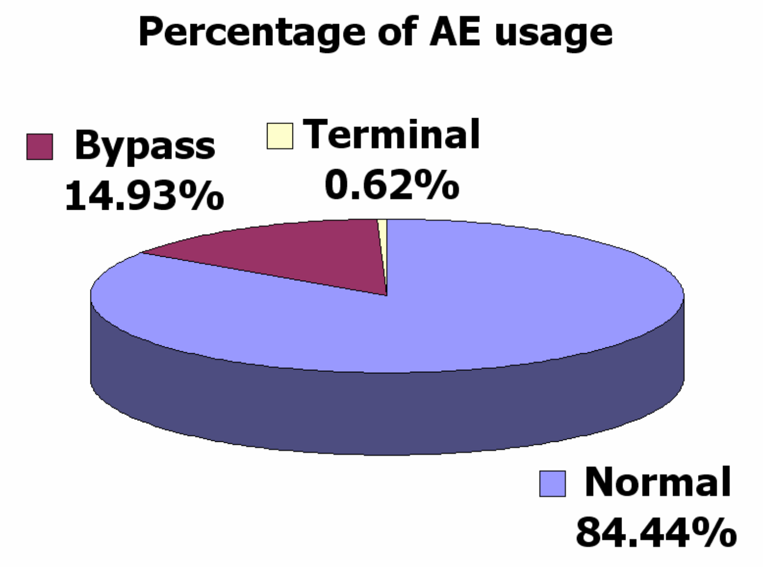


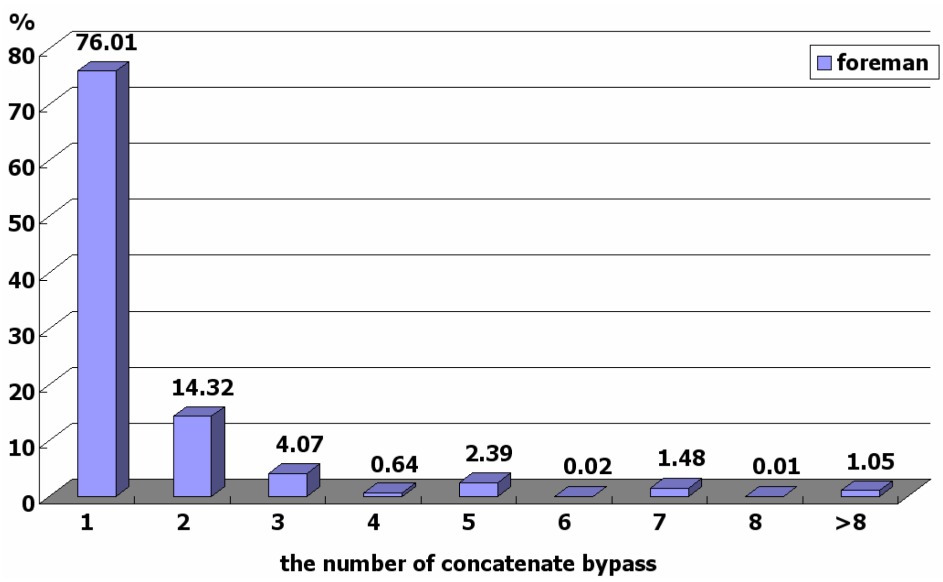
Figure 23 Percentage of the usage of the three encoding modes in AE

The pie chart (Figure 23) shows the usage rate of the three encoding modes in the arithmetic encoder. The normal encoding mode occupies about 84.44% usage rate to the entire times of arithmetic encoding demand. It is the highest usage rate for these three arithmetic encoding modes, so the process efficiency promoting methods which aim at normal encoding mode are more efficient. The methods will be introduced mainly in Section 3.2.3. The bypass encoding mode occupies about 14.93%. We propose multi-symbol architecture which is based on the statistic result of the bypass concatenate distribution to promoting the process efficiency of the bypass encoding mode. The detail of multi-symbol architecture will be introduced in Section 3.2.1. The terminal encoding mode occupies about 0.62%, it is the lowest usage rate for these three arithmetic encoding modes. This is because the terminal encoding mode is only used to judge if the current slice is completion, it works one or two times per macroblock. Thus, the terminal encoding mode is seldom used in CABAC encoding system. So we don't provide any efficiency improving method due to its usage rate occupies approximately 0%.

3.2.1 Multi-symbol architecture

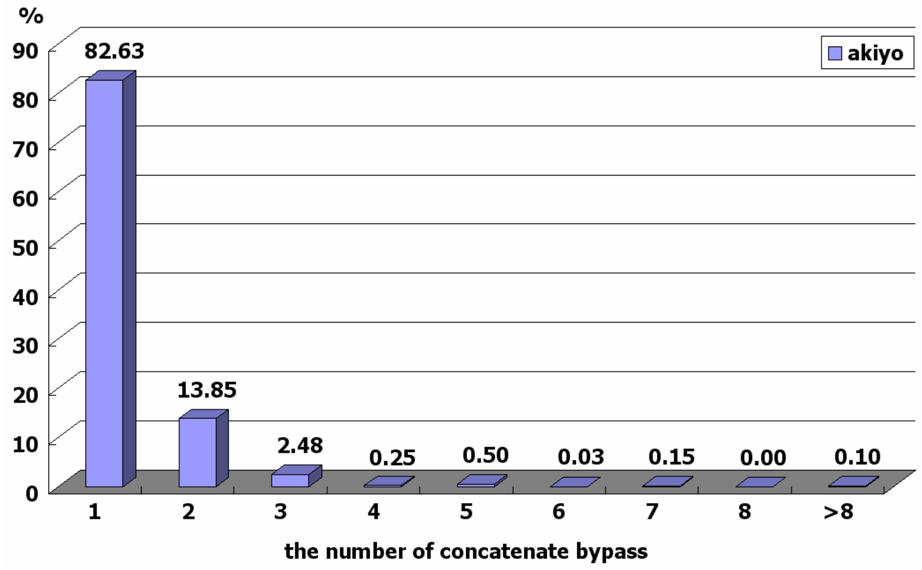
In this section we introduce the proposed first method, it is the multi-symbol architecture. The proposed multi-symbol architecture is only for the bypass encoding mode, it is based on the statistic result of bypass concatenate times per bypass encoding issue. Besides, the multi-symbol architecture is available due to the low complexity of the bypass encoding mode.

Figure 24 shows the statistic of the number and the percentage of the concatenate bypass encoding under executing the six typical video test sequences.



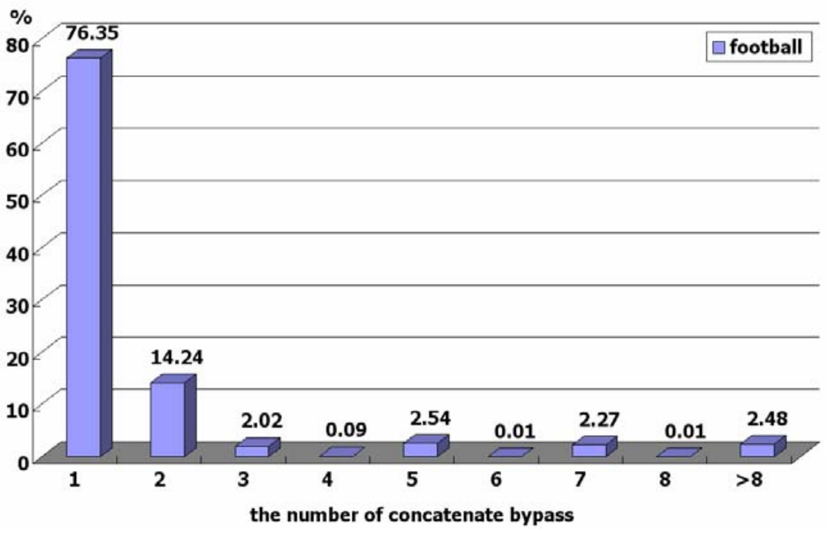
the number of concatenate bypass	1	2	3	4	5	6	7	8	> 8
times	500285	94267	26821	4211	15740	152	9763	56	6927
%	76.01	14.32	4.07	0.64	2.39	0.02	1.48	0.01	1.05

(a) foreman



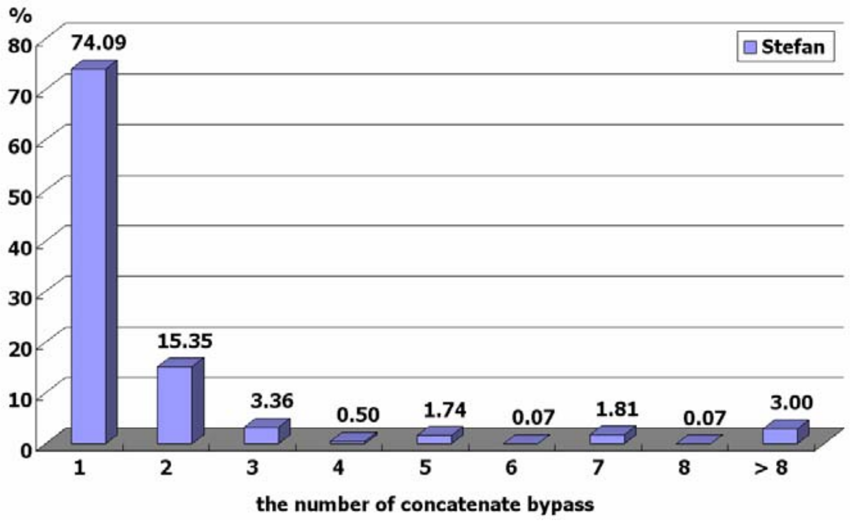
the number of concatenate bypass	1	2	3	4	5	6	7	8	> 8
times	163588	27424	4913	487	990	65	301	0	205
%	82.63	13.85	2.48	0.25	0.50	0.03	0.15	0.00	0.10

(b) akiyo



the number of concatenate bypass	1	2	3	4	5	6	7	8	> 8
times	1283650	239361	33932	1540	42726	144	38174	144	41654
%	76.35	14.24	2.02	0.09	2.54	0.01	2.27	0.01	2.48

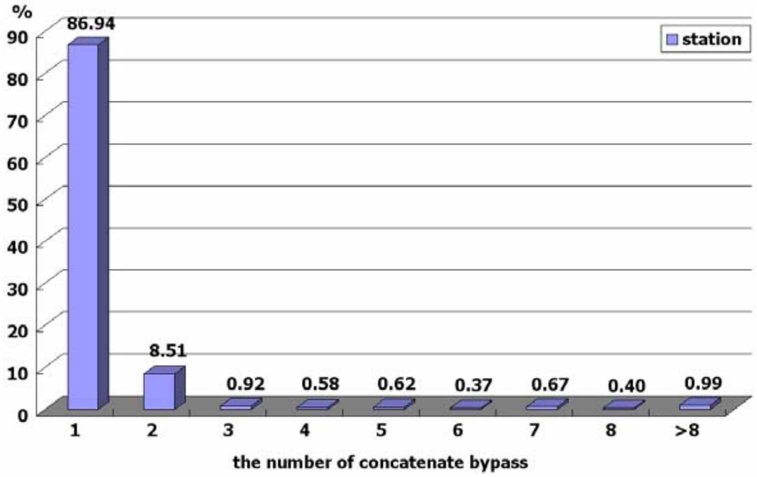
(c) football



the number of concatenate bypass	1	2	3	4	5	6	7	8	> 8
times	1705034	353327	77297	11550	40022	1632	41545	1644	69149
%	74.09	15.35	3.36	0.50	1.74	0.07	1.81	0.07	3.00

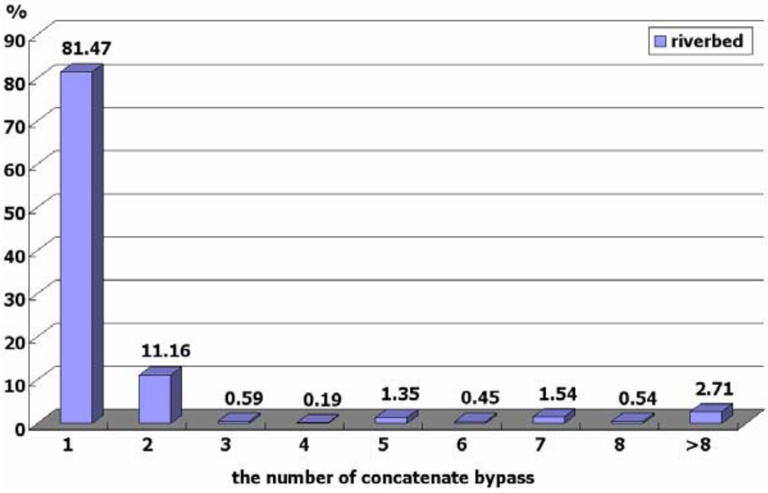
(d) Stefan

Figure 24 Percentage of the number of the concatenate bypass encoding per bypass demand under executing CIF frames for four typical video sequences



the number of concatenate bypass	1	2	3	4	5	6	7	8	> 8
times	503679	49280	5319	3378	3596	2138	3861	2328	5764
%	86.94	8.51	0.92	0.58	0.62	0.37	0.67	0.40	0.99

(a) station



the number of concatenate bypass	1	2	3	4	5	6	7	8	> 8
times	1229837	168489	8971	2859	20346	6761	23183	8206	40857
%	81.47	11.16	0.59	0.19	1.35	0.45	1.54	0.54	2.71

(b) riverbed

Figure 25 Percentage of the number of the concatenate bypass encoding per bypass demand under executing 1080HD frames for two typical video sequences

Figure 24 shows the percentage of the number of the concatenate bypass encoding per bypass demand under executing CIF frames for four typical video sequences. The first two typical video sequences which are the foreman (a) and the akiyo (b) belong to the data characteristic of the low complexity and slow motion. For the foreman sequence, its maximum number of the concatenate bypass is 16, and the average number of the concatenate bypass is 1.517 per bypass demand. For the akiyo sequence, its maximum number of the concatenate bypass is 11, and the average number of the concatenate bypass is 1.235 per bypass demand. The last two typical video sequences which are the football (c) and the Stefan (d) belong to the data characteristic of the high complexity and fast motion. For the football sequence, its maximum number of the concatenate bypass is 22, and the average number of the concatenate bypass is 1.633 per bypass demand. For the Stefan sequence, its maximum number of the concatenate bypass is 31, and the average number of the concatenate bypass is 1.710 per bypass demand.

In Figure 25, it shows the percentage of the number of the concatenate bypass encoding per bypass demand under executing 13 1080HD frames for two typical video sequences. The fist typical video sequence which is the station (a) belongs to the data characteristic of the high complexity but slow motion. The second typical video sequence which is the riverbed (a) belongs to the data characteristic of the low complexity but fast motion. For the station sequence, its maximum number of the concatenate bypass is 20, and the average number of the concatenate bypass is 1.316 per bypass demand. For the riverbed sequence, its maximum number of the concatenate bypass is 24, and the average number of the concatenate bypass is 1.581 per bypass demand.

According to the statistic result of the six typical video sequences shown in Figure 24 and Figure 25, we can find that the percentage of the concatenating two bypass encoding per bypass demand is almost 15% and 10% for CIF video and 1080HD video respectively, and the average concatenate number of bypass encoding which is greater than or equal to two occupies about 20.42% for these six video sequences. Besides, we find the video sequences which are higher complexity or faster motion cause bigger maximum number and bigger average number of the concatenate bypass encoding. This phenomenon is due to higher complexity causing more data in I frame and faster motion causing more data in P and B frame. The average number of concatenate bypass encoding got from the six video sequences is about 1.5, and the number increasing with the accelerating of motion or the increasing of complexity but never being in excess of 2. So we design the multi-symbol architecture consisted of concatenating two bypass encoding units as cascade form. For the video which has higher complexity or faster motion, the performance of our multi-symbol architecture is more efficient.

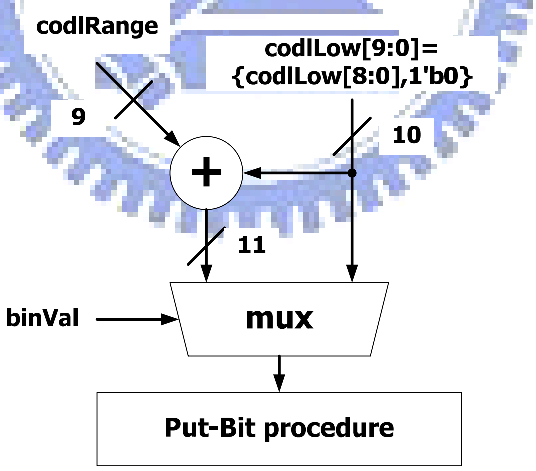


Figure 26 Bypass encoding unit

The multi-symbol architecture is available due to the low complexity of bypass encoding unit. According to the algorithm of bypass encoding introduced in Section 2.4.1, we know that bypass encoding process is quite simple due to it needn't refer context model and no iterant renormalization process. Figure 26 shows the architecture of bypass encoding unit, it mainly consists of one adder, one 2 to 1 multiplexer, and one no iterant renormalization process. The bypass encoding needn't execute renormalization, but the back of its process is similar to renormalization process which is no iterant operation controlled by *codlRange*. The no iterant renormalization process is much simpler than the renormalization process, the detail of this complexity comparison can be found in Section 2.4.1. So we modify the back process of bypass encoding to be hardware sharing with renormalization process, and it is named as one time renormalization namely the no iterant renormalization mentioned above. Besides, the whole bypass encoding unit is implemented only with the combinational circuit.

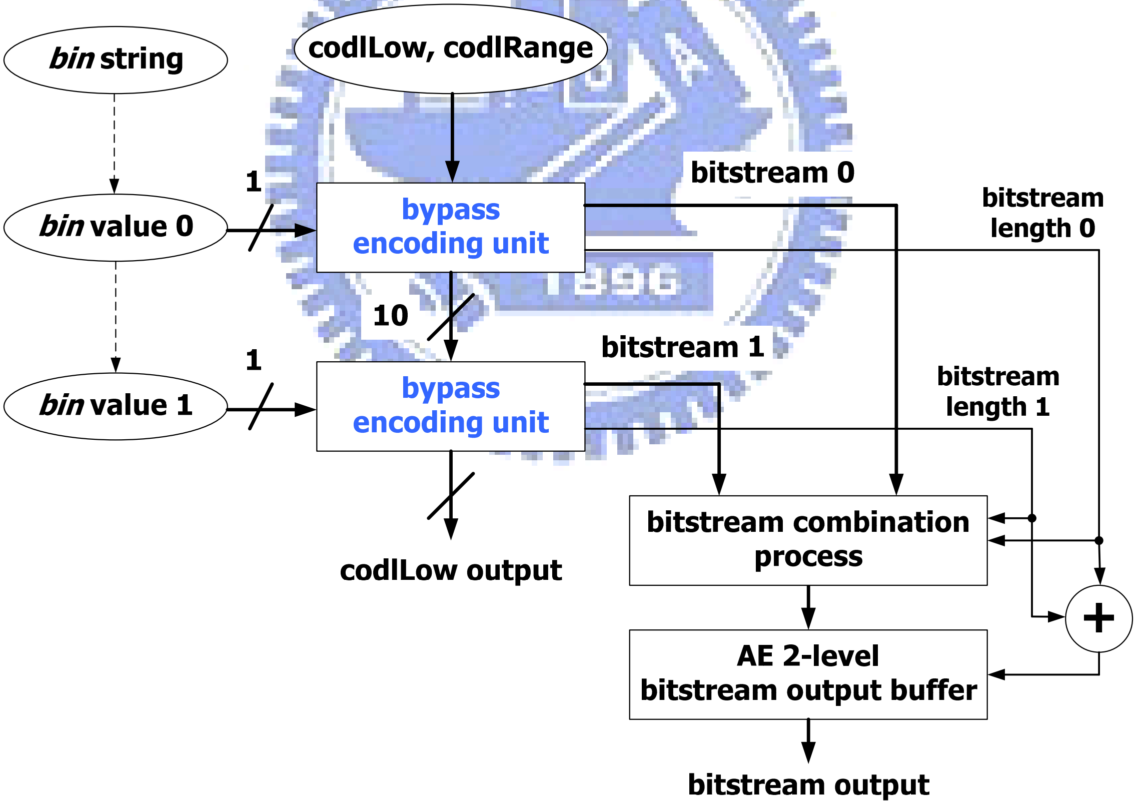


Figure 27 Architecture of the multi-symbol bypass encoding

Figure 27 shows the architecture of multi-symbol bypass encoding. It can improve the concatenate bypass encoding efficiency, and generating the two corresponding bit-streams every one cycle. The whole encoding flow of the multi-symbol architecture is described as follows. If the bypass flag is concatenate distribution namely the concatenate number is great than or equal to two, the multi-symbol architecture reads two *bin* value from the *bin* string every time, but it reads one *bin* value in the last time when its concatenate bypass number is odd. If it reads two *bin* value, it also generate two corresponding bit-stream meanwhile due to the whole bypass encoding unit is implemented only with the combinational circuit. These two generated bit-streams are combined into one bit-stream in the bit-stream combination process by referring the control signals of each bit-stream length. This bit-stream combination process is also implemented only with the combinational circuit. Then, the combined bit-stream is delivered to the arithmetic encoder (AE) output interface namely the AE 2-level bit-stream output buffer. Finally, the 8 bit bit-stream is outputted from the AE output interface. The detail of the AE output interface will be introduced in Section 3.2.3.4.

The multi-symbol architecture almost generates two bypass encoded bit-streams per cycle when the concatenate number of bypass encoding is greater then or equal to two. According to the statistic results of Figure 24 and Figure 25, the concatenate number which is greater than or equal to two occupies about 20.42%. The 20.42% concatenate bypass can be speeded up in encoding by the multi-symbol bypass architecture. The speeding up formula is shown in Eq. 14.

The speeding up formula of multi-symbol architecture:

Multiple of speeding up= {concatenate number/[ceil(concatenate number/2)]} (Eq. 14)

For example, if the concatenate number is 23, it originally needs 23 cycles at least to finish the 23 bypass encodings, but it only needs 12 cycles to encode the 23 bypass after applying the proposed multi-symbol architecture. Therefore, if the concatenate number is two, it only takes one cycle to finish these two bypass encoding. According to the statistic results shown in Figure 24 and Figure 25, they shows that there are about 92.487% concatenate bypass encoding can be finished in one cycle.

3.2.2 Pipeline organization

In this Section we introduce the proposed second method, it is the pipeline organization of arithmetic encoder. The proposed pipeline organization is mainly for normal encoding mode, because it must take extra cycles to refer the context model SRAM. The purpose of the pipeline organization is to make the reading and storing of context model SRAM more efficient.

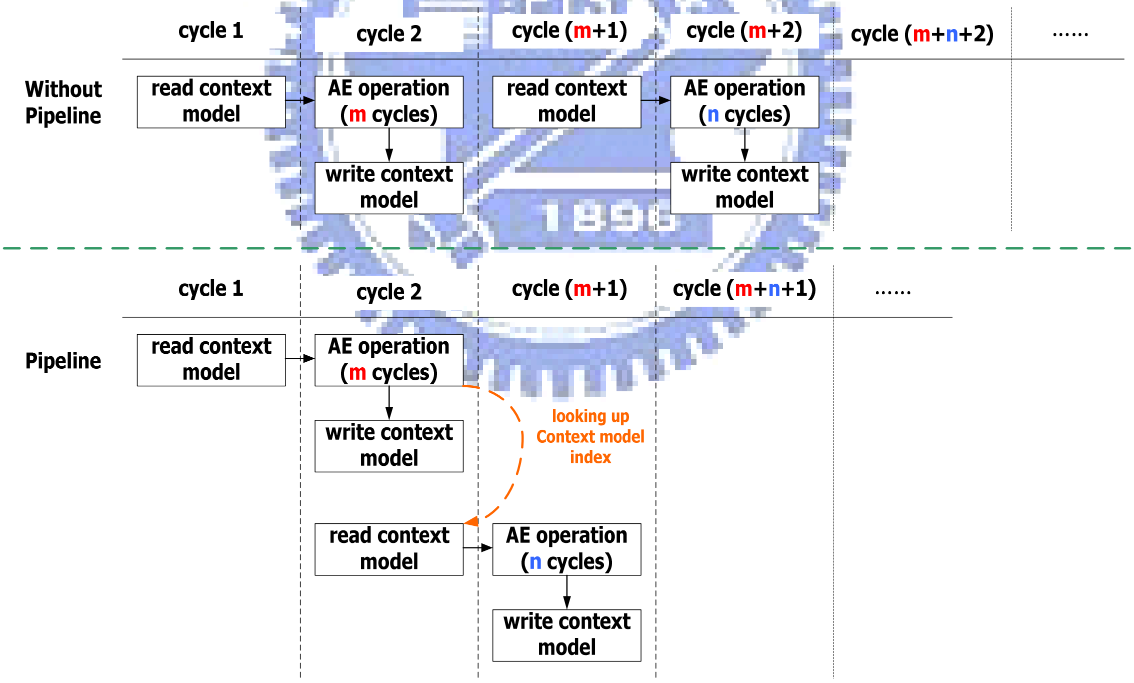


Figure 28 Timing diagram of the pipeline comparison

Figure 28 shows the timing diagram of no pipelining and pipelining for normal encoding in arithmetic encoder. We divide the normal encoding of arithmetic encoder into two stages. The first stage is to read the context model SRAM. The second stage is to encode the *bin* string into bit-stream and to write the probability back to the context model SRAM. We apply these two stages to schedule the pipeline organization.

The bottleneck of the pipeline organization is the variable executing cycle of the normal encoding caused by the bitsOutstanding accumulating in the renormalization process, and its maximum executing cycle is 31. The variable executing cycle leads to the cycle spending of each pipeline stage is not the same, so the pipeline organization is not efficient due to it is not balanced. If the normal encoding takes more than one cycle, the pipeline organization will be inefficient due to it only save one cycle every accessing context model SRAM but the fist access.

We propose a method which will be introduced in Section 3.2.3 to reduce the variable executing cycle to be one cycle in most situations. It not only makes the normal encoding more efficient but also makes the pipeline organization more balanced.

When the normal encoding only takes one cycle, the reading and writing of the context model SRAM will act at the same time. Such a situation leads to the resource conflict of reading and writing in the context model SRAM. Therefore, for the context model SRAM, we adopt dual port SRAM model to implement it. Applying such a dual port SRAM model can make the reading and writing actions of the context model SRAM act in the same cycle.

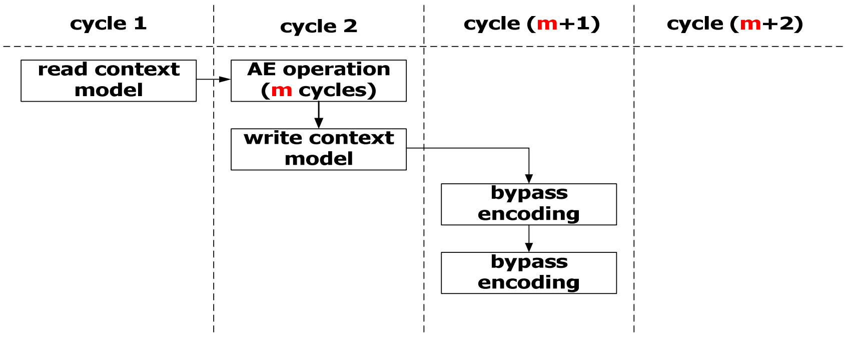
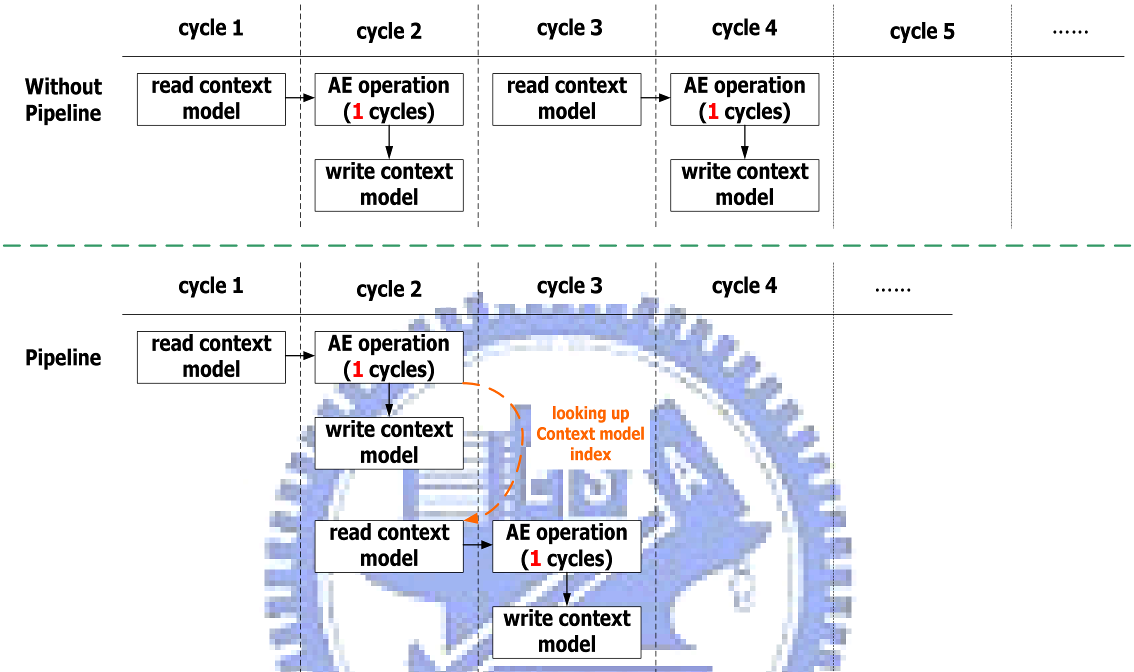
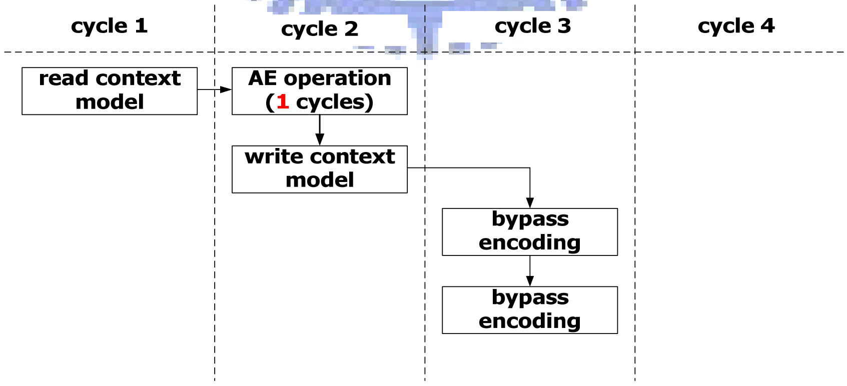


Figure 29 Timing diagram of bypass encoding just after normal encoding

Figure 29 shows the timing diagram of the bypass encoding just after the normal encoding when the concatenate bypass number is bigger than or equal to two. The multi-symbol bypass encoding only take one cycle, so it is efficient in the pipeline organization.



(a) normal encoding



(b) bypass encoding just after normal encoding

Figure 30 Timing diagram after taking the method shown in Section 3.2.3

Figure 30 shows the timing diagram of normal encoding and bypass encoding after adopting the method shown in Section 3.2.3 in most situations. The method makes the normal encoding take only one cycle in most situations, so the pipeline organization shown in Figure 30 is more efficient due to its balance in each stage.

3.2.3 Case Efficiency Architecture

In this section we introduce the proposed third method which is named as Case Efficiency Architecture, it is only for normal encoding and terminal encoding due to they need the renormalization process. The design of Case Efficiency Architecture is based on the statistic result for bitsOutstanding. Namely, according to the probability distribution of the cases of the bitsOutstanding accumulating, it can make the normal encoding more efficient.

Case Efficiency Architecture can improve the efficiency of pipeline organization mentioned in Section 3.2.2 due to it makes the most situations of normal encoding can be finished in only one cycle. So the more balanced pipeline stages make the pipeline organization more efficient after adopting.

This section is divided into four sub-sections. In Section 3.2.3.1, we draw the outline of the bottleneck of the throughput promoting in arithmetic encoding. Section 3.2.3.2 shows the proposed design to improve the throughput bottleneck. In Section 3.2.3.3, the cost issue of interface design risen due to the large variable range of the variable length bit-stream which is caused by the large variable number in bitsOutstanding accumulating (0~31) will be introduced. Section 3.2.3.4 shows the proposed interface design which is based on the statistic result of bitsOutstanding.

3.2.3.1 Throughput bottleneck of arithmetic encoding engine

The arithmetic encoder is the operating center of the CABAC encoder, it dominate the throughput of the whole CABAC encoding process. Besides, The probability of normal encoding is the highest for the three encoding mode of the arithmetic encoding, it is shown in Figure 23 of Section 3.2. The renormalization process takes the most time of normal encoding due to the bitsOutstanding accumulating. So improving the renormalization process is more efficient in throughput promoting of the CABAC encoder.

The bottlenecks of arithmetic encoding engine is the bitsOutstanding accumulating in renormalization process. The bitsOutstanding leads to the following two bottlenecks:

(1) the data dependence between successive symbols.

(2) the variable executing cycles of the renormalization process.

The first throughput bottleneck causes the multi-symbol architecture of normal encoding to be difficult to implement due to its data dependence. It is the intrinsic drawback of the arithmetic encoding algorithm. The second throughput bottleneck is caused by the bitsOutstanding accumulating. The range of the number of bitsOutstanding accumulating is from 0 to 31, it means that it have to take 0~31 cycles to accumulate the bitsOutstanding for one symbol. These two throughput bottlenecks can be illustrated in Figure 7 and Figure 8 of Section 2.4.1. We give an example shown as follows:

If the initial state of the renormalization process is:

The current codlRange is: 0_0000_01xx => it means that the shift = 6

The current codlLow is: 11_0110_1110

The initial bitsOutstanding = 0

The corresponding renormalization process:

(step 0): 11_0110_1110 => PutBit(1) => 1

(step 1): 10_1101_1100 => PutBit(1) => 1

(step 2): 01_1011_1000 => bitsOutstanding = 1

(step 3): 01_0111_0000 => bitsOutstanding = 2

(step 4): 00_1110_0000 => PutBit(0) => 011

(step 5): 01_1100_0000 => bitsOutstanding = 1

=> The bit-stream of current symbol is 1_1011

The remainder bitsOutstanding =1

(PS: The remainder bitsOutstanding carry on being used by the next symbol.)

(Eq. 15)

The remainder bitsOutstanding is the fist throughput bottleneck mentioned above, it leads to the multi-symbol architecture for normal encoding is difficult to be designed. The second throughput bottleneck is the bitsOutstanding accumulating. Namely, the normal encoding may take too many cycles in some situations due to it accumulates the bitsOutstanding cycle by cycle. In next section, we propose the Case Efficiency Architecture to improve the inefficiency caused by the second bottleneck.

3.2.3.2 Throughput efficiency design

The bitsOutstanding accumulating restricts the throughput of the whole arithmetic encoder, so we analyze the data characteristic of bitsOutstanding to look for an improving method. We get the statistic results of the following five pie charts corresponding to five different video test sequences which are the same as those shown in Figure 24 and Figure 25 of Section3.2.1.

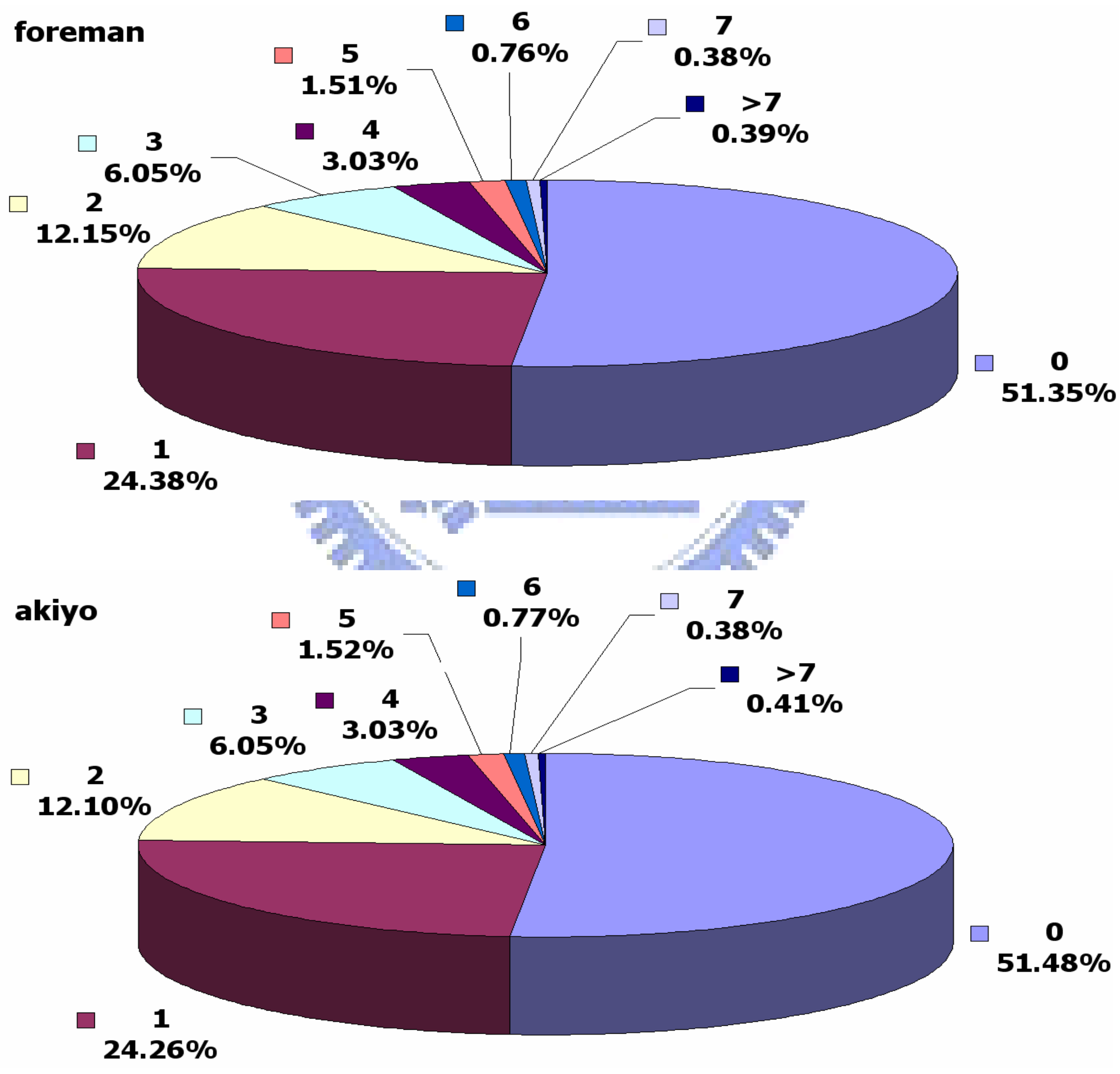


Figure 31 Statistic of bitsOutstanding for the low complexity and slow motion video sequence

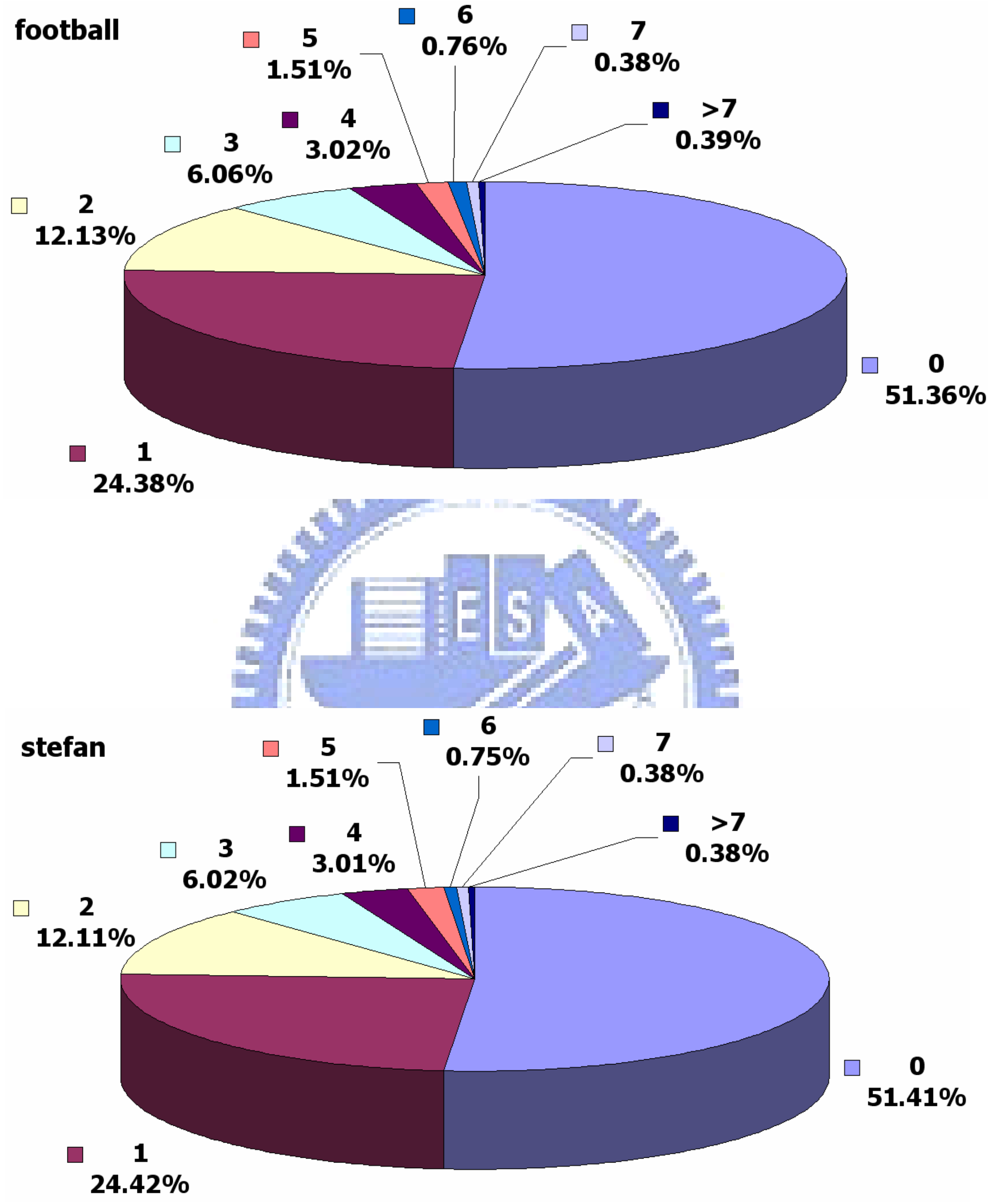


Figure 32 Statistic of bitsOutstanding for the high complexity and fast motion video sequence

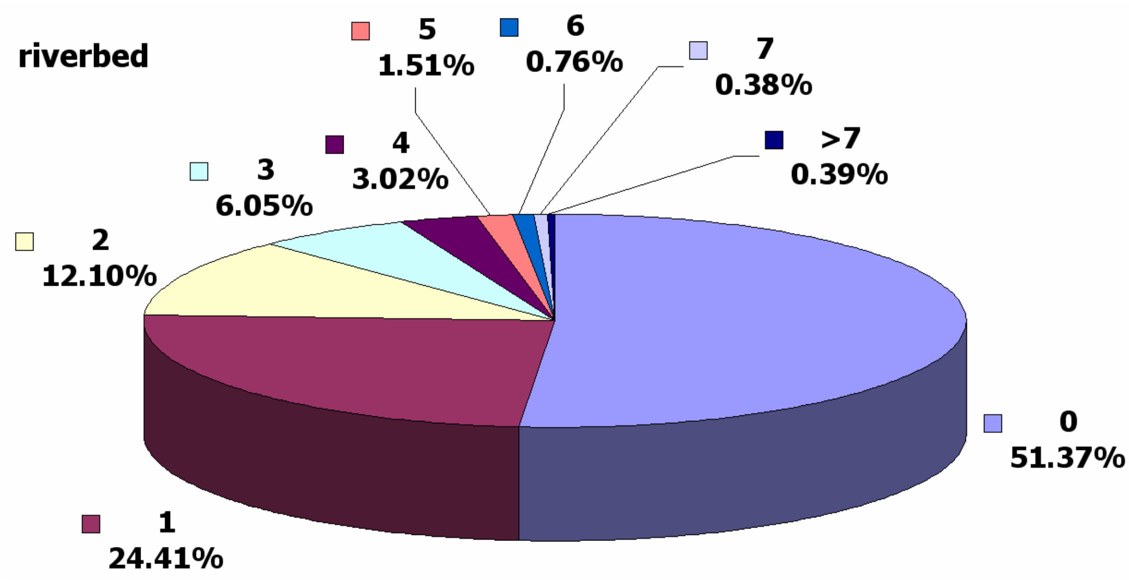


Figure 33 Statistic of bitsOutstanding for 1080HD video sequence

Figure 31, Figure 32, and Figure 33 have similar probability distribution for bitsOutstanding . We can find that the probabilities of bitsOutstanding become great with the getting small of the accumulating number of bitsOutstanding, so we will take the data characteristic of bitsOutstanding to divide the whole renormalization process into cases.

Figure 34 shows the tree map of renormalization process which is corresponding to the flowchart shown in Figure 7 of Section 2.4.1. In Figure 34, the PB(1) and the PB(0) mean the putting bit “1” process and the putting bit “0” process, and the (bs+1) means bitsOutstanding +1. The (9,8) means the bit 9 and the bit 8 of codlLow, and 7, 6, 5, 4…, and 0 means the bit 7, 6, 5, 4, …, and 0 of codlLow.

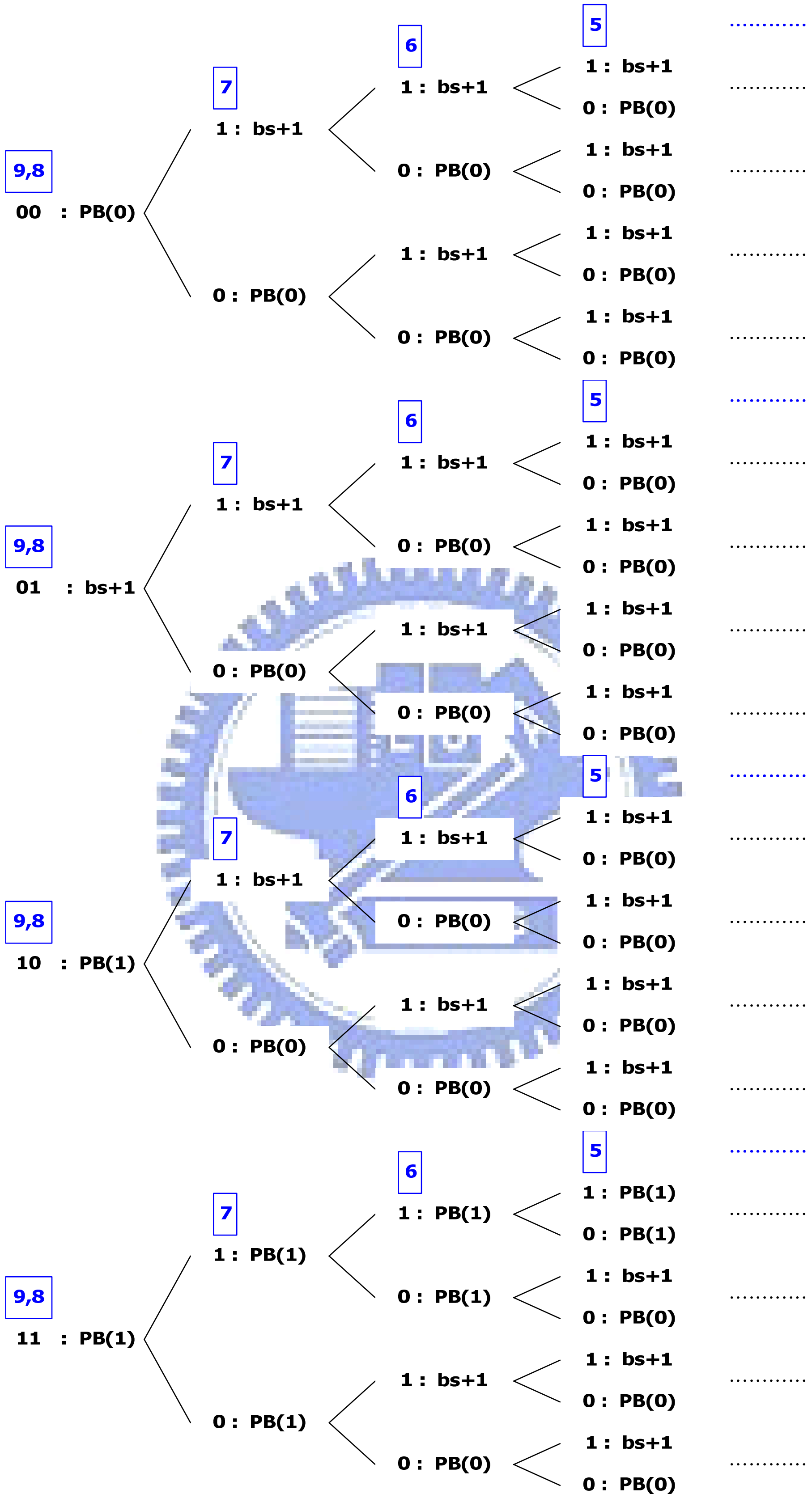


Figure 34 Tree map of the renormalization process

We divide the iterant renormalization process into several cases and estimate the corresponding bit-stream and remainder bitsOutstanding of these cases by the tree maps shown in Figure 34. The example of it is shown in (Eq. 15) of Section 3.2.3.1. The dividing of these cases is based on the codlLow of current symbol and the remainder bitsOutstanding of last symbol. The codlLow is 10 bit, and the maximum shift dominated by codlRange is 7, so the maximum bitsOutstanding number of one symbol is 7. The building of cases which are based on the accumulating number of bitsOutstanding is shown as following tables, and we only show several situation such those shown in Table 12, Table13, and Table14 here due to too many cases in smaller bisOutstanding. The cases shown in Table 12 and Table13 are the cases when their last remainder bitsOutstanding equal to zero, and the cases shown in Table 14 are the cases when their last remainder bitsOutstanding equal to 1.

Table 12 The cases corresponding to its accumulating number of bitsOutstanding=7 (It is under last remainder bitsOutstanding=0) (4 cases)

	The corresponding codlLow	Remainder bitsOutstanding	The generated bit-stream	bit-stream length
shift = 6	01_1111_11xx	7	x	0
shift = 7	00_1111_111x	7	0	1
	01_1111_110x	0	0111_1111	8
	10_1111_111x	7	1	1

Table 13 The cases corresponding to its accumulating number of bitsOutstanding=6 (It is under last remainder bitsOutstanding=0) (12 cases)

	The corresponding codlLow	Remainder bitsOutstanding	The generated bit-stream	bit-stream length
shift = 5	01_1111_1xxx	6	x	0
shift = 6	00_1111_11xx	6	0	1
	01_1111_10xx	0	011_1111	7
	10_1111_11xx	6	1	1
shift = 7	00_0111_111x	6	00	2
	00_1111_110x	0	0011_1111	8
	01_0111_111x	6	01	2
	01_1111_100x	0	0111_1110	8
	01_1111_101x	1	011_1111	7
	10_0111_111x	6	10	2
	10_1111_110x	0	1011_1111	8
	11_0111_111x	6	11	2

Table 14 The cases corresponding to its accumulating number of bitsOutstanding=5 (It is under last remainder bitsOutstanding=1) (32 cases)

	The corresponding codlLow	Remainder bitsOutstanding	The generated bit-stream	bit-stream length
shift = 4	01_1111_xxxx	5	x	0
shift = 5	00_1111_1xxx	5	01	2
	01_1111_0xxx	0	011_1111	7
	10_1111_1xxx	5	10	2

shift = 6	00_0111_11xx	5	010	3
	00_1111_10xx	0	0101_1111	8
	01_0111_11xx	5	011	3
	01_1111_00xx	0	0111_1110	8
	01_1111_01xx	1	011_1111	7
	10_0111_11xx	5	100	3
	10_1111_10xx	0	1001_1111	8
	11_0111_11xx	5	101	3
shift = 7	00_0011_111x	5	0100	4
	00_0111_110x	0	0_1001_1111	9
	00_1011_111x	5	0101	4
	00_1111_100x	5	0_1011_1110	9
	00_1111_101x	1	0101_1111	8
	01_0011_111x	5	0110	4
	01_0111_110x	0	0_1101_1111	9
	01_1011_111x	5	0111	4
	01_1111_000x	0	0_1111_1100	9
	01_1111_001x	1	0111_1110	8
	01_1111_010x	0	0_1111_1101	9
	01_1111_011x	2	011_1111	7
	10_0011_111x	5	1000	4
	10_0111_110x	0	1_0001_1111	9
	10_1011_111x	5	1001	4
	10_1111_100x	0	1_0011_1110	9
	10_1111_101x	1	1001_1111	8

	11_0011_111x	5	1010	4
	11_0111_110x	0	1_0101_1111	9
	11_1011_111x	5	1011	4

Table 15 The number of the corresponding cases for the accumulating number of bitsOutstanding of one symbol

(It is suitable for all kinds of last remainder bitsOutstanding)

The accumulating number of bitsOutstanding	The number of the corresponding cases
7	4
6	12
5	32
4	80
3	187
2	298
1	355
0	52

Table 15 shows the number of the corresponding cases for the accumulating number of bitsOutstanding when encoding one symbol, and these cases are 1020.

Verify the number of cases is 1020:

The range of shift which is decided by codlRange is: 0~7

(The detail of the relationship is shown in Table 17)

The codlLow is 10 bit, but the range of shift is 0~7, so we only consider the 9 bit codlLow[9:1]. It is illustrated in Figure 35.

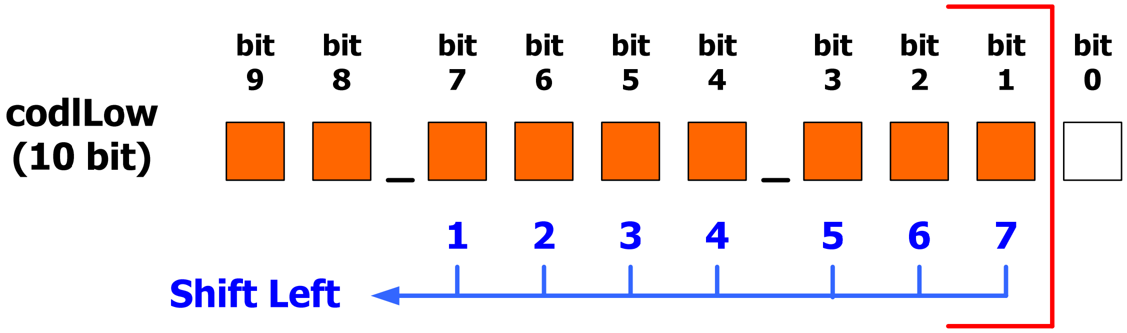


Figure 35 Illustration of the shifting left of codlLow

So the total cases can be estimated as follows:

Total cases

$$= \underset{(\text{shift}=7)}{2^9} + \underset{(\text{shift}=6)}{2^8} + \underset{(\text{shift}=5)}{2^7} + \underset{(\text{shift}=4)}{2^6} + \underset{(\text{shift}=3)}{2^5} + \underset{(\text{shift}=2)}{2^4} + \underset{(\text{shift}=1)}{2^3} + \underset{(\text{shift}=0)}{2^2}$$

$$= 1020 \text{ cases} \qquad \text{(Eq. 16)}$$

Besides the 1020 cases, the last remainder bitsOutstanding has to be considered. The range of remainder bitsOutstanding is 0 ~ 31, so there are 32 kinds of remainder bitsOutstanding. Therefore, the total number of the cases for renormalization process is 1020 x 32 = 32640 cases.

It is inefficient to build all of these cases due to too many cases causing the critical path too long. We analyze the utility rate of these cases based on the probability distribution of bitsOutstanding for typical video test sequence.

Table 16 The utility rate of each case for one symbol

bitsOutstanding	0	1	2	3	4	5	6	7
The numbers of case	52	355	298	187	80	32	12	4
The probability of appearance	51.39 %	24.37 %	12.12 %	6.05 %	3.02 %	1.51 %	0.76 %	0.38 %
Utility rate (probability/cases)	0.9883 %	0.0686 %	0.0407 %	0.0324 %	0.0378 %	0.0472 %	0.0633 %	0.095 %
Utility rank	1	4	6	8	7	5	3	2

Table 16 shows the analysis of utility rate for each case in one symbol, it is the same for all kinds of remainder bitsOutstanding. According to the utility rate of cases, we rank it in order. The cases whose utility rate is smaller than 0.041% are implemented with sequential circuit; namely, the bitsOutstanding accumulates itself one by one per cycle, and the generated bit-stream for one symbol is produced not only one cycle. The cases whose utility rate is greater than 0.041% to implement only with combinational circuit; namely, the generated bit-stream for one symbol is produced in only one cycle. The probability of these cases which are one executing cycle is about 78.41%.

Then, we have to consider the occupied probabilities of remainder bitsOutstanding. We only analyze the bigger probabilities of remainder bitsOutstanding; namely, the number of it is smaller than or equal to 7.

Table 17 The number of cases and the probabilities of the different containing range of remainder bitsOutstanding

the different containing range of remainder bitsOutstanding	the number of cases	Probability	case utility rate
(< 1) 0	455x 1= 455	78.41%x 51.37%= 40.28%	0.0885%
(< 2) 0+1	455x 2= 910	78.41%x 75.78%= 59.42%	0.0652%
(< 3) 0+1+2	455x 3= 1365	78.41%x 87.88%= 68.91%	0.0505%
(< 4) 0+1+2+3	455x 4= 1820	78.41%x 93.93%= 73.65%	0.0405%
(< 5) 0+1+2+3+4	455x 5= 2275	78.41%x 96.95%= 76.01%	0.0334%
(< 6) 0+1+2+3+4+5	455x 6= 2730	78.41%x 98.46%= 77.20%	0.0283%
(< 7) 0+1+2+3+4+5+6	455x 7= 3184	78.41%x 99.22%= 77.80%	0.0244%
(< 8) 0+1+2+3+4+5+6+7	455x 8= 3640	78.41%x99.60%=78.10%	0.0215%

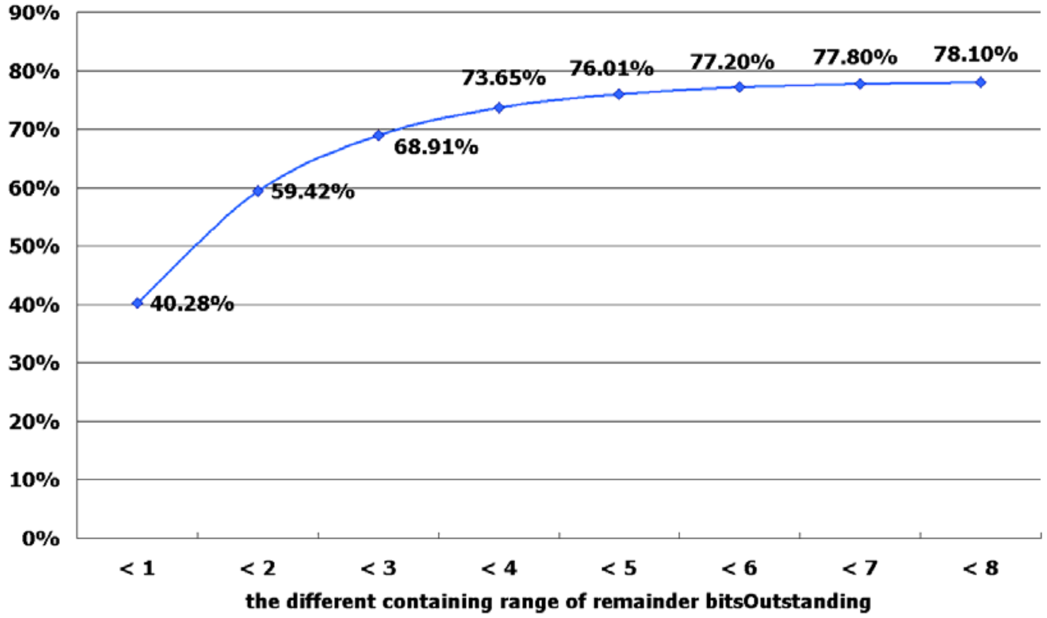


Figure 36 Probabilities of the different containing range of remainder bitsOutstaning based on Table 17

Table 17 shows the number of cases and the final probabilities of the different containing range of remainder bitsOutstanding. Figure 36 is the probability curve of it based on Table 17. The curve shows the probabilities will be saturation with the increasing of the range of remainder bitsOutstanding. So the containing range of remainder bitsOutstanding which is 3 is adopted by our design; namely, it supports the remainder bitsOutstanding equaling to 0, 1, and 2. The cases of it are 455 x 3 = 1365 cases, and these cases are implemented only with combinational circuit; namely, there are 68.91% renormalization process can be executed taking only one cycle.

Besides, the other cases are 32640 – 1365 = 31275 cases. These cases are not implemented only with combinational circuit due to too many and too inefficient for these cases. So these cases are implemented with sequential circuit.

According to the analysis of renormalization process mentioned above, we propose Case Efficiency Architecture shown in Figure 37 to promote the efficiency of the renormalization process.

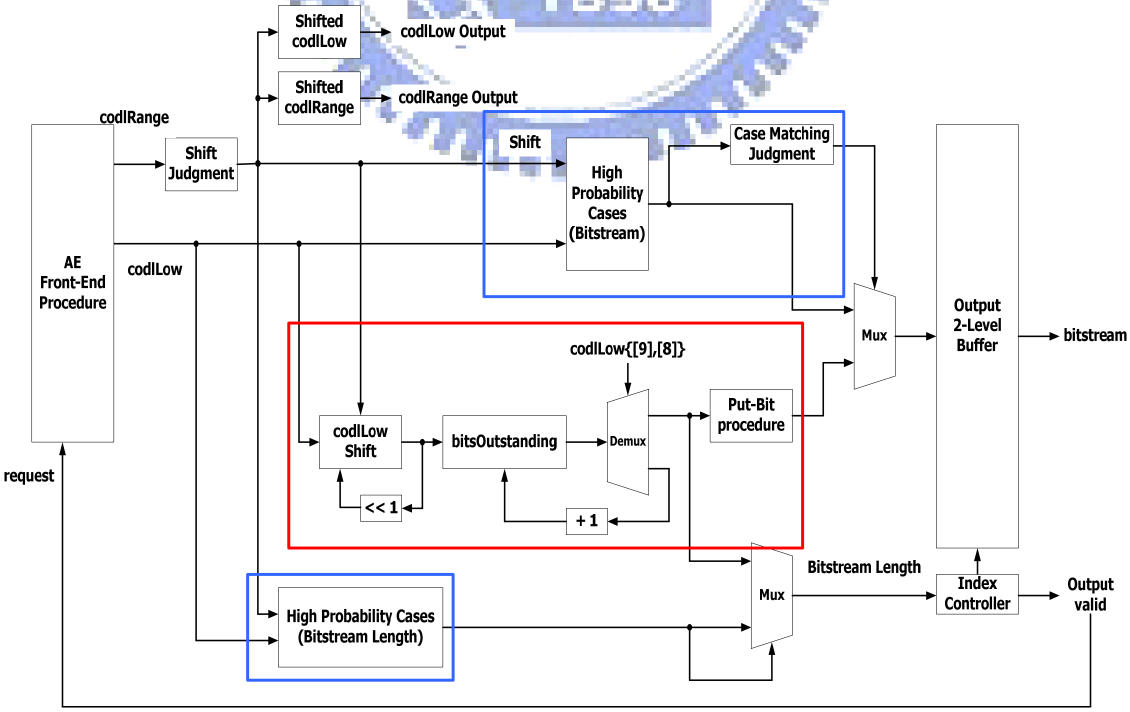
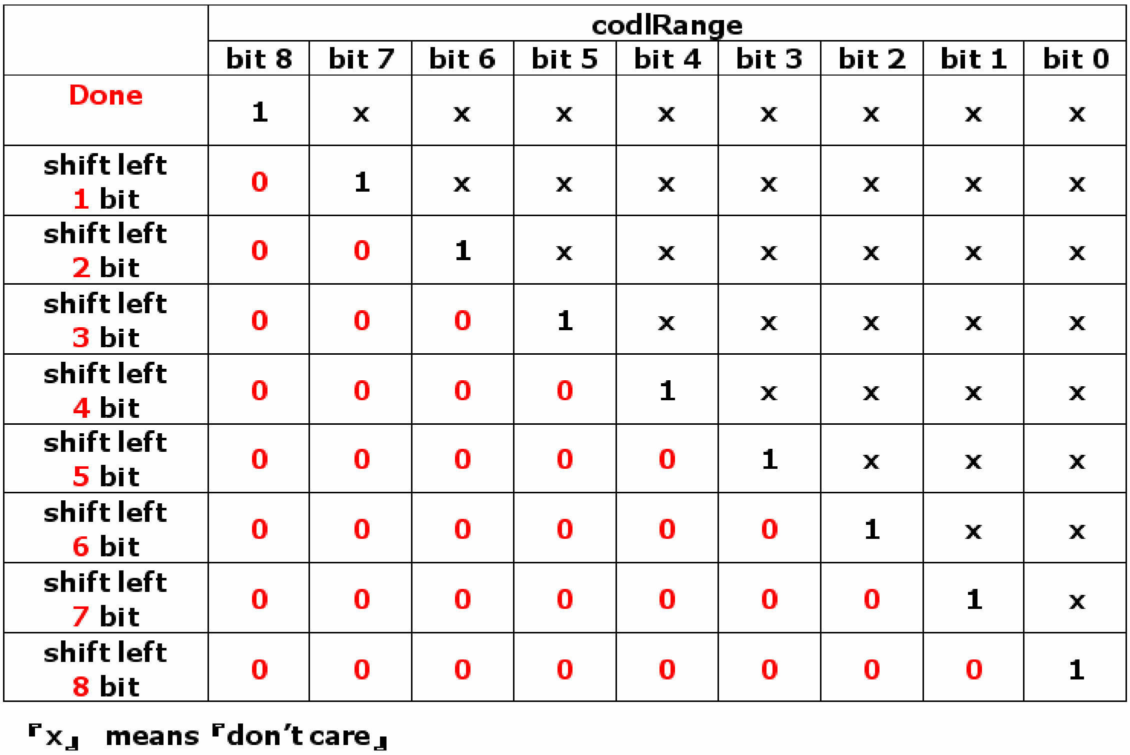


Figure 37 Case Efficiency Architecture for renormalization process

Figure 37 shows the Case Efficiency Architecture for renormalization process. The upper blue block is the high probability cases which are implemented only with combinational circuit. It can produce the bit-stream of current symbol in only one cycle. The generated bit-stream is variable length, so we design the control signal which is bit-stream length to put the bit-stream into the bit-stream output buffer. The bit-stream length is decided in lower blue block of Figure 37, and it is implemented only with combinational circuit, too. The red block is the low utility rate cases which are implemented with sequential circuit. Its bitsOutstanding accumulates itself one by one cycle, so generating the corresponding bit-stream for one symbol takes several cycles which is dominated by the shift number. The shift number is decided by current codlRange (9 bits), the relationship of them is shown in Table 18. The implementation of the Shift Judgement block shown in Figure 37 is based on the relationship. The relationship is decided by the codlRange decision branch shown in Figure 7 of Section 2.4.1. If the current colRange is smaller than 256_{10}, it has to be shift left until that it is not smaller than 256_{10} anymore. We simplify the process to be the rule shown in Table 18. According to the rule, the shift can be estimated in only one cycle by implementing it only with combinational circuit shown in the shift judgment block of Figure 37. For Table 18, the number of zero before the first "1" in MSB of the codlRange is the shift number. For example, if the current codlRange is 0_0010_1101, the corresponding shift is 3.

The conclusion of this section is that the proposed Case Efficiency Architecture can make almost 70% normal encoding process take only one cycle due to the efficiency of renormalization process is improved by Case Efficiency Architecture.

Table 18 The relationship between shift and codlRange



	codlRange								
	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Done	1	x	x	x	x	x	x	x	x
shift left 1 bit	0	1	x	x	x	x	x	x	x
shift left 2 bit	0	0	1	x	x	x	x	x	x
shift left 3 bit	0	0	0	1	x	x	x	x	x
shift left 4 bit	0	0	0	0	1	x	x	x	x
shift left 5 bit	0	0	0	0	0	1	x	x	x
shift left 6 bit	0	0	0	0	0	0	1	x	x
shift left 7 bit	0	0	0	0	0	0	0	1	x
shift left 8 bit	0	0	0	0	0	0	0	0	1

「x」 means 「don't care」

3.2.3.3 Cost issue in output interface

Each symbol (*bin* string) processed by arithmetic encoder will produce variable length bit-stream, and the variable length is mainly dominated by the number of last remainder bitsOutstanding. The value range of remainder bitsOutstanding which has been mentioned in Section 3.2.3.2 is 0 ~ 31. The variable length bit-stream causes the cost issue in bit-stream output buffer due to its large variable range in the length of bit-stream.

If we design two levels 32 bit bit-stream output buffer, the advantage of it is that all kinds of different length bit-stream can be written into it taking only one cycle. The drawback of it is leading to higher cost due to the 32 kinds of bit-stream lengths and the 32 kinds of position indexes. It causes 32 x 32 = 1024 cases for the bit-stream output buffer.

3.2.3.4 Cost efficiency design

According to the statistic of the remainder bitsOutstanding for the five video test sequences shown in Section 3.2.3.2, we find that the probability distribution of it is not uniform. So we can design the bit-stream output buffer which is cost efficiency based on the characteristic of probability distribution of the remainder bitsOutstanding.

Table 19 Probability of bitsOutstanding being greater than 7

	bitsOutstanding	<=7	>7
Low complexity and Slow motion	**Foreman (CIF)**	99.6120%	0.3880%
	Akiyo (CIF)	99.5903%	0.4097%
High complexity and Fast motion	**Football (CIF)**	99.6104%	0.3896%
	Stefan (CIF)	99.6186%	0.3814%
	Riverbed (1080HD)	99.6088%	0.3912%

Table 19 shows the statistic result of the five video test sequences, it is divided by the remainder bitsOutstanding being greater than 7. We can find that the probability of it being smaller than or equaling to 7 occupies about 99.6% for this five video test sequences. So we design the two levels 8 bit buffer as bit-stream output buffer for our CABAC encoder system.

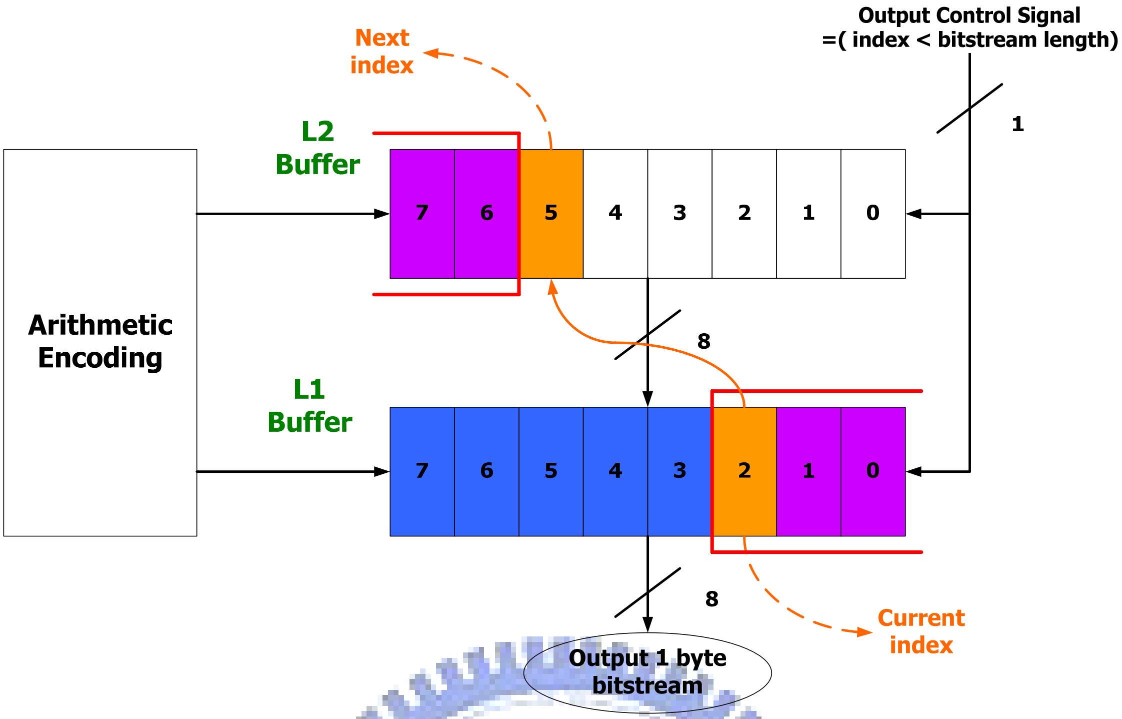


Figure 38 The proposed bit-stream output interface

Figure 38 is the proposed bit-stream output interface based on the analysis of the remainder bitsOutstanding shown in Table 19. The proposed bit-stream output buffer is composed of two levels 8 bit buffer. The reason of 8 bit is that the maximum reminder bitsOutstanding of 99.6% shown in Table 19 is 7, and the first bit of put bit procedure is 1 bit. The reason of the two levels is to ensure that it can receive any bit-stream whose length is smaller than or equal to 8 in only one cycle. Figure 38 shows an example, and the detailed description is shown as follows.

The operation of the bit-stream output buffer:

If the current situation is:

The current position index is: 2

The current generating bit-stream length is: 5 bit

(It is the situation of the position index being smaller than the bit-stream length, so the level 1 buffer is not sufficient to receive all the bit-stream, and the partial bit-stream has to be written to the level 2 buffer.)

The position index update: (2 – 5) = (0010 + 0011) = 0101

(Ignoring the MSB, and the remainder 3 bit is the next position index.)

Table 20 Required buffer slot for the different length bit-stream

Index of L1 cache	**bit-stream length**	**L1 buffer(bit)**								**L2 buffer(bit)**							
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
	0																
7	1	V															
	2	V	V														
	3	V	V	V													
	4	V	V	V	V												
	5	V	V	V	V	V											
	6	V	V	V	V	V	V										
	7	V	V	V	V	V	V	V									
	8	V	V	V	V	V	V	V	V								
6	1		V														
	2		V	V													
	3		V	V	V												
	4		V	V	V	V											
	5		V	V	V	V	V										
	6		V	V	V	V	V	V									
	7		V	V	V	V	V	V	V								
	8		V	V	V	V	V	V	V	V							
5	1			V													
	2			V	V												
	3			V	V	V											
	4			V	V	V	V										
	5			V	V	V	V	V									
	6			V	V	V	V	V	V								
	7			V	V	V	V	V	V	V							
	8			V	V	V	V	V	V	V	V						

4	1				V												
	2				V	V											
	3				V	V	V										
	4				V	V	V	V									
	5				V	V	V	V	V								
	6				V	V	V	V	V	V							
	7				V	V	V	V	V	V	V						
	8				V	V	V	V	V	V	V	V					
3	1					V											
	2					V	V										
	3					V	V	V									
	4					V	V	V	V								
	5					V	V	V	V	V							
	6					V	V	V	V	V	V						
	7					V	V	V	V	V	V	V					
	8					V	V	V	V	V	V	V	V				
2	1						V										
	2						V	V									
	3						V	V	V								
	4						V	V	V	V							
	5						V	V	V	V	V						
	6						V	V	V	V	V	V					
	7						V	V	V	V	V	V	V				
	8						V	V	V	V	V	V	V	V			
1	1							V									
	2							V	V								
	3							V	V	V							
	4							V	V	V	V						
	5							V	V	V	V	V					
	6							V	V	V	V	V	V				
	7							V	V	V	V	V	V	V			
	8							V	V	V	V	V	V	V	V		
0	1								V								
	2								V	V							
	3								V	V	V						
	4								V	V	V	V					
	5								V	V	V	V	V				
	6								V	V	V	V	V	V			
	7								V	V	V	V	V	V	V		
	8								V	V	V	V	V	V	V	V	

“V” denotes the required buffer slot

Table 20 shows all kinds of situations for the different bit-stream length and the different position index. The gray rows denote that the level 2 buffer needs to be used. If the level 1 buffer is full, its control signal is (position index < bit-stream length), and the bit-stream output interface outputs one 8 bit bit-stream to decoder meanwhile. At the same time, the level 2 buffer is duplicated to level 1 buffer. The position index needn't any modification due to the level 1 buffer and the level 2 buffer use the same position index.

For the bit-stream length being greater than 8, it will be divided by 8. For example, if the current bit-stream length is 28 bits, it will be divided by 8. Then we get (28 bits) = (8 bits) + (8 bits) + (8 bits) + (4 bits), taking these four set of sub-bit-stream into the bit-stream output buffer will take four cycle due to the maximum receiving bits of the bit-stream output buffer is 8 bits.

According to Table 19, the proposed bit-stream output interface can receive about 99.6% bit-stream in only one cycle.

Chapter 4
CABAC Codec System Architecture

This chapter focuses on the cost efficiency of the whole CABAC codec system. We propose some hardware sharing methods to reduce the cost of the CABAC codec. Besides, we also estimate the required memory which can support to 1080HD of our CABAC System.

The CABAC decoding process is contrary to the CABAC encoding process, the detail of it shown in Chapter 2. Because of it, there are many sub-modules can be hardware shared to reduce the cost of the CABAC codec.

The required memory of our CABAC codec system is mainly dominated by the encoded/decoded frame size. The bigger the frame size is, the more the memory requires. So the frame size and the required memory are direct proportion.

This chapter is organized as follows. In Section 4.1, we introduce the proposed hardware sharing methods and the CABAC codec system architecture after adopting these hardware sharing methods. Section 4.2 shows the required memory of our CABAC codec system.

4.1 Hardware Sharing Methods

The CABAC encoder is a table-based encoder, it has many tables to look up. All of these tables are the same as the CABAC decoder. The proposed strategies for it are shown as follows.

1. (1) The probability table (rangeTabLPS) and the probability index table (transIdxLPS and transIdxMPS) can be table reuse for arithmetic encoding/decoding.
2. (2) The context model can be reused.

Besides, the binary tree of binarization in the CABAC encoder is the same as the CABAC decoder. The proposed strategy for it is finite state machine (FSM) sharing for the binarization in encoder and the debinarization in decoder.

Table 21 The states of FSM which are similar procedures between binarization and debinarization

States of Debinarization	States of Binarization similar to Debinarization	States Description (Binarization)
mode_mbType		
mode_IntraPre		
mode_IntraCpred	writeCIPredMode_CABAC	encode the chroma intra prediction mode of an 8X8 block
mode_readDquant		
mode_readRefFrame	writeRefFrame_CABAC	encode the reference parameter of a given MB
mode_readField	writeFieldModeInfo_CABAC	encode the field mode info of a given MB
mode_CBPLuma	writeCBP_Luma	encode the coded block pattern of a macroblock
mode_CBPChroma	writeCBP_Chroma	
mode_ACStart		
mode_residual		
mode_mbSkip	writeMB_skip_flagInfo_CABAC	encode macroblock skip flag
mode_b8		
mode_mvd	writeMVD_CABAC	encode the motion vector data of a B-frame MB
mode_EOS		

The middle column of Table 21 shows the states of FSM which are similar procedure between binarization and debinarization, and we take these states to be FSM sharing.

We divide the whole CABAC encoding process shown as Figure 2 of Section 2.1 into three main parts. Figure 39 shows the three main parts of CABAC encoder.

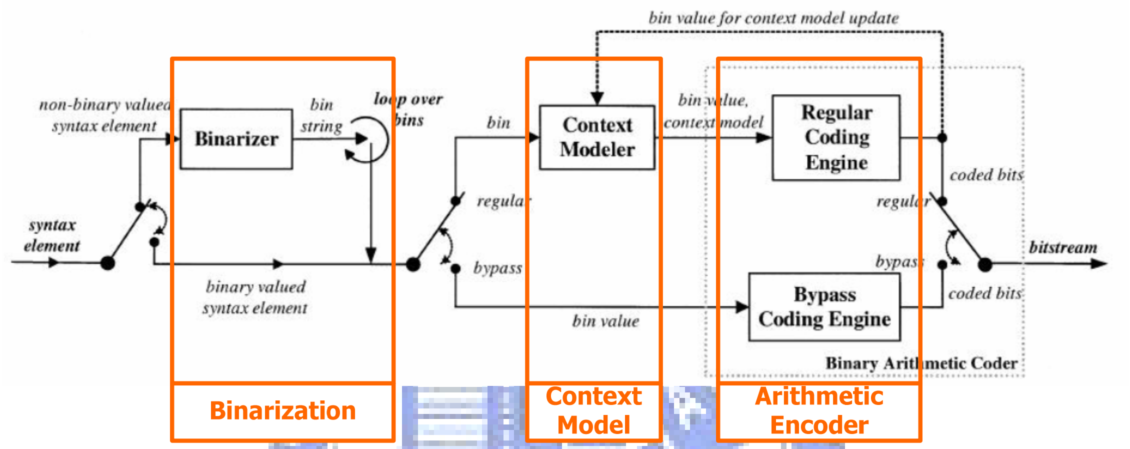


Figure 39 Three main parts of CABAC encoder

We take these three main parts to illustrate the hardware sharing. Figure 40 shows what can be hardware sharing between the CABAC encoder and the CABAC decoder for the three main parts.

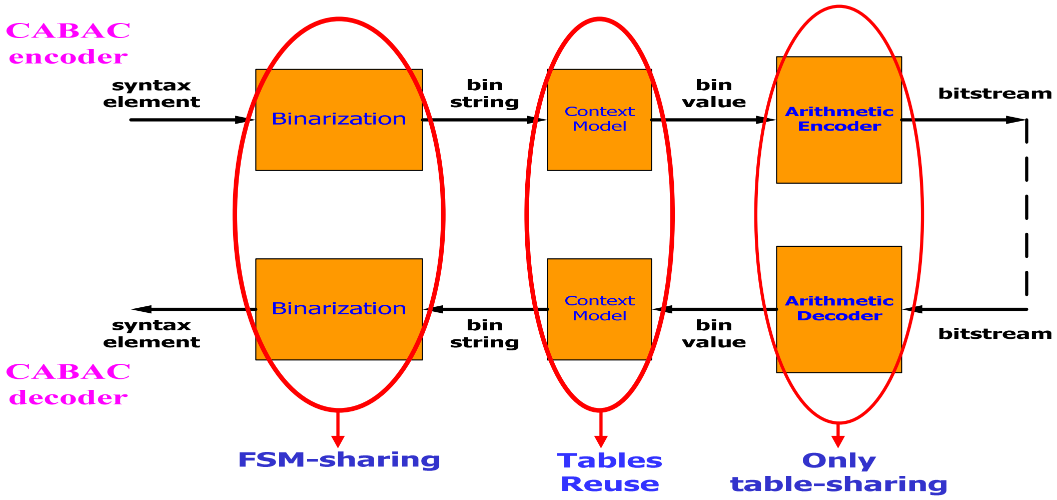


Figure 40 Illustration of hardware sharing for CABAC encoder and decoder

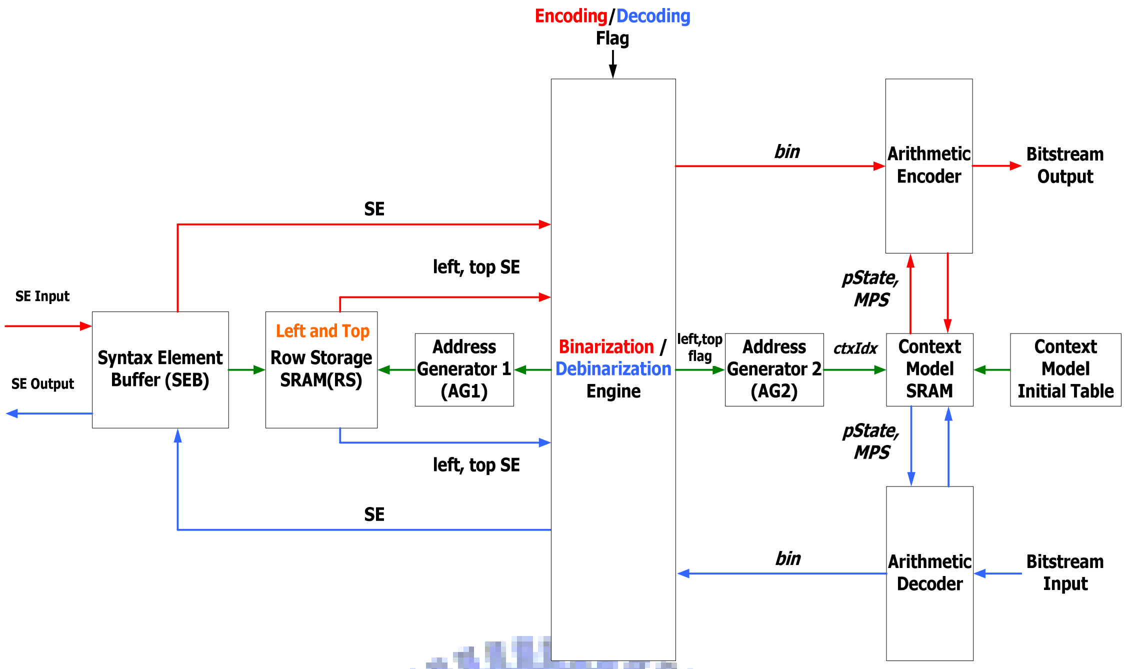


Figure 41 Architecture of the hardware sharing CABAC codec system

Figure 41 shows the CABAC codec system architecture after adopting the proposed hardware sharing methods. Comparing with the CABAC encoding system architecture in Figure 21 and the CABAC decoding system architecture Figure 22 of Section 3.1, we fine that the syntax element buffer, row storage SRAM, context model initial table and Context Model SRAM are hardware sharing. The red lines of it are the encoding procedure, and the blue lines mean the decoding procedure. The green lines denote the procedure executed in both the encoding and the decoding.

The hardware sharing CABAC codec system architecture not only combines the encoder and the decoder into CABAC codec but also it adopts some hardware sharing methods to reduce the cost of CABAC codec.

4.2 *Memory Requirement*

There are two SRAM modules in our CABAC codec system, one is context model dual ports SRAM, the other is row storage (RS) single port SRAM. The required memory of CABAC system is mainly dominated by the encoded/decoded frame size.

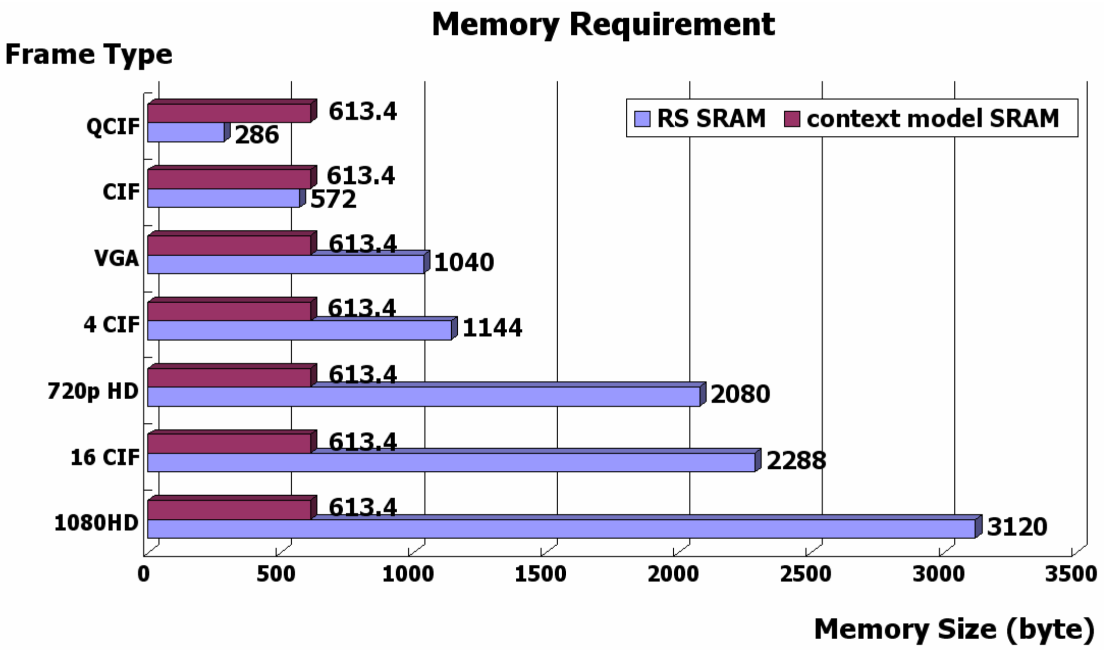


Figure 42 The required memory size of our CABAC codec system

Figure 42 shows the required memory size of our CABAC codec system at main profile encoding/decoding. For context model SRAM, its required memory size is fixed for different frame types. It always needs (399 x 7 bits) / 8 = 349.1 bytes for baseline profile and (701 x 7 bits) / 8 = 613.4 bytes for main profile. The required context model SRAM size which we choose is 613.4 bytes due to the target specification of our CABAC codec is for 1080 HD encoding/decoding belongs to main profile of H.264/AVC standard. For row storage (RS) SRAM, the required memory size is dominated by frame size. We find that the required memory size increases with the getting large frame size. For our CABAC codec system, we focus on encoding/decoding the video of 1080HD frame size, so the required memory size of our row storage SRAM is (120 x 208 bits) / 8 = 3120 bytes.

Chapter 5

Simulation and Implementation Result for Digital TV Applications

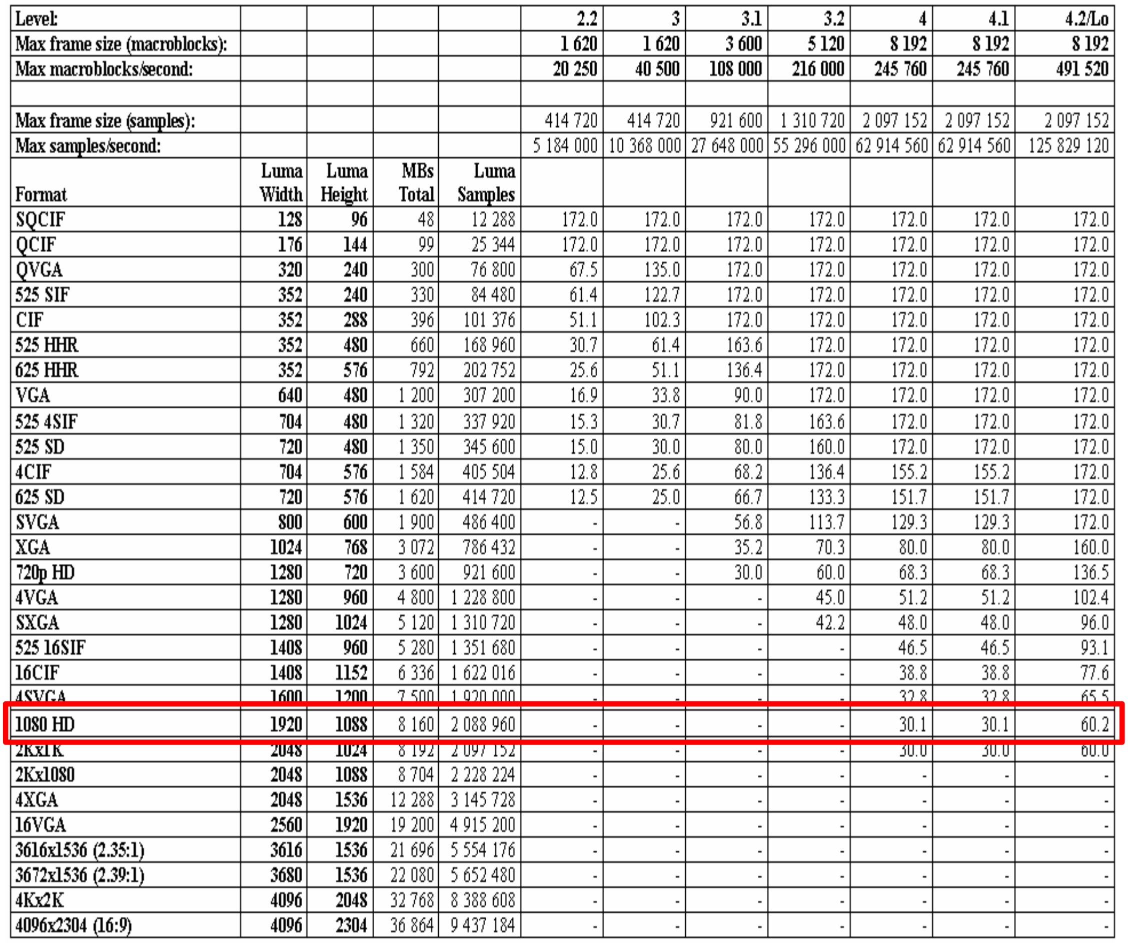
In this chapter, we show the simulation result by presenting the characteristic curves of our CABAC encoder. The characteristic curves of CABAC decoding are not shown here due to two reasons. One is that the CABAC decoding of our CABAC codec continues using [2] which we presented in August 2006. Its throughput can achieve the level 4.0; namely, it can decode the 1080HD@30fps at maximum bit-rate being 20Mbps. The detail of it is shown in [5]. We don't propose any additional high throughput method to promote the process efficiency of CABAC decoding in the thesis. The other is that the proposed three high throughput methods which are introduced in Chapter 3 are all for CABAC encoding process.

Our CABAC encoding system focuses on supporting the encoding of 1080HD, so we only take the 1080HD (1920x1088) video sequences as our simulation input. There are two characteristic curves of our CABAC encoding for each different video sequence. One is the throughput curve whose throughput unit is macroblock per second (MB/s), and the other is PSNR curve for Y (luminance). These two characteristic curves are obtained by modifying different quantization parameter (QP) under the maximum operating frequency (110MHz) of our CABAC encoder.

This chapter is organized as follows. In section 5.1, we introduce the specification of H.264 for the different levels. We take the maximum frame rates and the maximum bit rate to explain the differences for different levels. Section 5.2 shows the simulation and implementation result. We take two 1080HD video sequences to simulate the characteristic curves of our CABAC encoder.

5.1 *Specification of different levels*

Table 22 Maximum frame rates (fps) for some different frame types [1]



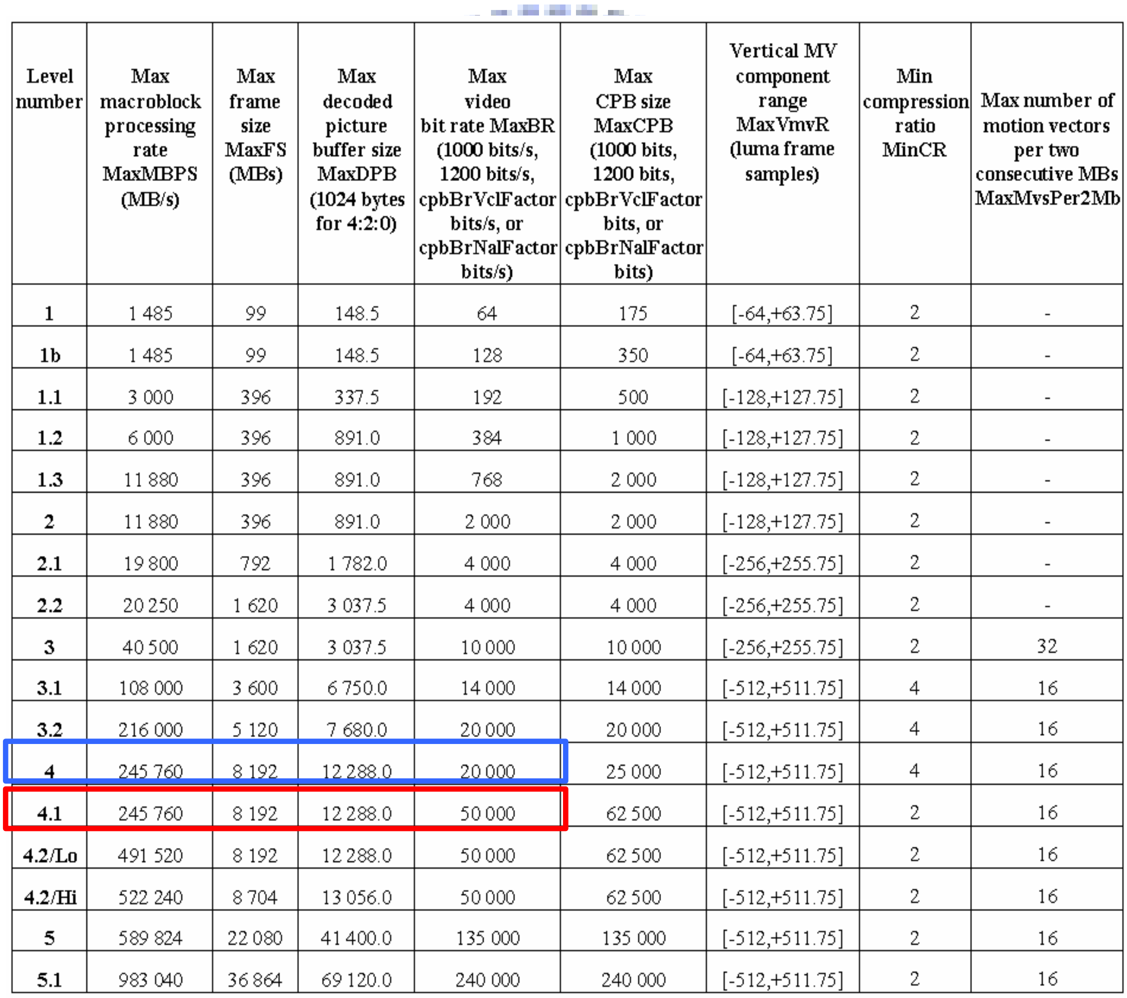
Level:					2.2	3	3.1	3.2	4	4.1	4.2/Lo
Max frame size (macroblocks):					1 620	1 620	3 600	5 120	8 192	8 192	8 192
Max macroblocks/second:					20 250	40 500	108 000	216 000	245 760	245 760	491 520
Max frame size (samples):					414 720	414 720	921 600	1 310 720	2 097 152	2 097 152	2 097 152
Max samples/second:					5 184 000	10 368 000	27 648 000	55 296 000	62 914 560	62 914 560	125 829 120
Format	Luma Width	Luma Height	MBs Total	Luma Samples							
SQCIF	128	96	48	12 288	172.0	172.0	172.0	172.0	172.0	172.0	172.0
QCIF	176	144	99	25 344	172.0	172.0	172.0	172.0	172.0	172.0	172.0
QVGA	320	240	300	76 800	67.5	135.0	172.0	172.0	172.0	172.0	172.0
525 SIF	352	240	330	84 480	61.4	122.7	172.0	172.0	172.0	172.0	172.0
CIF	352	288	396	101 376	51.1	102.3	172.0	172.0	172.0	172.0	172.0
525 HHR	352	480	660	168 960	30.7	61.4	163.6	172.0	172.0	172.0	172.0
625 HHR	352	576	792	202 752	25.6	51.1	136.4	172.0	172.0	172.0	172.0
VGA	640	480	1 200	307 200	16.9	33.8	90.0	172.0	172.0	172.0	172.0
525 4SIF	704	480	1 320	337 920	15.3	30.7	81.8	163.6	172.0	172.0	172.0
525 SD	720	480	1 350	345 600	15.0	30.0	80.0	160.0	172.0	172.0	172.0
4CIF	704	576	1 584	405 504	12.8	25.6	68.2	136.4	155.2	155.2	172.0
625 SD	720	576	1 620	414 720	12.5	25.0	66.7	133.3	151.7	151.7	172.0
SVGA	800	600	1 900	486 400	-	-	56.8	113.7	129.3	129.3	172.0
XGA	1024	768	3 072	786 432	-	-	35.2	70.3	80.0	80.0	160.0
720p HD	1280	720	3 600	921 600	-	-	30.0	60.0	68.3	68.3	136.5
4VGA	1280	960	4 800	1 228 800	-	-	-	45.0	51.2	51.2	102.4
SXGA	1280	1024	5 120	1 310 720	-	-	-	42.2	48.0	48.0	96.0
525 16SIF	1408	960	5 280	1 351 680	-	-	-	-	46.5	46.5	93.1
16CIF	1408	1152	6 336	1 622 016	-	-	-	-	38.8	38.8	77.6
4SVGA	1600	1200	7 500	1 920 000	-	-	-	-	32.8	32.8	65.5
1080 HD	1920	1088	8 160	2 088 960	-	-	-	-	30.1	30.1	60.2
2Kx1K	2048	1024	8 192	2 097 152	-	-	-	-	30.0	30.0	60.0
2Kx1080	2048	1088	8 704	2 228 224	-	-	-	-	-	-	-
4XGA	2048	1536	12 288	3 145 728	-	-	-	-	-	-	-
16VGA	2560	1920	19 200	4 915 200	-	-	-	-	-	-	-
3616x1536 (2.35:1)	3616	1536	21 696	5 554 176	-	-	-	-	-	-	-
3672x1536 (2.39:1)	3680	1536	22 080	5 652 480	-	-	-	-	-	-	-
4Kx2K	4096	2048	32 768	8 388 608	-	-	-	-	-	-	-
4096x2304 (16:9)	4096	2304	36 864	9 437 184	-	-	-	-	-	-	-

Table 22 shows the maximum frame rates whose unit is frames per second (fps) for different frame types. The target of our CABAC encoder is designed to support the 1080HD, so we only consider the 1080HD row which is highlighted by the red block.

In the 1080HD row we find that the maximum frame rates of the level 4.0 and the level 4.1 are the same, both of they are 30fps.

The difference of the level 4.0 and the level 4.1 can be observed in level limits shown in Table 23. The maximum macroblock processing rate (MB/s) of the level 4.0 and the level 4.1 are also the same, they are 245760 MB/s. The main difference of them is the maximum video bit rate. For the level 4.0, its maximum video bit rate is 20Mbps, and the level 4.1 is 50Mbps.

Table 23 level limits [1]



Level number	Max macroblock processing rate MaxMBPS (MB/s)	Max frame size MaxFS (MBs)	Max decoded picture buffer size MaxDPB (1024 bytes for 4:2:0)	Max video bit rate MaxBR (1000 bits/s, 1200 bits/s, cpbBrVclFactor bits/s, or cpbBrNalFactor bits/s)	Max CPB size MaxCPB (1000 bits, 1200 bits, cpbBrVclFactor bits, or cpbBrNalFactor bits)	Vertical MV component range MaxVmvR (luma frame samples)	Min compression ratio MinCR	Max number of motion vectors per two consecutive MBs MaxMvsPer2Mb
1	1 485	99	148.5	64	175	[-64,+63.75]	2	-
1b	1 485	99	148.5	128	350	[-64,+63.75]	2	-
1.1	3 000	396	337.5	192	500	[-128,+127.75]	2	-
1.2	6 000	396	891.0	384	1 000	[-128,+127.75]	2	-
1.3	11 880	396	891.0	768	2 000	[-128,+127.75]	2	-
2	11 880	396	891.0	2 000	2 000	[-128,+127.75]	2	-
2.1	19 800	792	1 782.0	4 000	4 000	[-256,+255.75]	2	-
2.2	20 250	1 620	3 037.5	4 000	4 000	[-256,+255.75]	2	-
3	40 500	1 620	3 037.5	10 000	10 000	[-256,+255.75]	2	32
3.1	108 000	3 600	6 750.0	14 000	14 000	[-512,+511.75]	4	16
3.2	216 000	5 120	7 680.0	20 000	20 000	[-512,+511.75]	4	16
4	245 760	8 192	12 288.0	20 000	25 000	[-512,+511.75]	4	16
4.1	245 760	8 192	12 288.0	50 000	62 500	[-512,+511.75]	2	16
4.2/Lo	491 520	8 192	12 288.0	50 000	62 500	[-512,+511.75]	2	16
4.2/Hi	522 240	8 704	13 056.0	50 000	62 500	[-512,+511.75]	2	16
5	589 824	22 080	41 400.0	135 000	135 000	[-512,+511.75]	2	16
5.1	983 040	36 864	69 120.0	240 000	240 000	[-512,+511.75]	2	16

5.2 Simulation and implementation result

We take two 1080HD video sequences to simulate their characteristic curves shown as follows.

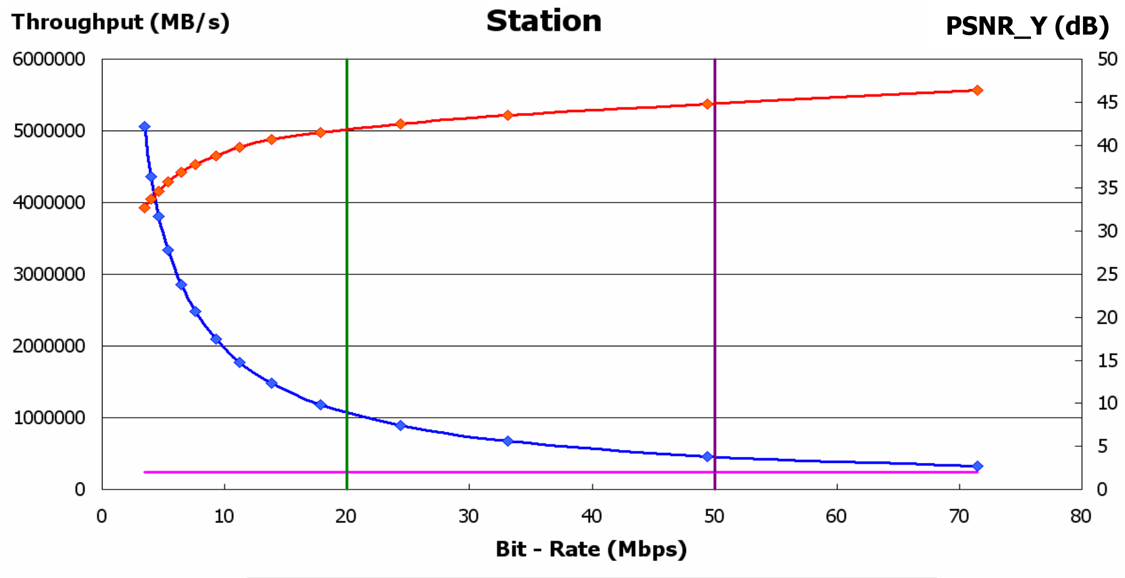


Figure 43 The characteristic curves of station video sequence (QP: 42 ~ 16)

Figure 43 shows the characteristic curves of station video sequence under the maximum operating frequency (110MHz) of our CABAC encoder, and the quantization parameter is modified from 42 to 16 whose interval is 2. According to these two characteristic curves of station video sequence, the proposed CABAC encoder can achieve 245760MB/s under 50Mbps; namely, it supports the specification of level 4.1. Besides, the PSNR of Y (luminance) is about 45 dB under the limits of level 4.1.

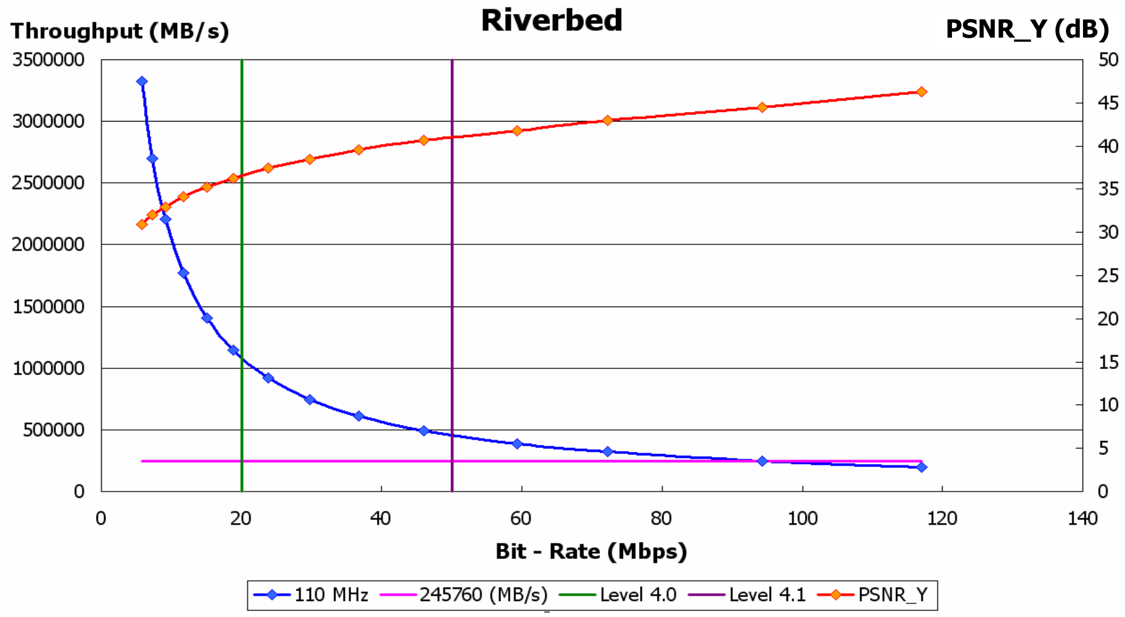


Figure 44 The characteristic curves of riverbed video sequence (QP: 42 ~ 16)

Figure 44 shows the characteristic curves of riverbed video sequence under 110MHz, and the quantization parameter is also modified from 42 to 16 whose interval is 2. The riverbed video can be regarded as worst case due to its fast motion (the bigger motion vector) causing massive data.

According to these two characteristic curves of riverbed video sequence, the proposed CABAC encoder also can achieve 245760MB/s under 50Mbps; namely, it also supports the specification of level 4.1. Besides, the PSNR of Y (luminance) is about 40 dB under the limits of level 4.1.

Table 24 The proposed arithmetic encoder comparing with the existing designs

	proposed	Ha'[7]	Shojaina'[8]	Núñez'[9]	Osorio'[10]	Osorio'[11]
Spec.	H.264	H.264	H.264	H.264	H.264	H.264
Function	Arithmetic Encoder	Arithmetic Encoder	Arithmetic Encoder	Arithmetic Encoder	Arithmetic Encoder	Arithmetic Encoder
Technology	0.18 μm UMC	Xilinx Virtex-II	0.18 μm TSMC	0.13 μm UMC	na	0.35 μm AMS
Frequency (MHz)	167	30	155	330	na	186
Encoding Throughput (symbol/cycle)	0.7866	0.2	1	1	0.91	1.9 to 2.3

Table 25 The proposed CABAC codec comparing with the existing CABAC designs

		proposed	Shojania'[12]
Spec.		H.264@MP	H.264@MP
Function		codec	encoder
Technology		0.18 μm UMC	0.18 μm
Frequency (MHz)		110	263
Encoding	Processing cycle (cycles/MB)	241 (QP=18)	na
	Encoding rate (Mbps)	91.79 (QP=18)	87
	Bit-rate (Mbps)	49.4 (QP=18)	na
	PSNR_Y	44.778 (QP=18)	na
Gate count (without Memory)		38436	na
Gate count (with Memory 1)		84873	area: 0.423 mm^2 (~43k)
Gate count (with Memory 1+2)		173303	na
Encoding target H.264 spec.		1080 HD@30fps (Level 4.1)	na

		proposed	Chen'[13]	Yang'[14]	Yu'[15]
Spec.		H.264 @MP	H.264 @MP	H.264 @MP	H.264 @MP
Function		codec	decoder	decoder	decoder
Technology		0.18 μm UMC	0.13 μm TSMC	0.18 μm TSMC	0.18 μm
Frequency (MHz)		110	200	120	150 (6.7ns)
Decoding	I (cycles/MB)	524 (QP26)	1661	463 (QP36)	na
	P (cycles/MB)	269 (QP26)	328	308 (QP26)	na
	B (cycles/MB)	141 (QP26)	576	254 (QP26)	na
	AVG (cycles/MB)	208 (QP26)	570	na	500
	Bit-rate (Mbps)	22.113	na	na	na
	PSNR_Y	38.487	na	na	na
Gate count (without Memory)		38436	na	na	area:0.3 mm^2 (~30k)
Gate count (with Memory 1)		84873	138226	83157	na
Gate count (with Memory 1+2)		173303	na	na	na
Decoding target H.264 spec.		1080 HD @30fps (Level 4.0)	720 HD @30fps (Level 3.1)	1080i HD @30fps (Level 4.0)	4Mbps (Level 2.2)

ps: QP: quantization parameter

Memory 1: context model SRAM

Memory 2: row storage SRAM (RS SRAM)

Table 24 shows the comparison of the proposed arithmetic encoder and the other state-of-the-art designs. Table 25 shows the comparison of the proposed CABAC codec for encoding and decoding respectively. The choice of the quantization parameter (QP) is based on the maximum bit-rate of level defined by H.264/AVC standard. The maximum video bit-rate of level 4.1 is 50Mbps, so the selected QP approximating the target bit-rate is 18 due to that the level 4.1 is the target specification of the proposed CABAC encoding. The maximum video bit-rate of level 4.0 is 20 Mbps, so the selected QP approximating the target bit-rate is 26 due to that the level 4.0 is the target specification of the proposed CABAC decoding.

The gate count of the context model dual-port SRAM is 46437 and of the row storage single-port SRAM is 88430. Besides, the RS SRAM is usually regarded as system level memory, so it isn't counted into CABAC memory size.

Table 26 Percentage of the cycle reduction for the proposed three throughput promoting methods

Proposed method	**Target encoding mode**	**Percentage of the cycle reduction (%)**
Multi-Symbol Architecture	Bypass	20.7 %
Pipeline Organization	Normal, Bypass, Terminal (mainly for Normal)	50.6%
Case Efficiency Architecture	Normal, Terminal (mainly for Normal)	47.5%

Table 26 shows the percentage of the cycle reduction for the proposed three throughput promoting methods. The detail estimations of executing cycle for these three throughput promoting methods are shown as follows respectively.

1.

For the Multi-Symbol Architecture (MSA) method:

Before adopting MSA:

84.21% bypass issue: 1 cycle

15.79% bypass issue: 3.776 cycles

Average 1.438 cycles

After adopting MSA:

94.04% bypass issue: 1 cycle

5.96% bypass issue: 3.36 cycles

Average 1.14 cycles

2.

For Pipeline Organization (PO) method:

Before adopting PO (under Normal encoding mode):

Average 3.99 cycles

After adopting PO (under Normal encoding mode):

Average 1.97 cycles

3.

For Case Efficiency Architecture (CEA) method

(The result is under having adopted PO method):

Before adopting CEA (under Normal encoding mode):

Average 1.97 cycles

After adopting CEA (under Normal encoding mode):

Average 1.04 cycles

Chapter 6
Conclusion and Future Work

6.1 Conclusion

We propose three high throughput methods such as multi-symbol architecture, pipeline organization and case efficiency architecture to improve the process efficiency of our CABAC encoder. Besides, we also propose the hardware sharing methods to reduce the cost of CABAC codec. The CABAC decoding of our CABAC codec continues using [2] which we proposed in August 2006. Its throughput can achieve the specification of level 4.0; namely, it supports to decode the 1080HD H.264 video sequence at 30 fps. The maximum video bit rate which it supports is 20 Mbps. The detail of it is shown in [5].

In our work, we implement a H.264@main profile CABAC codec under UMC 0.18μm CMOS Process. The total gate count is about 38436 without embedded SRAM and about 173303 with embedded SRAM. The maximum operating frequency is 110 MHz. Both CABAC encoding and CABAC decoding of our CABAC codec supports to encode/decode 1080HD H.264 video, and they achieve the different levels of H.264 specification. For decoding is level 4.0, and for encoding is level 4.1; namely, it can encode 1080 HD video at 30 fps. The maximum video bit rate which the encoder supports is 50Mbps, and its PSNR of Y (luminance) is about 44.8 dB. It achieves the throughput of 241 cycles per macroblock under the sequence type of

"IBBBPBBBP…".

6.2 *Future work*

Our CABAC codec can support the encoding and decoding for 1080HD at 30 fps, but it will be insufficient to satisfy the requirement of the future digital TV. In order to achieve the high quality video, the frame rate of 30fps doesn't correspond to the requirement of our digital TV market. The high resolution and high frame rate becomes the target of the human life. Hence, the large frame and high speed video playing is essentially for the digital TV application. To play the videos of 1080HD at 60fps is the basic requirement for the point view of CABAC. Thus, CABAC has to achieve the 1080HD of 60fps under the maximum bit-rate of 50,000,000 bit-per-second, which means the specification of level 4.2 for H.264/AVC is the future work for CABAC. Comparing to the level 4.0 and the level 4.1, it has to accelerate CABAC for 5 times. Hence, the acceleration of CABAC is the essential work in the advanced application.

6.3 *Discussion from H.264/AVC system view*

In this section, we discuss the proposed CABAC codec system from the whole H.264/AVC system view. The CABAC system is the sub-system of H.264/AVC system. This discussion focuses on the interface issue between the CABAC sub-system and the next sub-system, and we divided the discussion into the CABAC encoder and the CABAC decoder these two aspects.

For CABAC encoder, its last sub-system is DCT and Quantization. The CABAC encoder receives the syntax element from Quantization sub-system to encode to be bit-stream. Under considering the throughput matching issue, we design the syntax element buffer storage shown in Figure 41 to buffer the throughput mismatch. The CABAC encoder is the termination of the whole H.264/AVC encoding flow, and next to it is the forward error correction (FEC) which belongs to channel coding. We design two levels buffer storage between CABAC encoder and FEC, and the purpose of it is accumulating variable length bit-stream. Besides, the throughput mismatch buffer storage is design in FEC system.

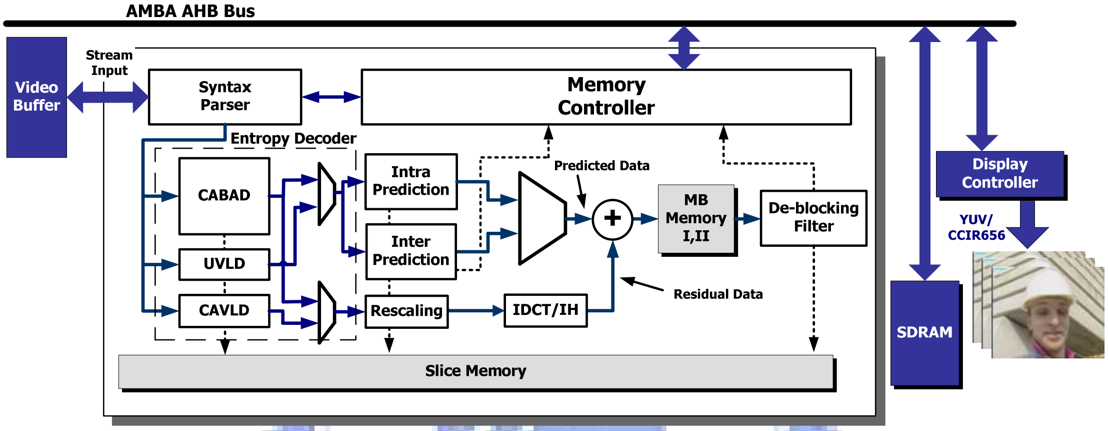


Figure 45 System block diagram of H.264/AVC decoding for main profile

For CABAC decoder (CABAD), the syntax parser dominates that which scheme (CABAD, UVLD, or CAVLD) is selected for current entropy decoding. Figure 45 shows the system block diagram of H.264/AVC for main profile. The syntax parser belongs to the system level control signal, and it employs in decoding the bit-stream on NAL layer, picture layer, and slice layer, shown as Figure 46. The syntax parser is also the top module to control all sub-system such as CABAD, VLD, intra-prediction, inter-prediction, IDCT, and so on. Hence, CABAD is the passive unit and is requested by the syntax parser and decodes the bit-stream of the macroblock layer in Figure 46. The bit-stream is also fetched through the syntax parser.

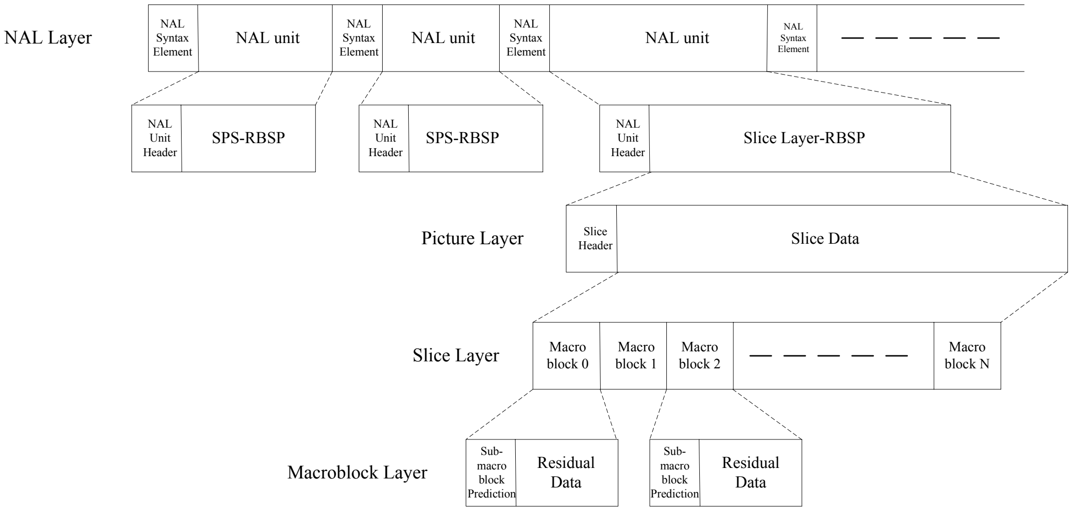


Figure 46 Bit-stream structure of H.264/AVC

The CABAC decoder is the first sub-system of the whole H.264/AVC decoding flow. As CABAC encoder, the syntax element buffer storage is also designed between the CABAC decoder and the next sub-system. Between the CABAC decoder and channel decoder (FEC), we design two levels buffer storage to buffer the interacting throughput mismatch.

Bibliography

---

[1] Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC), July 2004.

[2] Yi-Hong Huang and Ping-Chang Lin, “A High-Throughput SRAM-Based Context Adaptive Binary Arithmetic Decoder (CABAD) for H.264/AVC,” *The 17th VLSI Design/CAD Symposium*, August 2006

[3] J. Ostermann, J. Bormans, et al, “Video Coding with H.264/AVC: Tools, Performance, and Complexity,” *IEEE Mag. Circuits and Syst.*, vol. 4, pp. 7 – 28, First quarter 2004.

[4] Detlev Marpe, Member, IEEE, Heiko Schwarz, and Thomas Wiegand. “Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard,” pages: 209 – 217, *CSVT 2003*.

[5] Yi-Hong Huang, “Context Adaptive Binary Arithmetic Decoder of H.264/AVC for Digital TV Application,” *Thesis in NCTU EE*, July 2006.

[6] JVT H/AVC Reference Software JM 9.2

[7] V. H. Ha, W.–S. Shim, and J.–W. Kim, “Real-time MPEG-4 AVC/H.264 CABAC Entropy Coder,” in *Int. Conf. on Consumer Electronics (ICCE)*, pp. 255 – 256, 2005.

[8] H. Shojaina and S. Sudharsanan, “A VLSI Architecture for High Performance CABAC Encoding,” in *Proc. of the SPIE, Visual Communications and Image Processing*, pp. 1444 – 1454, 2005.

[9] J. Núňez and V. Chouliaras, "High-Performance Arithmetic Coding VLSI Macro for the H.264 Video Compression Standard," *IEEE Trans. Consumer Election.*, vol. 51, no. 1, pp. 144 – 151, February 2005.

[10] Roberto R. Osorio and Javier D. Bruguera, "Arithmetic Coding Architecture for H.264/AVC CABAC Compression System," *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, pp. 62 – 69, August 2004.

[11] Roberto R. Osorio and Javier D. Bruguera, "High-Throughput Architecture for H.264/AVC CABAC Compression System," *Circuits and Systems for Video Technology, IEEE Transactions on*, pp. 1376 – 1384, November. 2006.

[12] Hassan Shojania and Subramania Sudharsanan, "A High Performance CABAC Encoder," *IEEE-NEWCAS Conference, 2005. The 3rd International*, pp. 315 – 318, June 2005.

[13] Jian-Wen Chen, Cheng-Ru Chang, Youn-Long Lin, "A Hardware Accelerator for Context-Based Adaptive Binary Arithmetic Decoding in H.264/AVC," *ISCAS 2005. IEEE International Symposium on 23-26 May 2005*, pp. 4525 – 4528, 2005

[14] Yao-Chang Yang[1], Chien-Chang Lin[1], Hsui-Cheng Chang[1] et al, "A High Throughput VLSI Architecture Design for H.264 Context-Based Adaptive Binary Arithmetic Decoding with Look Ahead Parsing," *Multimedia and Expo, 2006 IEEE International Conference on*, pp. 357 – 360, July 2006.

[15] Wei Yu and Yun He, "A High Performance CABAC Decoding Architecture," *IEEE Transaction on Consumer Electronics*, pp. 1352 – 1359, November 2005

[16] L. Bottou, P. G. Howard, and Y. Bengio, "The Z-coder adaptive binary coder," *ln Proceedings of the Data Compression Conference*, pp. 13-- 22, March 1998.

作者簡歷

姓名　　：林秉昌

出生地　：台灣省台南市

出生日期：1976.05.25

學　歷：	1983.9～1989.6	高雄縣立阿蓮國民小學
	1989.9～1992.6	高雄縣立阿蓮國民中學
	1993.9～1996.6	國立台南二中
	2000.9～2004.6	私立義守大學 電機工程學系 學士
	2005.2～2007.1	國立交通大學 電機學院 IC設計產業研發碩士班 碩士

發表論文

- Yi-Hong Huang, Ping-Chang Lin, Chen-Yi Lee,” A High-Throughput SRAM-Based Context Adaptive Binary Arithmetic Decoder (CABAD) for H.264/AVC”, *Proceedings of the 17th VLSI/CAD Symposium*, August 2006.