

國立交通大學

電子工程學系電子研究所碩士班

碩士論文

H.264/AVC及SVC熵解碼器之分析與設計
Analysis and Design of Entropy Decoder for
H.264/AVC and Scalable Extension

研究生：廖元歆

指導教授：張添烜 教授

中華民國 九十九年 八月

H.264/AVC及SVC熵解碼器之分析與設計
Analysis and Design of Entropy Decoder for
H.264/AVC and Scalable Extension

研究生：廖元歆

Student：Yuan-Hsin Liao

指導教授：張添烜博士

Advisor：Dr. Tian-Sheuan Chang

國立交通大學

電子工程學系電子研究所碩士班

碩士論文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master of Science

in

Electronics Engineering

August 2010

Hsinchu, Taiwan, Republic of China

中華民國 九十九年八月

H.264/AVC 及 SVC 熵解碼器之分析與設計

研究生：廖元歆

指導教授：張添烜博士

國立交通大學

電子工程學系電子研究所碩士班

摘 要

近年來，由於 H.264/AVC 較以前的視訊標準有更佳的編碼效率，至今已被廣泛使用在視訊應用系統中。要想實現高解析度畫面即時解碼，熵解碼器的效能需求非常的高。因此，我們需要設計一個高效能的積體電路來加速熵解碼器的解碼速度。

本篇研究提出一個適用於 H.264/AVC 以及 SVC 的高產量熵解碼器硬體設計。首先，我們提出一個延遲均衡的雙符號內容適應性變動長度解碼器，並將解碼程序中多餘的解碼步驟省略以加速解碼的進行。工作頻率相較於傳統的設計可提高 21%，而整體產量相較於我們之前的設計可提升 28.2%。接著，針對 H.264/AVC 的另一種亂度編碼，我們提出一個以混合式記憶體為架構之高產量內容適應性二元算數解碼器。在整個解碼架構中，我們將語法單元剖析及其解碼進行合併，並提出以混合式記憶體為架構的雙符號平行解碼技術來加速解碼速度。更進一步的，我們利用一個有效率的預測機制以及透過數學上的轉換來提升解碼效能。

基於聯華電子 90 奈米製程，我們的內容適應性變動長度解碼器的最高工作頻率可達 390 MHz，13.88k 個邏輯閘。而我們的內容適應性二元算數解碼器的最高工作頻率可達 264 MHz，42.37k 個邏輯閘。我們的解碼器在節省了 48.6%的

硬體成本下的產量為每秒 451.4 百萬個符號，高於其他已被發表的設計。此外，我們將硬體設計拓展到 SVC。在工作頻率 135 MHz 下，我們所提出的熵解碼器可支援 3 層解析度，最高 1920x1080、三層播放頻率、最高每秒 60 張畫面、以及三層畫面品質的及時解碼。



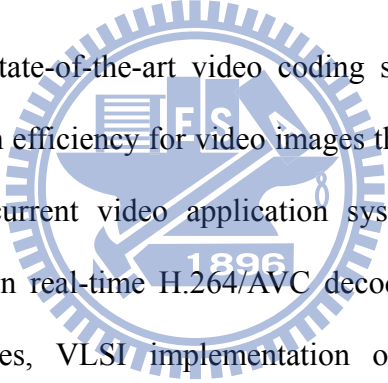
Analysis and Design of Entropy Decoder for H.264/AVC and Scalable Extension

Student: Yuan-Hsin Liao

Advisor: Dr. Tian-Sheuan Chang

Department of Electronics Engineering & Institute of Electronics
National Chiao-Tung University

Abstract

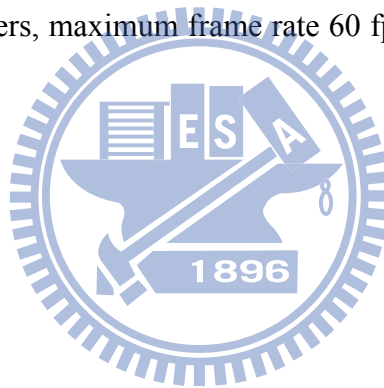


In recent years, the state-of-the-art video coding standard H.264/AVC which provides better compression efficiency for video images than the earlier standards has been widely adopted in current video application system. To satisfy the heavy performance requirement on real-time H.264/AVC decoding systems especially for large-scale video sequences, VLSI implementation of the entropy decoder is necessary since it dominates the overall decoder system performance.

In this thesis, we propose a high-throughput and fully hardwired entropy decoder for H.264/AVC and its scalable extension. First, we present a delay balanced two-level CAVLC decoder with 21% shorter critical path delay in comparison to traditional two-level decoder. Furthermore, a skipping mechanism is adopted to remove unnecessary decoding processes. The overall CAVLC throughput is 28.2% better than our previous design. Second, for the CABAC decoder, we propose a high throughput CABAC decoding design which combines SE parsing and decoding with a new hybrid memory two-symbol parallel decoding technique to accelerate the decoding speed while reducing the hardware cost. Further speedup is achieved to

avoid stalls for most of the cases by the prediction-based method. In addition, an efficient mathematical transform method is also proposed to further decrease the critical path delay of two-symbol binary arithmetic decoding procedure by 28%.

The proposed entropy decoder is implemented by UMC 90nm technology and experimental results show that our CAVLC decoder can operate at 390 MHz with 13.88k gate count, besides, our CABAC decoder can operate at 264 MHz with 42.37k gate count, and the throughput is 451.4 Mbin/sec, which surpasses previous design with 48.6% hardware cost saving. Furthermore, we extend our entropy decoder towards SVC extension of H.264/AVC. At the working frequency 135 MHz, our proposed entropy decoder can support 3 spatial layers, maximum resolution 1920x1080, 3 temporal layers, maximum frame rate 60 fps, and 3 CGS quality layers real-time SVC decoding.



誌 謝

在此首先要感謝我的指導教授—張添烜博士。在這兩年的研究期間，不論是在課業方面的問題，研究上遭遇到的困難，甚至是在面對人生未來時所感到的迷惑，老師總是會在我有需要的時候為我提供幫助，給予我很多建議與想法。

同時也要感謝我的口試委員們，中央大學電機工程系蔡宗漢教授及交通大學電子工程系李鎮宜教授，感謝兩位能從百忙中專程抽空過來指導，教授們寶貴的意見與指教將使本篇論文更臻完備。

接著我要謝謝實驗室的同仁們。感謝李國龍學長，從我申請上研究所之後就進行前期的指導，引領我進入視訊處理這門學問的殿堂，並教導我如何做研究及找資料。感謝曾宇晟學長，在研究上以及軟體工具使用上給了我相當大的幫助。感謝陳之悠學長、許博淵學長、沈孟維學長以及黃筱珊學姊，你們所教導我的硬體設計程式技巧，至今仍讓我受用無窮。再來要感謝我的研究夥伴陳宥辰，在與你共事的過程中讓我學到了很多東西，也成長了很多。此外要感謝其他實驗室成員，張彥中學長、王國振學長、還有許博雄、洪瑩蓉、陳奕均，有了你們的鼓勵才得以讓這篇論文能夠順利完成。白駒過隙，碩士班短短兩年的時光一眨眼就過去了，但與各位一同努力一同歡笑的日子將成為我一生中難以遺忘的回憶。

最後我要感謝我的父親、母親以及女友阿兔，你們支持與關心，是我能夠順利完成學業的最大動力。

謹以這篇論文獻給所有愛我以及支持我的人。

Contents

CHAPTER 1 INTRODUCTION	1
1.1 Motivation and Contribution	1
1.2 Thesis Organization	2
CHAPTER 2 OVERVIEW OF CAVLC	3
2.1 Context-based Adaptive Variable Length Coding.....	3
2.1.1 CAVLC Decoding Flow.....	4
2.2 Design Challenges and Related Works	7
CHAPTER 3 OVERVIEW OF CABAC.....	10
3.1 Arithmetic Coding	10
3.2 Context-based Adaptive Binary Arithmetic Coding	12
3.2.1 Binarization.....	13
3.2.2 Context Modeling.....	17
3.2.3 Adaptive Binary Arithmetic Coding	19
3.3 CABAC Decoding Algorithm Overview	23
3.4 Design Challenges and Related Works	26
CHAPTER 4 PROPOSED ENTROPY DECODER.....	30
4.1 Proposed CAVLC Decoder.....	31
4.1.1 Analysis.....	32
4.1.2 Proposed Delay Balanced Two-level Decoder Architecture	37
4.1.3 CAVLC Decoding Architecture Design.....	39
4.1.4 Experimental Results.....	42
4.2 Proposed CABAC Decoder	46
4.2.1 Analysis.....	46
4.2.2 MCS Stage.....	52
4.2.3 Context Model Memory Design	58
4.2.4 TSBAD Stage.....	62
4.2.5 Experimental Results.....	66
CHAPTER 5 EXTENDING TOWARDS SVC.....	71
5.1 Design target and Design Challenges	71
5.2 Proposed Entropy Decoder for SVC.....	72
CHAPTER 6 CONCLUSION AND FUTURE WORK.....	78
6.1 Conclusion	78

6.2 Future Work79

REFERENCE.....81

BIOGRAPHICAL NOTES84



List of Figures

(Chapter2)

Fig. 1	CAVLC decoding flow	6
Fig. 2	Transmitted bitstream for a 4 x 4 residual block	7

(Chapter3)

Fig. 3	Example for interval subdivision	11
Fig. 4	Recursive interval subdivision for the sequence (C, B, C, E)	11
Fig. 5	CABAC encoder block diagram.....	13
Fig. 6	Pseudo code for k-th order Exp-Golomb code construction	16
Fig. 7	Neighboring syntax elements involved in context model selection of current syntax element	19
Fig. 8	Probability transition rule.....	20
Fig. 9	Flow diagram of binary arithmetic encoding process. (a) Regular coding mode. (b) Bypass coding mode	21
Fig. 10	Flowchart of (a) renormalization process and (b) PutBit(B)	22
Fig. 11	CABAC parsing flow	24
Fig. 12	Flow diagram of (a) regular bin decision process, (b) renormalization process, and (c) bypass bin decision process.....	26
Fig. 13	Pipelining scheme of CABAC decoding.....	27
Fig. 14	Data hazard caused by significance map. (a) 4x4 residual block. (b) Flow diagram of the CABAC decoding scheme for significance map. (c) Example for decoding the significance map. (d) Illustration of cycle stall of CABAC decoding	28

(Chapter4)

Fig. 15	Framework of proposed entropy decoder.....	31
Fig. 16	Original level decoding procedure defined in H.264/AVC standard.....	35
Fig. 17	MSD decoding procedure.....	36
Fig. 18	Modified level decoding procedure with MSD algorithm.....	37
Fig. 19	Proposed delay balanced two-level decoding architecture.....	39
Fig. 20	Proposed CAVLC decoder.....	41
Fig. 21	Residual block reconstruction architecture	42
Fig. 22	SE parsing flow for the H.264/AVC	48

Fig. 23 Proposed CABAC decoder architecture..... 52

Fig. 24 Pipeline scheduling of (a) prediction miss and (b) prediction hit..... 54

Fig. 25 Memory operation in the significance map decoding process..... 61

Fig. 26 Mathematical reordering. (a) $O-(R-R_{LPS})$. (b) $(O-R)+R_{LPS}$ 65

Fig. 27 Mathematical transform for the second bin decision process..... 65

Fig. 28 Architecture of proposed two-symbol arithmetic decoding engine..... 66

(Chapter5)

Fig. 29 Framework of proposed entropy decoder for SVC 74



List of Tables

(Chapter2)

Table 1	CAVLC DECODING PROCEDURE FOR THE 4 x 4 RESIDUAL BLOCK DEPICTED IN FIG. 2.....	7
---------	---	---

(Chapter3)

Table 2	DECODING PROCEDURE FOR INPUT NUMBER.....	12
Table 3	UNARY BINARIZATION.....	14
Table 4	TRUNCATED UNARY BINARIZATION.....	14
Table 5	FIXED-LENGTH BINARIZATION.....	15
Table 6	UEG3 BINARIZATION FOR ABSOLUTE VALUES OF MOTION VEXTOR DIFFERENCES.....	16
Table 7	SYNTAX ELEMENT AND CORRESPONDING CONTEXT INDICES.....	17
Table 8	CONTEXT CATEGORY DEPENDING ON SYNTAX ELEMENTS AND BLOCK TYPES.....	19

(Chapter4)

Table 9	THRESHOLD VALUE FOR SUFFIXLENGTH TRANSITION.....	35
Table 10	EXAMPLE OF RESIDUAL BLOCK RECONSTRUCTION PROCESS.....	42
Table 11	COMPARISON OF CAVLC DECODING PERFORMANCE.....	43
Table 12	CAVLC DECODER IMPLEMENTATION RESULT COMPARISONS DIFFERENT DESIGNS.....	44
Table 13	MAXIMUM FRAME RATES FOR SOME EXAMPLE FRAME SIZES.....	44
Table 14	WORKING FREQUENCY FOR DIFFERENT LEVEL CONDITIONS.....	46
Table 15	STATISTICAL RESULT OF BIN DISTRIBUTION WITH I CODING STRUCTURE AND QP28.....	48
Table 16	STATISTICAL RESULT OF BIN DISTRIBUTION WITH I CODING STRUCTURE AND QP20.....	49
Table 17	STATISTICAL RESULT OF BIN DISTRIBUTION WITH I CODING STRUCTURE AND QP12.....	49
Table 18	STATISTICAL RESULT OF BIN DISTRIBUTION WITH IPPP CODING STRUCTURE AND QP28.....	49
Table 19	STATISTICAL RESULT OF BIN DISTRIBUTION WITH IPPP CODING STRUCTURE AND QP20.....	50
Table 20	STATISTICAL RESULT OF BIN DISTRIBUTION WITH IPPP CODING STRUCTURE AND QP12.....	50
Table 21	STATISTICAL RESULT OF BIN DISTRIBUTION WITH IBPPP CODING STRUCTURE AND QP28.....	51
Table 22	STATISTICAL RESULT OF BIN DISTRIBUTION WITH IBPPP CODING STRUCTURE AND QP20.....	51

Table 23	STATISTICAL RESULT OF BIN DISTRIBUTION WITH IBBBP CODING STRUCTURE AND QP12	52
Table 24	IMPROVEMENT OF PREDICTION ACCURACY USING THE PROPOSED METHODS WITH I CODING STRUCTURE	55
Table 25	IMPROVEMENT OF PREDICTION ACCURACY USING THE PROPOSED METHODS WITH IPPP CODING STRUCTURE.....	55
Table 26	IMPROVEMENT OF PREDICTION ACCURACY USING THE PROPOSED METHODS WITH IBBBP CODING STRUCTURE	56
Table 27	BIN INDEX TRANSITION RELATION IN SIGNIFICANCE MAP.....	57
Table 28	CONTENT OF SRAM AFTER REORGANIZATION OF OUR PROPOSAL	61
Table 29	CONTENT OF REGISTER AFTER REORGANIZATION OF OUR PROPOSAL	62
Table 30	CABAC DECODING PERFORMANCE OF THE PROPOSED ARCHITECTURE WITH I CODING STRUCTURE	67
Table 31	CABAC DECODING PERFORMANCE OF THE PROPOSED ARCHITECTURE WITH IPPP CODING STRUCTURE	68
Table 32	CABAC DECODING PERFORMANCE OF THE PROPOSED ARCHITECTURE WITH IBBBP CODING STRUCTURE	69
Table 33	CABAC DECODER IMPLEMENTATION RESULT COMPARISONS OF DIFFERENT DESIGNS	69
Table 34	WORKING FREQUENCY FOR DIFFERENT LEVEL CONDITIONS	70
 <i>(Chapter5)</i>		
Table 35	CONTENT OF SRAM FOR SVC QUALITY ENHANCEMENT LAYER	74
Table 36	CONTENT OF REGISTER FOR SVC QUALITY ENHANCEMENT LAYER.....	75
Table 37	CONTENT OF SRAM FOR SVC BASE LAYER.....	75
Table 38	CONTENT OF REGISTER FOR SVC BASE LAYER	76
Table 39	SYNTHESIS RESULTS	76



Chapter 1 INTRODUCTION

H.264/AVC is the state-of-the-art video coding standard developed by the Joint Video Team (JVT) of ISO/IEC Moving Picture Experts Group and the ITU-T Video Coding Experts Group (MPEG and VCEG). With many advanced techniques, it provides better compression efficiency for video than the earlier MPEG-4 and H.263 standards do. Recently, H.264/AVC has been widely adopted in current video application system such as Blu-ray Disc, Youtube, television service, and real-time videoconferencing.

H.264/AVC specifies two entropy coding tools: Context-based Adaptive Variable Length Coding (CAVLC), and Context-based Adaptive Binary Arithmetic Coding (CABAC) [1], [2]. Both methods employ context-based adaptive modeling in their entropy coding framework and achieve better compression efficiency compared to previous video coding standards. In CAVLC, an adaptive VLC table switching method depending on already coded symbols is used, and in CABAC, an adaptive probability model estimation technique is utilized for binary arithmetic coding. For the reason that the adaptation of CAVLC can not perfectly match actually conditional symbol statistics and the limitation of 1 bit/symbol imposed on variable length codes, CABAC can achieve averaged bit-rate savings of 9% to 14% at the cost of higher computation complexity in comparison to CAVLC [3].

1.1 Motivation and Contribution

In recent years, as network transmission speed rises and high-definition television gains popularity, the demand for better visual quality grows fast. That means video application system is expected to support high-definition (HD) resolution

encoding and decoding. In addition to the heavy decoding requirement of H.264/AVC, this trend leads to the result that more data has to be processed in the same time for video decoders, and makes it more difficult to work in real-time for CPUs. In that event, it is necessary to accelerate the decoding speed of entropy decoder with hardware since its throughput dominates the overall decoder system performance. However, the inherently strong data dependency significantly restricts the throughput of entropy decoder and is generally considered as the main design challenge in hardware implementation. In order to achieve high decoding performance and low hardware cost real-time entropy decoding systems, a fully hardwired entropy decoder is proposed in this thesis.

1.2 Thesis Organization

The rest of this thesis is organized as follows. We briefly describe the entropy codec (CAVLC and CABAC) and their design challenges in hardware implementation in Chapter 2 and Chapter 3, respectively. In Chapter 4, the proposed entropy decoding architecture is presented and we provide simulation results to demonstrate the performance of our entropy decoder design. In Chapter 5, we extend our proposed entropy decoder towards the Scalable Video Coding (SVC) extension of the H.264/AVC standard. Finally, the conclusion is given in Chapter 6.

Chapter 2 OVERVIEW OF CAVLC

Variable-length coding (VLC) is an entropy coding method that converts each data symbol to a variable length codeword, and achieves data compression by utilizing the various probabilities of occurrence of data symbols. Symbols with high probabilities of occurrence are represented by short codewords while symbols with low probabilities of occurrence are represented by long codewords. There are two constraints on the VLC, one is that the bit string must consist of integral number of bits, another one is that each codeword must be uniquely decodable.

2.1 Context-based Adaptive Variable Length Coding

CAVLC and Exp-Golomb coding are the baseline entropy coding methods of H.264/AVC. In spite of the advantage of Exp-Golomb coding in computational efficiency, the compression efficiency is not good enough for real application alone. To enhance the compression efficiency, a more efficient entropy coding technique CAVLC is designed for encoding quantized transformed coefficients of 4×4 and 2×2 residual blocks by taking advantage of several characteristics of quantized blocks. After decorrelated by the Discrete Cosine Transform (DCT) and quantization, most of the quantized coefficients are zero while a few nonzero coefficients are clustered around the top left of the block. Afterward, by a reordering, nonzero coefficients are grouped together and the level of nonzero coefficients tends to be larger at the low frequencies (start of the reordered array) and smaller toward the high frequencies (end of the reordered array). Moreover, high-frequency nonzero coefficients are often a series of ± 1 (*TrailingOnes*). To efficiently represent the large number of zeros, a

run-level coding technique can be applied to reduce the redundancy of the data symbols. However, *Run* and *Level* are not quite correlated. Consequently, to achieve better compression efficiency, *Run* and *Level* are coding separately in CAVLC.

Distinct from conventional VLC that VLC table is unique; CAVLC switches VLC tables for different syntax elements relying on already transmitted symbols. That is why it is named context-based adaptive. Although better compression efficiency is achieved by exploiting inter-symbol redundancies, the rise in computational complexity and data dependency imposed on the CAVLC decoder makes it hard to be speeded up by parallelism and pipelining. In the following, the decoding flow of CAVLC alone with its design challenges is discussed in more detail.

2.1.1 CAVLC Decoding Flow

A residual block is represented by five types of SEs in CAVLC. These syntax elements are defined as follows:

- 1) *coeff_token*: This syntax element indicates the total number of nonzero coefficients (*TotalCoeffs*) including *TrailingOnes*. Since the coding units of CAVLC are 4 x 4 and 2 x 2 blocks, *TotalCoeffs* can be any value from 0 to 16 and *TrailingOnes* can be anything from 0 to 3. There are three variable-length codeword tables and a fixed-length codeword table using for coding *coeff_token*. The choice of look-up table depends on the total number of nonzero coefficients to the left and on top of the current block, nA and nB respectively.
- 2) *trailing_ones_sing_flag*: This 1-bit syntax element indicates the sign of *TrailingOnes*, and is coded in reverse order.
- 3) *level*: The syntax element *level* represents the value of remaining nonzero coefficients and is also coded in reverse order. Each *level* is composed of a

prefix part (*level_prefix*) and a suffix part (*suffix_part*).

- 4) *total_zeros*: The sum of zero coefficients, except for zeros after the last nonzero coefficient, is represented by this syntax element. The choice of VLC table depends on the total number of nonzero coefficients of the current block.
- 5) *run_before*: Number of zeros preceding each nonzero coefficient is encoded as this syntax element. The VLC table for coding each *run_before* is chosen according to the number of zeros left (*zerosLeft*).

Fig. 1 shows the flow diagram of CAVLC decoding. The decoding process consists of six steps: *coeff_token* parsing, *trailing_ones_sing_flag* parsing, *level* parsing, *total_zeros* parsing, *run_before* parsing, and residual block reconstruction. Table 1 shows an example for the decoding procedure of a CAVLC coded residual block as depicted in Fig. 2 and its corresponding decoded information. The input bitstream provided for CAVLC decoder is “00001000_11100101_11101101”, after the decoding procedure, the 4 x 4 residual block, “0, 3, 0, 1, -1, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0”, is reconstructed.

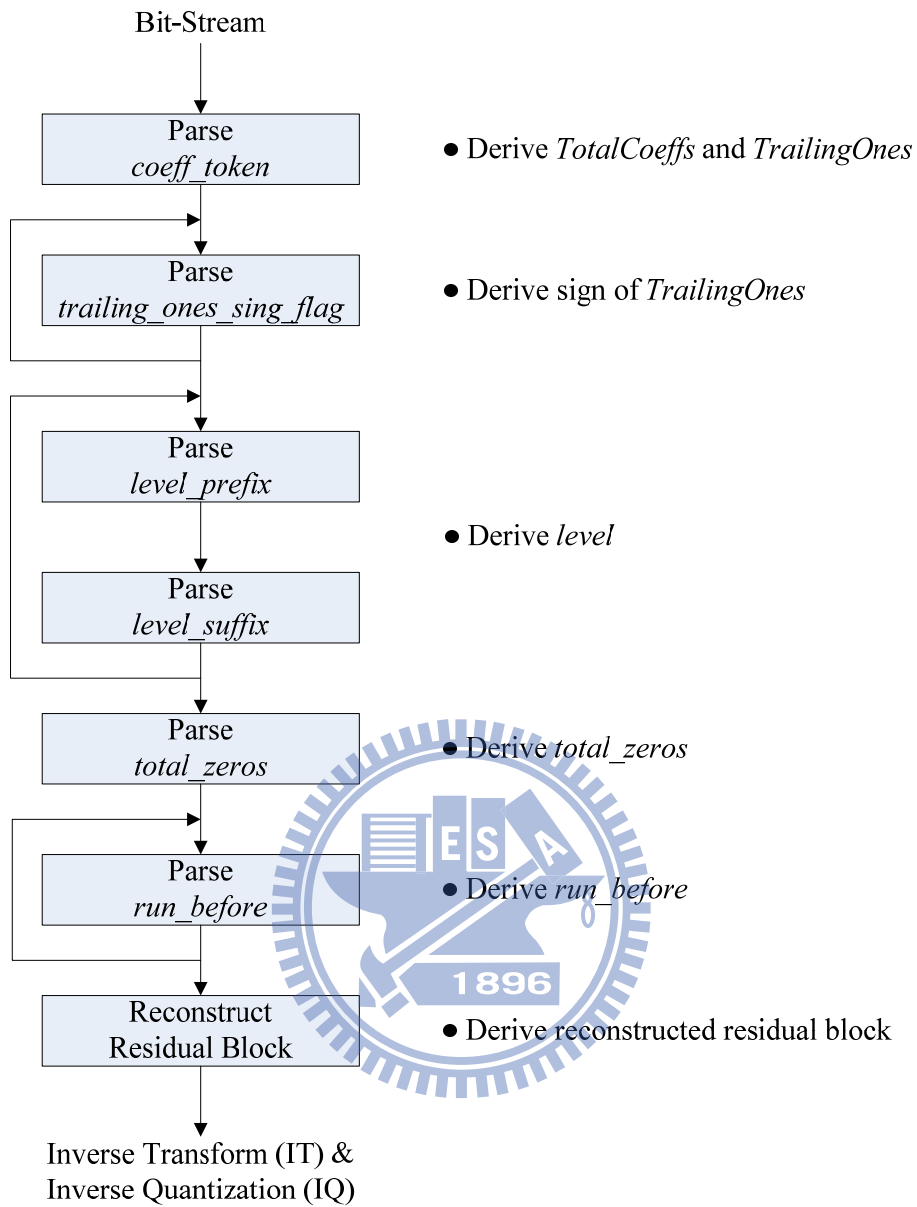


Figure 1. CAVLC decoding flow.

4 x 4 residual block

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

Reordered block: 0, 3, 0, 1, -1, -1, 0, 1, 0...

Encoded CAVLC bitstream: 000010001110010111101101

Figure 2. Transmitted bitstream for a 4 x 4 residual block.

TABLE 1. CAVLC DECODING PROCEDURE FOR THE 4 X 4 RESIDUAL BLOCK DEPICTED IN FIG. 2

Bitstream: 000010001110010111101101			
Syntax Element	Codeword	Value	Output Array
<i>coeff_token</i>	100	<i>TotalCoeffs = 5, TrailingOnes = 3</i>	N/A
<i>TrailingOne sign</i>	0	+	<u>1</u>
<i>TrailingOne sign</i>	1	-	<u>-1</u> , 1
<i>TrailingOne sign</i>	1	-	<u>-1</u> , -1, 1
<i>level</i>	1	+1	<u>1</u> , -1, -1, 1
<i>level</i>	0010	+3	<u>3</u> , 1, -1, -1, 1
<i>total_zeros</i>	111	3	3, 1, -1, -1, 1
<i>run_before</i>	10	1	3, 1, -1, -1, <u>0</u> , 1
<i>run_before</i>	1	0	3, 1, -1, -1, 0, 1
<i>run_before</i>	1	0	3, 1, -1, -1, 0, 1
<i>run_before</i>	01	1	3, <u>0</u> , 1, -1, -1, 0, 1
Reconstructed block: 0, 3, 0, 1, -1, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0			

2.2 Design Challenges and Related Works

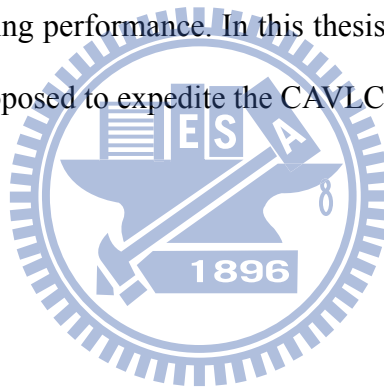
In hardware implementation, the VLC decoding can be realized as a finite state machine in essence. One bit or several bits of bitstream are scanned in each clock

cycle. According to the chosen VLC table, if the bit string matches a codeword, the corresponding value is returned. Otherwise, more bits will be scanned in the next cycle. Since the bitstream boundary between successive codewords is unknown until the codeword length of the former one is detected, the decoding procedure is inherently sequential and thus the throughput of CAVLC decoder is therefore hard to be elevated.

Intuitively, multi-symbol decoding is an effective way to raise throughput, especially for *trailing_ones_sing_flag*, *level*, and *run_before* parsing stages which are critical loops in the CAVLC decoding procedure. However, the main obstacle to parallel decoding is how to break the recursive dependencies between codewords. In *trailing_ones_sing_flag* parsing stage, since the number of *TrailingOnes* is already derived in *coeff_token* parsing stage, [4] and [5] implemented the parsing procedure in a single cycle. In *level* parsing stage, two *level* decoders are cascaded to produce two *level* symbols in one cycle [6]. However, it induces a huge critical path delay. In *run_before* parsing stage, since the codewords of VLC table used for *run_before* is much less and shorter than others, the data dependency obstacle is much easier to be overcome, and thus several efficient multi-*run_before* decoding architectures had proposed to boost the throughput of CAVLC decoder. When *run_before* is equal to 0, the corresponding codeword is composed of “1” bits. Therefore by counting the bit length of the series of “1” bits of input bitstream, multiple *run_before* symbols valued 0 can be parsed in one cycle [6]. This method is effective in the high bit-rate coding but inefficient in the low bit-rate coding where the residual blocks are very sparse. Since the sub VLC tables of *run_before* are separated by *zerosLeft*, unless *zerosLeft* is larger than 6, the *zerosLeft* for choosing the next *run_before* look-up table is predictable. By utilizing this character, Yu *et al.* [7] proposed a combined look-up table for decoding successive two *run_before* symbols at the same time. At the

expense of significant hardware cost raise, Wen *et al.* [8] adopted a bit-position VLC decoding approach that all *run_before* symbols are decoded using less than 3 cycles in one block to achieve high throughput. Lee *et al.* [9] presented a multi-symbol decoder that can decode three *run_before* symbols in one cycle. Furthermore, a pattern-search method had been reported in [10]. In this method, a block can be reconstructed directly without performing CAVLC decoding procedure if a pattern is matched in a pre-established look-up table.

For the two critical loops, *level* parsing process and *run_before* parsing process, which mainly affect the overall decoding performance, a lot of techniques have been proposed to speed up *run_before* parsing process, whereas there are few effective ways to improve *level* parsing performance. In this thesis, a highly efficient two-level decoding architecture is proposed to expedite the CAVLC decoding speed.



Chapter 3 OVERVIEW OF CABAC

For a data symbol with probability of occurrence P to be encoded, the theoretical optimum number of bits is $\log_2(1 / P)$. It is usually a fraction instead of an integer. As a result, in essence, entropy coding based on integral number of bits long codewords can not achieve optimal data compression. As a practical alternative, arithmetic coding provides a technique that can encode a sequence of data symbols into a single fractional number and thus can more closely approach the theoretical optima.

3.1 Arithmetic Coding

The arithmetic coding algorithm is a recursive subdivision of an interval based on the probability of occurrence of encoded symbols. In the encoding procedure, first, the range (0.0, 1.0) is subdivided into subranges depending on the probability of occurrence of each symbol as Fig. 3 shows. Then, whenever a symbol is encoded, the new range is set to the corresponding subrange. Finally, the sequence of data symbols can be represented by any fractional number in the final range. An example for encoding the sequence (C, B, C, E) is presented in Fig. 4. After the first symbol is encoded, the new range is (0.3, 0.7), and the next new range is (0.34, 0.42). Progressively, the initial range becomes smaller. At the end of sequence of data symbols, a number 0.394 which lies within the final range (0.3928, 0.396) is outputted.

Symbol	Probability	$\log_2(1/P)$	Subrange
A	0.1	3.32	(0.0, 0.1)
B	0.2	2.32	(0.1, 0.3)
C	0.4	1.32	(0.3, 0.7)
D	0.2	2.32	(0.7, 0.9)
E	0.1	3.32	(0.9, 1.0)

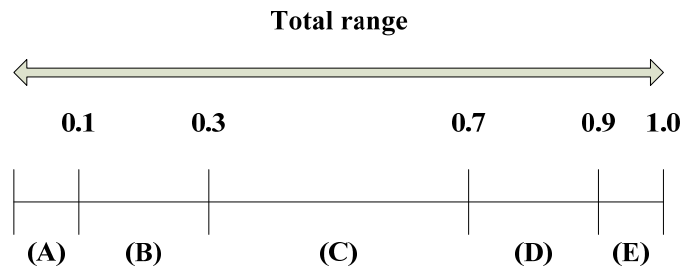


Figure 3. Example for interval subdivision.

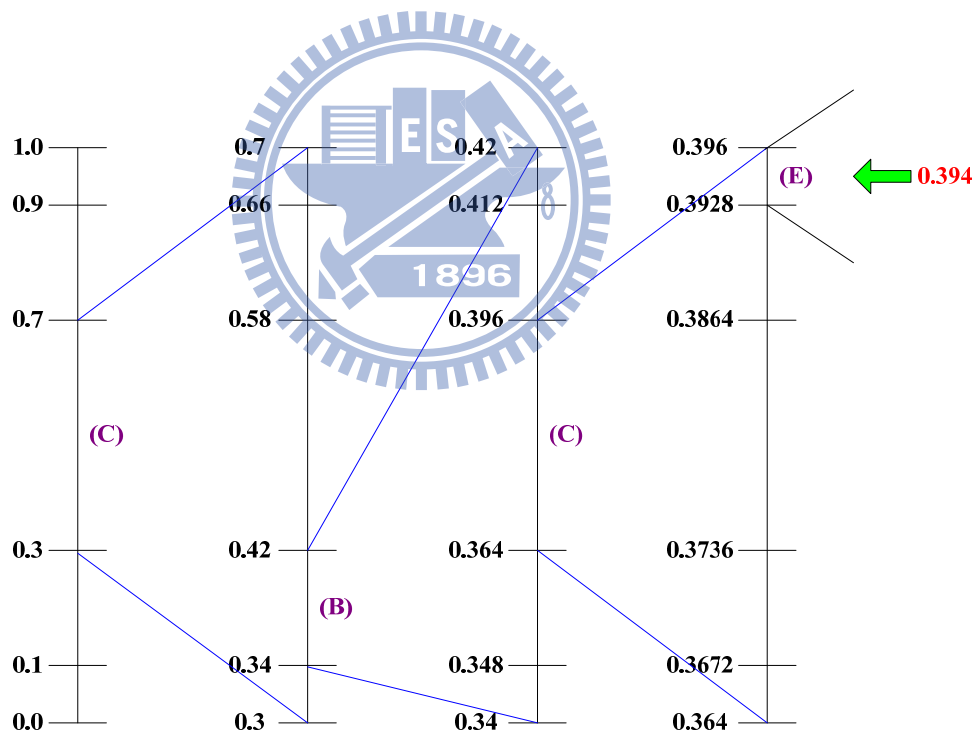


Figure 4. Recursive interval subdivision for the sequence (C, B, C, E).

In the decoding procedure, each symbol is decoded depending on the subrange where the input number falls. Then, the new range is updated to this subrange. Table 2 shows an example for decoding a fractional number 0.394 encoded by the encoding

procedure mentioned above. When decoding the first symbol, because 0.394 falls within the subrange (0.3, 0.7), it is decoded as (C). Then, range is set to the subrange which belongs to (C). The next symbol is decoded as (B) since 0.394 lies within the subrange (0.34, 0.42), and so on. The decoding does not halt until the entire sequence of data symbols (C, B, C, E) is decoded.

TABLE 2. DECODING PROCEDURE FOR INPUT NUMBER 0.394

Decoding Procedure	Range	Subrange	Decoded Symbol
1) Set the initial range	(0.0, 1.0)		
2) Find the subrange where the number falls and decode the symbol		(0.3, 0.7)	(C)
3) Set the new range	(0.3, 0.7)		
4) Find the subrange where the number falls and decode the symbol		(0.34, 0.42)	(B)
5) Set the new range	(0.34, 0.42)		
6) Find the subrange where the number falls and decode the symbol		(0.364, 0.396)	(C)
7) Set the new range	(0.364, 0.396)		
8) Find the subrange where the number falls and decode the symbol		(0.3928, 0.396)	(E)

3.2 Context-based Adaptive Binary Arithmetic Coding

In spite of the fact that the algorithm of arithmetic coding is simple in definition, the hardware and software implementations suffer from its high computational complexity. The limited throughput (symbols/second) is generally considered as its main disadvantage. To solve this problem while maintaining the compression efficiency, CABAC introduces an adaptive binary arithmetic coding technique combined with well-designed context models. Furthermore, the interval is subdivided by using addition and look-up take to avoid multiplication operation, and the

probabilities updating is simplified by look-up table.

Fig. 5 shows the block diagram of CABAC encoding process. The encoding process consists of three steps: binarization, context modeling, and binary arithmetic coding [2]. In the first step, a syntax element is transferred from non-binary value into a series of binary bins, called a bin string. For each bin to be encoded, two coding modes are candidates. In the regular mode, a context model representing probability model is first selected according to previous encoded syntax elements. Then, based on the context model, the bin value is encoded by the regular coding engine, and context model updating follows. In the bypass mode, a bypass coding engine without the usage of context model is executed to speed up the encoding process. The three functional blocks are discussed in more detail in the following.

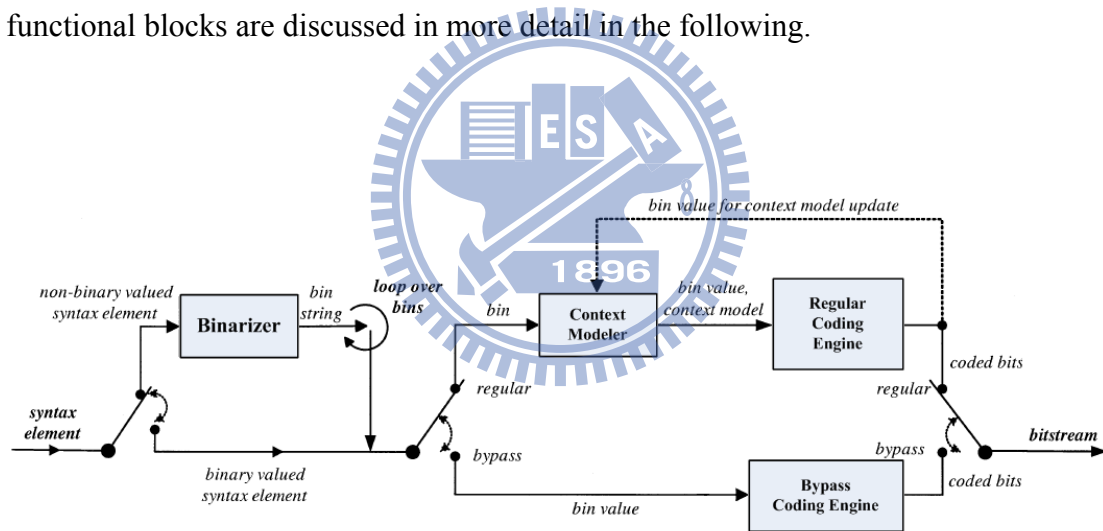


Figure 5. CABAC encoder block diagram.

3.2.1 Binarization

The binarization design in CABAC depends on a few code trees that provide a simple computation to derive codewords. There are four types of binarization process specified in CABAC: the unary (U) binarization process, the truncated unary (TU) binarization process, the fixed-length (FL) binarization process, and the concatenated

unary/ k-th order Exp-Golomb (UEGk) binarization process. However, there is an exception. Instead of computing by means of a structured coding scheme, look-up tables are used for mapping macroblock types and submacroblock types into binary sequences.

Table 3 shows the bin strings of U binarization. For each unsigned integer valued syntax element x , the bin string consists of x “1” bits followed by a terminating “0” bit.

TABLE 3. UNARY BINARIZATION

Value of syntax element (x)	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
...
Bin index	0	1	2	3	4	5 ...

The bin strings of TU binarization are shown in Table 4. A number $cMax$ is defined for mapping x with $[0, cMax]$. For $x < cMax$ the bin strings are the same as U codes, whereas $x = cMax$ the bin string is given by a bin string of length $cMax$ with “1” bits only.

TABLE 4. TRUNCATED UNARY BINARIZATION

Value of syntax element (x)	Bin string ($cMax = 7$)					
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	
6	1	1	1	1	1	1	0
7	1	1	1	1	1	1	1
Bin index	0	1	2	3	4	5	6

As shown in Table 5, the bin strings of FL binarization are given by fixedLength-bit binary representations, where $\text{fixedLength} = \text{Ceil}(\text{Log}_2(cMax + 1))$. The FL binarization process is mainly applied to the syntax elements which are nearly uniform distribution.

TABLE 5. FIXED-LENGTH BINARIZATION

Value of syntax element (x)	Bin string ($cMax = 7$)		
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
Bin index	0	1	2

The UEGk binarization process is applied to absolute values of motion vector differences (MVD) and absolute values of transform coefficient levels (ABS_LEVEL). The UEGk bin string consists of a prefix and a suffix code word. The prefix bit string is constructed by TU binarization process with $cMax = \text{Min}(uCoff, \text{Abs}(x))$, where

$ucoff$ is the cutoff value which also represents the maximum length of the prefix bit string. After the prefix part is obtained, if $Abs(x)$ is larger than or equal to the cutoff value, the EGk binarization process is invoked to derive the suffix part. The first part of EGk code is formed with a unary code with $l(y) = \text{Floor}(\log_2(y / 2^k + 1))$. The second part is constructed as the binary representation of $y + 2^k(1 - 2^{l(y)})$ with $(k + l(y))$ bits. The pseudo code of computational procedure is depicted in Fig. 6. Table 6 shows the bin strings for MVD valued from 0 to 13, where the prefix parts are in gray shadow.

```

if( Abs( x ) >= uCoff ) {
  y = Abs( x ) - uCoff
  while( 1 ) {
    //unary first part of EGk
    if( y >= (1 << k) ) {
      put( 1 )
      y = y - (1 << k)
      k++
    }
    else {
      put( 0 ) //terminating "0" of first part
      while( k-- ) //binary second part of EGk
      put( (y >> k) & 1 )
      break
    }
  }
}

```

Figure 6. Pseudo code for k-th order Exp-Golomb code construction.

TABLE 6. UEG3 BINARIZATION FOR ABSOLUTE VALUES OF MOTION VECTOR DIFFERENCES

MVD	Bin string ($uCoff = 9$)
-----	----------------------------

	Prefix (TU code)								Suffix (EG3 code)					
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								
6	1	1	1	1	1	1	0							
7	1	1	1	1	1	1	1	0						
8	1	1	1	1	1	1	1	1	0					
9	1	1	1	1	1	1	1	1	1	0	0	0	0	
10	1	1	1	1	1	1	1	1	1	0	0	0	1	
11	1	1	1	1	1	1	1	1	1	0	0	1	0	
12	1	1	1	1	1	1	1	1	1	0	0	1	1	
13	1	1	1	1	1	1	1	1	1	0	1	0	0	
...	
Bin index	0	1	2	3	4	5	6	7	8	9	10	11	12	...

3.2.2 Context Modeling

The probability models supplying for binary arithmetic coding is an important part since it dominates the overall coding efficiency. Consequently, the context model has to be selected by taking into account conditional probability estimation and keep updated during encoding. In CABAC, to reduce the complexity requirement, only the neighbors of current syntax element are involved in context model selection such that only a few choices are left.

TABLE 7. SYNTAX ELEMENTS AND CORRESPONDING CONTEXT INDICES

Syntax Element	Slice Type		
	I/SI	P/SP	B
mb_skip_flag		11–13	24–26
mb_field_decoding_flag	70–72	70–72	70–72
end_of_slice_flag	276	276	276

mb_type	0/3–10	14–20	27–35
transform_size_8x8_flag	399–401	399–401	399–401
coded_block_pattern	73–84	73–84	73–84
mb_qp_delta	60–63	60–63	60–63
prev_intra4x4_pred_mode_flag	68	68	68
rem_intra4x4_pred_mode	69	69	69
prev_intra8x8_pred_mode_flag	68	68	68
rem_intra8x8_pred_mode	69	69	69
intra_chroma_pred_mode	64–67	64–67	64–67
ref_idx		54–59	54–59
mvd (horizontal)		40–46	40–46
mvd (vertical)		47–53	47–53
sub_mb_type		21–23	36–39
coded_block_flag	85–104	85–104	85–104
significant_coeff_flag	105–165, 277–337	105–165, 277–337	105–165, 277–337
last_significant_coeff_flag	166–226, 338–398	166–226, 338–398	166–226, 338–398
coeff_abs_level_minus1	227–275	227–275	227–275
significant_coeff_flag (8x8)	402–416, 436–450	402–416, 436–450	402–416, 436–450
last_significant_coeff_flag (8x8)	417–425, 451–459	417–425, 451–459	417–425, 451–459
coeff_abs_level_minus1 (8x8)	426–435	426–435	426–435

All context models are listed in Table 7. Each context model, which contains a 6-bit probability state and the value of most probable symbol, is identified by a context index (*ctxIdx*). The calculation of *ctxIdx* is defined as

$$ctxIdx = ctxIdxBase + ctxCat + ctxIdxInc \quad (1)$$

where *ctxIdxBase* denotes the base context index, which is defined as the lower value of the range contained in Table 7, *ctxCat* represents context category, which is only valid for syntax elements of residual type and is given in Table 8, and *ctxIdxInc* denotes the context index increment, which is derived based on bin index (*binIdx*),

previously encoded bins, or neighboring syntax elements to the left and on top of the current syntax element, as illustrated in Fig. 7.

TABLE 8. CONTEXT CATEGORY DEPENDING ON SYNTAX ELEMENTS AND BLOCK TYPES

Syntax element	Context category (<i>ctxCat</i>)					
	Luma-16x16	Luma-16x16	Luma-4x4	Chroma	Chroma	Luma-8x8
	DC	AC		DC	AC	
<code>coded_block_flag</code>	0	4	8	12	16	0
<code>significant_coeff_flag</code>	0	15	29	44	47	0
<code>last_significant_coeff_flag</code>	0	15	29	44	47	0
<code>coeff_abs_level_minus1</code>	0	10	20	30	39	0

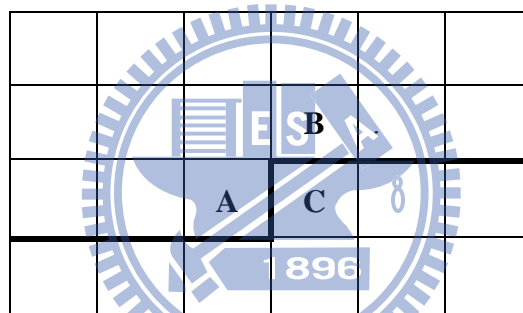


Figure 7. Neighboring syntax elements involved in context model selection of current syntax element.

3.2.3 Adaptive Binary Arithmetic Coding

Binary arithmetic coding is based on the principle of progressive interval subdivision. In terms of symbols to be encoded, only most probable symbol and least probable symbol (MPS and LPS) with probabilities of occurrence P_{MPS} and P_{LPS} are specified. Based on this setting, the given interval represented by a lower bound (L) and an interval range (R) is subdivided into R_{MPS} and R_{LPS} as follows:

$$\begin{aligned}
 R_{LPS} &= R \times P_{LPS} \\
 R_{MPS} &= R - R_{LPS}
 \end{aligned} \tag{2}$$

However, the computational requirement of multiplication operations becomes the bottleneck that limits the overall throughput. To solve this problem, a novel multiplication-free solution with negligible performance degradation is developed in CABAC.

Motivated by introducing some approximations of the range R or of the probability P_{LPS} in substitution for their actual values, the basic idea of the new multiplication-free binary arithmetic coding scheme for H.264/AVC relies on the assumption that the estimated probabilities of each context model can be represented by a sufficiently limited set of representative values [3]. Total 128 probability states are effectively used for representing the approximate probability estimation of each context model. Each probability state is composed of a 6-bit state index ($stateIdx$) indicating the LPS probability and a 1-bit value that represent the MPS value ($valMPS$). The numbering of state index is guided by the principle that with state index equaling to 0 corresponds an LPS probability value of 0.5, the higher the number of state index, the lower LPS probability value is assigned. Whenever the encoding procedure of each symbol is completed, the context model updating process is executed to keep context models “up to date”. The determination of probability updating is illustrated in Fig. 8. In practical implementation, the transition of probability states can be realized by a table-based transition process. This continuous update makes the binary arithmetic coding engine adaptive.

$$P_{new} = \begin{cases} \text{Max}(\alpha \cdot P_{old}, P_{62}), & \text{if current bin is MPS} \\ \alpha \cdot P_{old} + (1 - \alpha), & \text{if current bin is LPS} \end{cases}$$

$$\alpha = \left(\frac{0.01875}{0.5} \right)^{\frac{1}{63}} \cong 0.095$$

Figure 8. Probability transition rule.

Fig. 9 shows the flow diagram of binary arithmetic encoding process. Two coding modes are specified in CABAC, one is regular coding mode, where probability estimation is utilized, and another one called bypass coding mode is used to encode symbols with approximately uniform probability distribution.

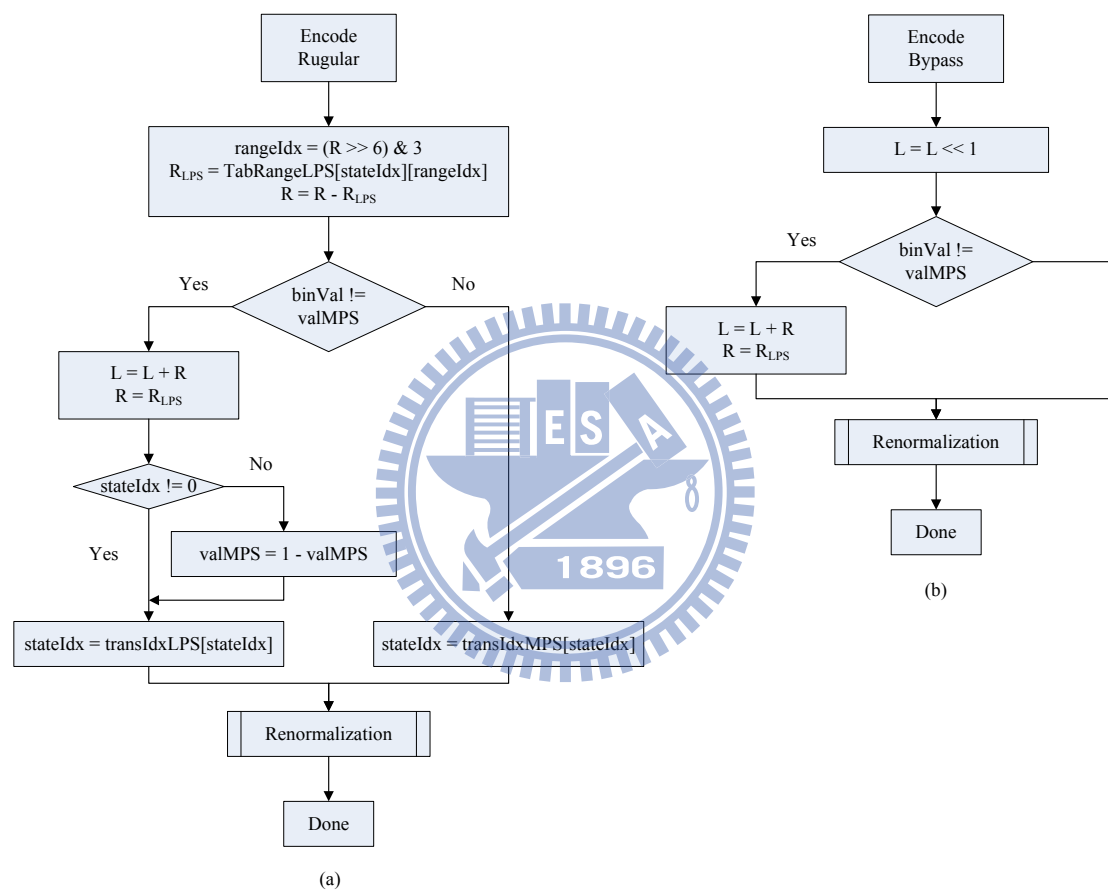


Figure 9. Flow diagram of binary arithmetic encoding process. (a) Regular coding mode. (b) Bypass coding mode.

Fig. 9(a) illustrates the regular coding mode. In the first step, with a table which contains 64×4 pre-computed LPS subranges, the interval range is subdivided depending on the state index and range without multiplication operation. Then, according to the given bin value (*binVal*), the corresponding process is performed.

Finally, since the interval range has to stay within $[2^8, 2^9]$ to keep a fixed precision, a renormalization process is necessary if the updated interval range R is smaller than $0x100$. Fig. 10 shows the flow diagram of renormalization process. The output bits are recursively generated during the renormalization. If the interval range is in the bottom half, $PutBit(0)$ is performed; else if the interval range is in the top half, $PutBit(1)$ is performed; otherwise bitsOutstanding (BO) is increased by 1.

With regard to bypass coding mode, the probability distribution of symbol to be encoded is nearly uniform. That means $R_{LPS} = R_{MPS} = R/2$. Consequently, the usage of context model is not required and the subdivision operation can be simplified to accelerate the encoding speed. Furthermore, only one-loop renormalization process using double decision thresholds without doubling R and L is performed in the final step. The flow diagram of bypass coding mode is depicted in Fig. 9(b).

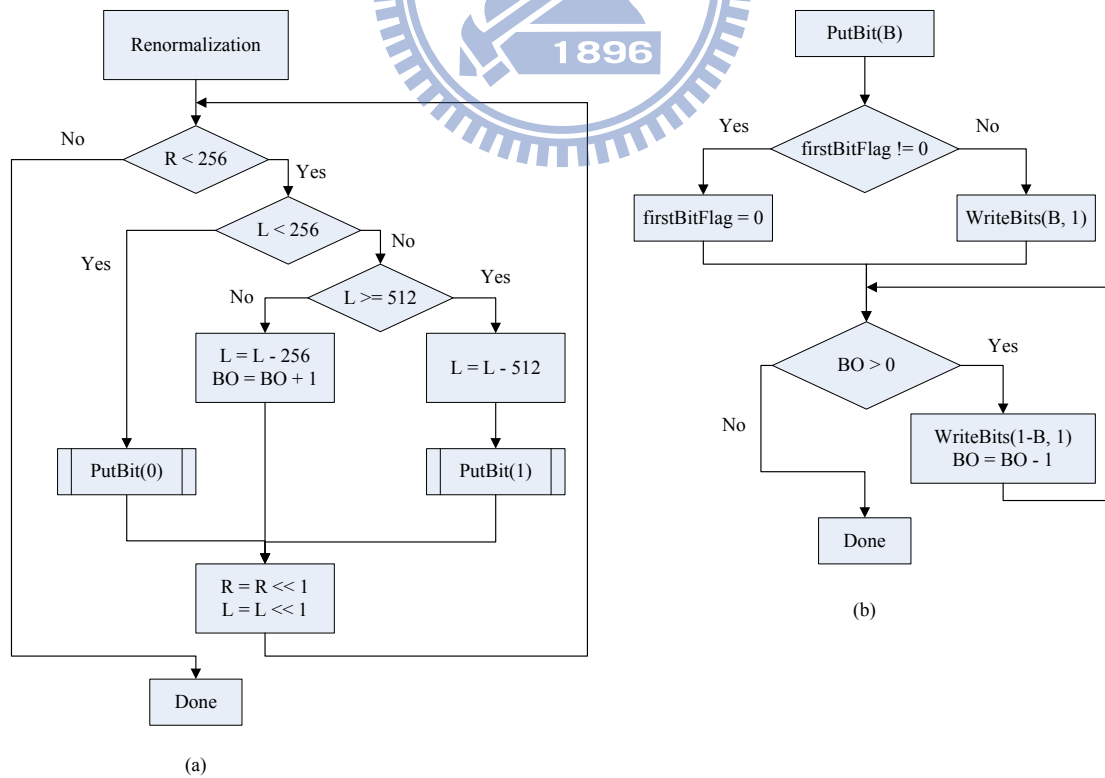
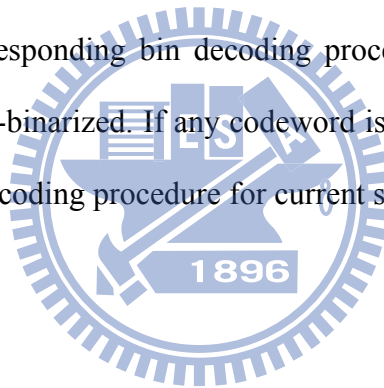


Figure 10. Flowchart of (a) renormalization process and (b) PutBit(B).

3.3 CABAC Decoding Algorithm Overview

In CABAC, every syntax element (SE) is composed of a series of bins. Given the bitstream, combined with syntax element parsing, the object of CABAC decoder is to transfer the decoded bin string into actual value and return it. Fig. 11 depicts the generic CABAC parsing process. Prior to decoding a new slice, an initialization process is performed that all context models are initialized depending on the slice type and quantization parameter, moreover, the interval range and coding offset are reset to 0x1FE and first 9 bits of the bitstream, respectively. In the parsing flow, each syntax element is parsed sequentially. After the type of syntax element is decided, depending on the bin index, the corresponding bin decoding process is executed. Finally, the constructed bin string is de-binarized. If any codeword is matched, the corresponding value is returned and the decoding procedure for current syntax element is complete.



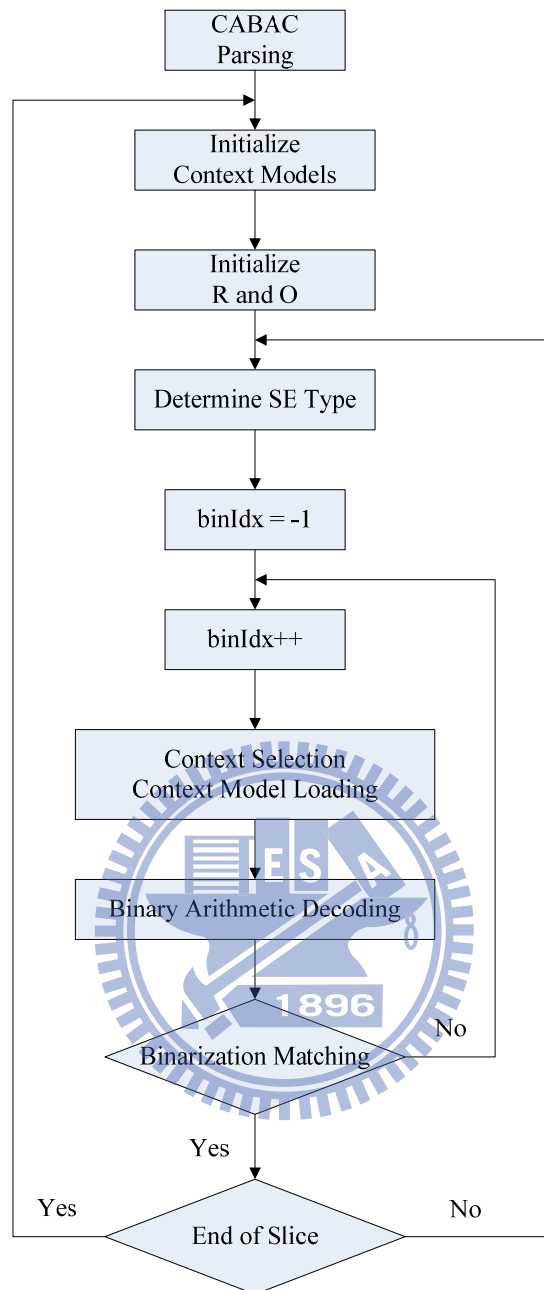


Figure 11. CABAC parsing process.

In the hardware realization, the bin decoding process consists of four elementary steps: context selection (CS), context model loading (CL), binary arithmetic decoding (BAD), and binarization matching (BM). In the first step, context index which acts as the context model address is calculated. After the address is obtained, a context model (CM) loaded from CM memory is passed to the BAD stage. In BAD stage, a bin is

decoded to be MPS or LPS according to a probability model provided by the CM. Afterward, the constructed bin string is de-binarized in the final stage to decide whether the decoding process of current SE is finished or not, and the context model update (CU) process takes place at the same time.

To read the specific context model from the context model memory, the memory address must be calculated first. Generally, the memory address of each context model is the same as its corresponding context index. However, the organization of context models in H.264/AVC is clearly not the most economical. Therefore, reorganization is allowed to achieve better performance as designer's wish.

In the binary arithmetic decoding procedure, most symbols are decoded by the regular bin decision process depending on the location of coding offset. Fig. 12(a) shows the flowchart of regular decoding process. In the first step, according to the state index provided by context model and current interval range, LPS subrange is selected from a look-up table and MPS subrange is calculated as $R_{MPS} = R - R_{LPS}$. Then, by comparing the coding offset (O) with the MPS subrange (R_{MPS}), if the coding offset falls within the LPS subrange, the bin is identified as LPS. Otherwise, if the coding offset falls within the MPS subrange, the bin is identified as MPS. In the meanwhile, R and O are assigned to the corresponding subinterval, and the probability state is transferred in the end.

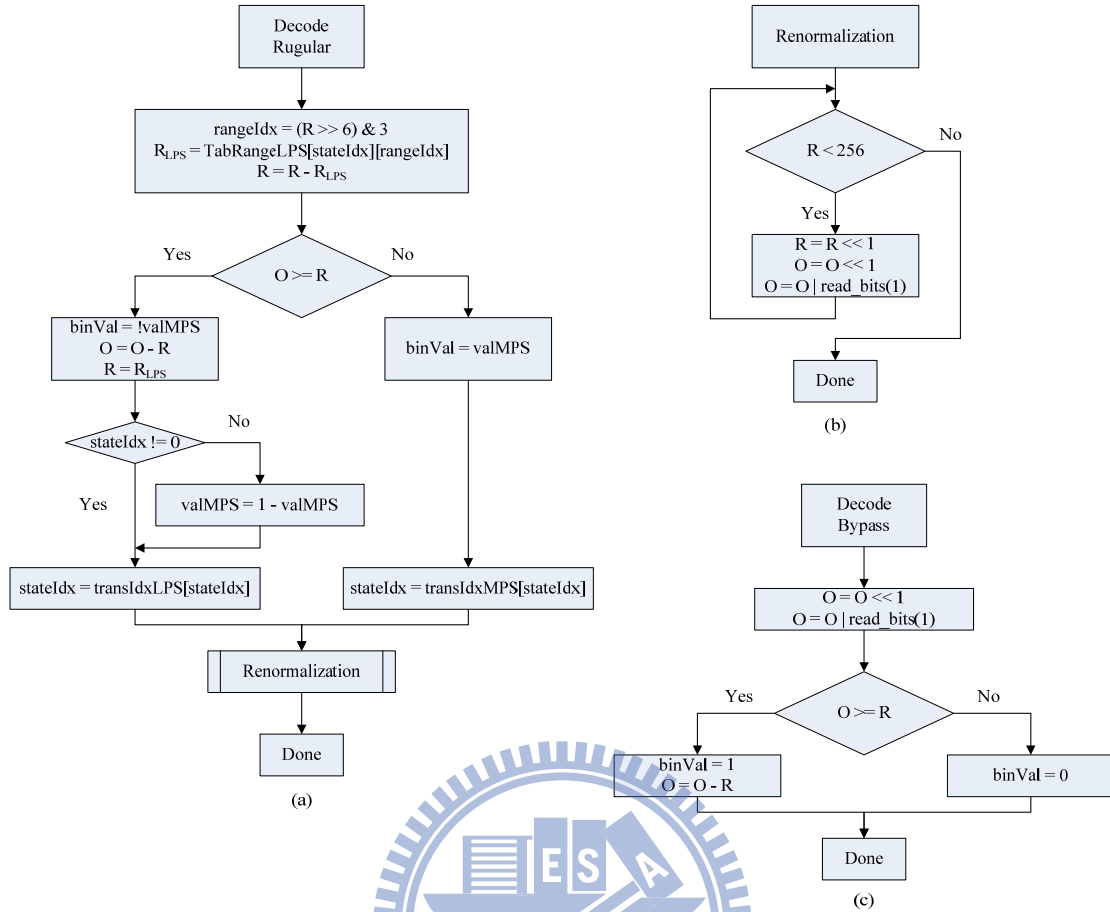


Figure 12. Flow diagram of (a) regular bin decision process, (b) renormalization process, and (c) bypass bin decision process.

A renormalization operation is required whenever the interval range (R) is out of its legal range ($R < 0x100$). Fig. 12(b) depicts the flowchart of renormalization. Recursively, the left-shift of R and O does not halt until R is larger than or equal to $0x100$. During the renormalization procedure, the input bits coming from bitstream are appended to coding offset.

Besides, the other symbols with approximately uniform probability distribution are decoded by bypass bin decision process. The flowchart of bypass decoding process can be seen in Fig. 12(c).

3.4 Design Challenges and Related Works

In hardware implementation, to achieve high throughput, pipelined architecture and parallel architecture are considered helpful methods generally. Since every bin is decoded by the same chain of operations (CS→CL→BAD→BM and CU), the decoding performance can be elevated by exploiting the pipelining scheme presented in [11]. Fig. 13 shows a 4-stage pipelining CABAC decoder design. However, the boost of throughput is limited by the pipeline stalling caused by data hazards. Take significance map (significant_coeff_flag and last_significant_coeff_flag) which occupies the major portion of syntax elements in slice data for example, as shown in Fig. 14, the choice of the bin right after significant_coeff_flag and corresponding context model depends on the current bin value since the bin may be significant_coeff_flag or last_significant_coeff_flag. As a result, two cycles are unavoidable to resolve this data hazards.

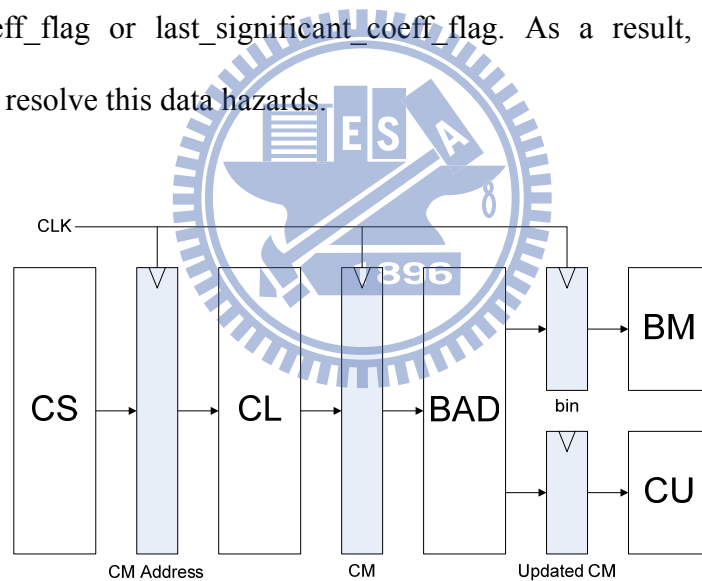


Figure 13. Pipelining scheme of CABAC decoding.

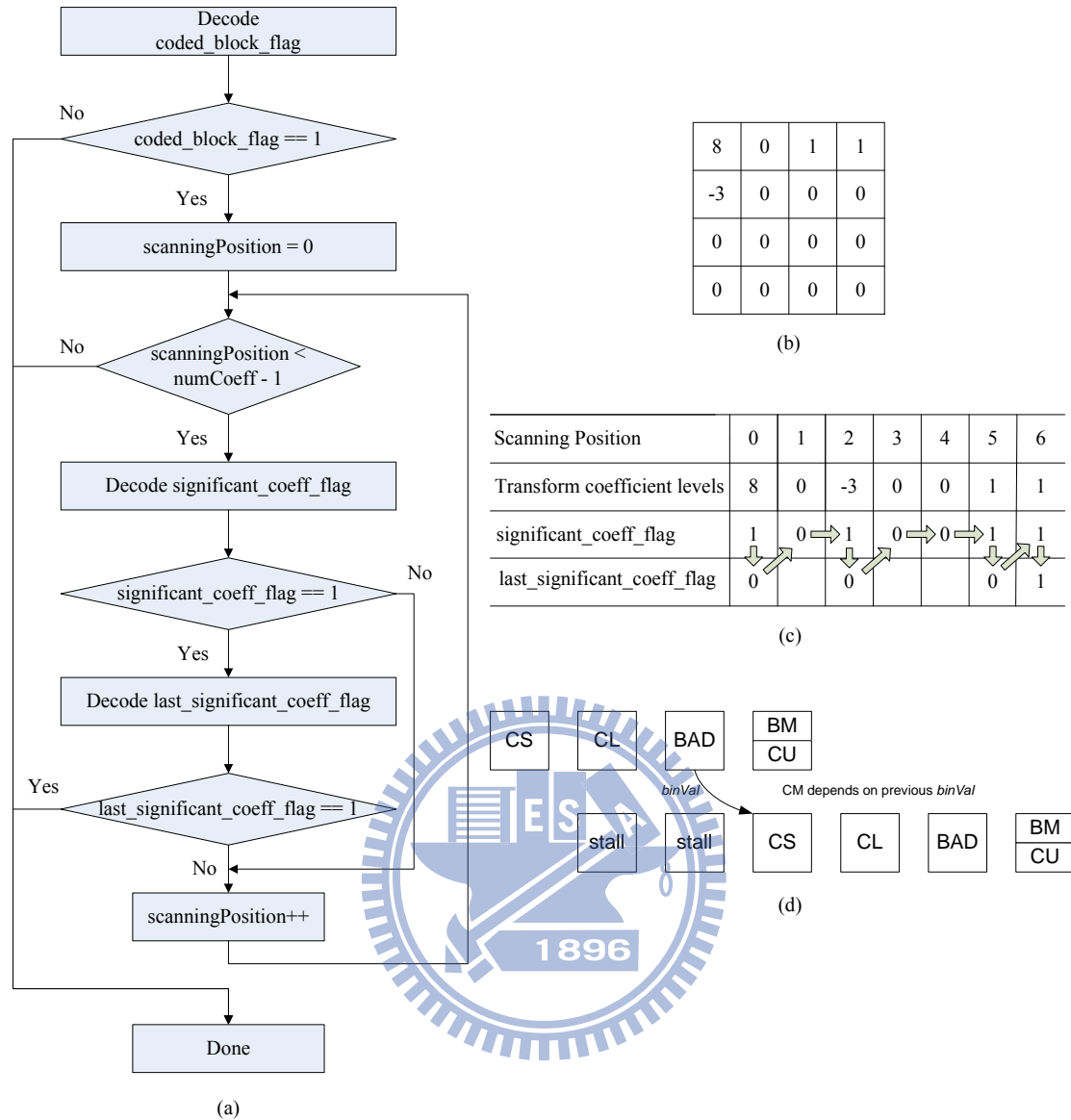


Figure 14. Data hazard caused by significance map. (a) 4x4 residual block. (b) Flow diagram of the CABAC decoding scheme for significance map. (c) Example for decoding the significance map. (d) Illustration of cycle stall of CABAC decoding.

To relieve the performance degradation originated in syntax element switching overhead, a prediction-based pipelined architecture was proposed in [12], where the correlation between successive SEs are exploited to achieve higher prediction accuracy in comparison to the prediction that just predicts current symbol to be MPS. Furthermore, multi-symbol decoding architecture design is also an effective way to

speed up decoding procedure. A parallel decoding method was proposed to enhance decoding performance by predicting that the current symbol is MPS [16]. The architecture in [13] employed a branch selection two-symbol parallel decoding technique to resolve data dependency problem, and can process two bins within one cycle for general cases, but suffers from high area cost. Chen *et al.* [14] proposed a fully hardwired CABAC decoder that is capable of decoding at most two bins in one cycle for certain syntax elements: `coeff_abs_level_minus1`, `significant_coeff_flag`, `last_significant_coeff_flag`, and `mvd`.

However, in some works such as [15], [16], their architecture only focuses on bin decoding process, while leaving SE parsing to another processor. Although the separation of SE parsing and decoding makes the implementation of CABAC decoding much simpler, it results in that the actual throughput can not reach its theoretical maximum, since whenever SE switching takes place, the context model has to be reloaded.

A fully hardwired CABAC decoder design which combines SE parsing with decoding is proposed in this thesis. The characteristics of SE parsing flow and bin distribution among SEs are analyzed to design the decoding architecture which not only can decode multiple bins in one cycle without stalls for most cases but also can keep low hardware cost by employing hybrid context model memory architecture. Moreover, with the efficient mathematical transform method for two-symbol binary arithmetic decoding (TSBAD) engine, the decoding speed can be further elevated.

Chapter 4 PROPOSED ENTROPY DECODER

Fig. 15 shows the system level architecture of proposed entropy decoder for H.264/AVC. It contains a CAVLC decoder, a CABAC decoder, a SE parser, a neighboring information fetcher, a bitstream fetcher, and a memory controller. According to the entropy coding mode, the SE parser chooses the corresponding decoder to decode SEs. When `entropy_coding_mode_flag` is equal to 0, SEs of residual blocks are decoded by using the CAVLC decoding scheme, and other SEs are decoded by using the VLC decoding scheme which is included in the CAVLC decoder. When `entropy_coding_mode_flag` is equal to 1, SEs lying at macroblock layer and below are decoded using the CABAC decoder, and other SEs belonging to slice layer and above are decoded by the VLC decoder.

In the entropy decoding procedure, the bitstream fetcher reads bitstream which is stored in external memory by the memory controller and transmits it to the CAVLC decoder and the CABAC decoder. The neighboring data involving in entropy decoding process, such as *total_coeff* and *mvd* used for calculating *nC* and *ctxIdxInc*, are stored in the upper macroblock information memory. Furthermore, if entropy coding mode is binary arithmetic coding, in the beginning of decoding each slice, all the CMs are reset to the initial values stored in ROM. Whenever a SE is decoded, if it is related to the remaining SE parsing flow, it will be buffered in the SE register. The detail of our proposed entropy decoder is presented in the following.

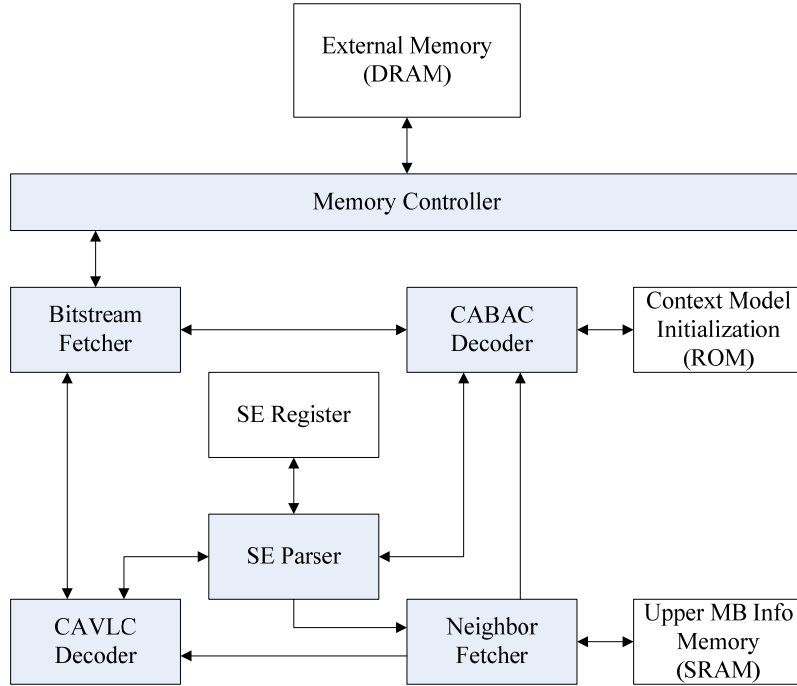


Figure 15. Framework of proposed entropy decoder.

4.1 Proposed CAVLC Decoder

It is apparent that the greatest obstacle to further boosting the throughput of CAVLC decoder originates in *level* parsing procedure which is based on arithmetic operations and accounts for a critical loop in the whole CAVLC decoding procedure. In terms of multi-*level* decoding, since the inter-codeword dependency and succession of arithmetic operations lead to an unavoidably long critical path, we can not gain throughput from cascading level decoders directly. Moreover, the inter-*level* dependency of *suffixLength* which can not be calculated until the value of current level is determined makes it unable to exploit pipeline structure. It seems both multi-symbol decoding and pipelining scheme are not workable for *level* decoding process.

Our destination is to find a method that can break the inter-*level* dependency and the inter-codeword dependency. If this goal is reached, we can make a breakthrough

and thus the CAVLC decoding performance can be further improved. Consequently, first of all, we investigate the characteristics of *level* decoding flow.

4.1.1 Analysis

Fig. 16 shows the flowchart of *level* decoding procedure defined in the H.264/AVC standard. The decoding procedure can be divided into two parts: the first part is bitstream scanning process and the second part is for computing the value of *level*. The bit string of each *level* is formed with *level_prefix* and *level_suffix* as

$$\begin{aligned}
 & \textit{level_bitString} \\
 & = [\textit{level_prefix}][\textit{level_suffix}] \\
 & = [0\dots01][\textit{level_suffix}]
 \end{aligned} \tag{3}$$

where *level_prefix* consists of a series of “0” bits followed by a terminating “1” bit. The value of *level_prefix* is constrained in the range 0 to 15 in general profiles. In the bitstream scanning process, after the value of *level_prefix* is determined by detecting the leading zeros in the bitstream, the parameter *levelSuffixSize* which represented the bit length of *level_suffix* is calculated as

$$\begin{aligned}
 & \text{if}(\textit{level_prefix} == 15) \\
 & \quad \textit{levelSuffixSize} = 12 \\
 & \text{else if}(\textit{level_prefix} == 14 \ \&\& \ \textit{suffixLength} == 0) \\
 & \quad \textit{levelSuffixSize} = 4 \\
 & \text{else} \\
 & \quad \textit{levelSuffixSize} = \textit{suffixLength}
 \end{aligned} \tag{4}$$

Based on the *levelSuffixSize*, bits belonging to *level_suffix* are scanned, and the initial value of *levelCode* is calculated as

$$\textit{levelCode} = (\textit{level_prefix} \ll \textit{suffixLength}) + \textit{level_suffix} \tag{5}$$

In the second part, *levelCode* is adjusted in case of special conditions. If *level_prefix* is equal to 15 and *suffixLength* is equal to 0, *levelCode* will be increased by 15, and if the number of *TrailingOnes* is less than 3, the first *levelCode* in the *level* decoding procedure will be increased by 2. Once the final value of *levelCode* is obtained, the value of *level* will be determined as: if *levelCode* is even, $level = (levelCode + 2) / 2$. Otherwise, $level = (-levelCode - 1) / 2$. Finally, since the absolute value of *level* tends to be larger in the *level* decoding procedure, to obtain high compression efficiency, adaptive probability model is used depending on previous decoded *level*. As a result, by examining the absolute value of decoded level, if it is larger than the thresholds listed in Table 9, *suffixLength* must be modified to a more suitable value since small *suffixLength* is fit for small *level*; large *suffixLength* is just the opposite.

The main barriers to exploit parallel decoding are inter-*level* dependency of *suffixLength* and the unknown demarcation between successive codewords. Although the codeword length can be derived in the first part of *level* decoding procedure as follows:

$$CodewordLength = level_prefix + 1 + levelSuffixSize \quad (6)$$

, the updated *suffixLength* which affect the *levelSuffixSize* of next *level* can not be obtained until the value of current *level* is determined. However, a modified *suffixLength* detector (MSD) algorithm was presented to advance the computation of *suffixLength* prior to the determination of the value of current *level* [4]. Fig. 17 depicts the MSD decoding procedure, the input signal of MSD is *level_prefix* instead of the value of *level*. From the current decoding information and the *level_prefix*, the *suffixLength* provided for next level decoding process can be calculated in the first part. With this efficient algorithm, the *level* decoding process can be realized as Fig.

18 shows. However, despite the fact that the MSD algorithm shortens the critical path delay of *level* decoding process, multi-level decoding based on cascaded level decoders still leads to an unavoidably long critical path, and thus remains unsuitable for implementation.

In our approach, to further expedite the throughput of CAVLC decoder, instead of straight cascading level decoders, we take advantage of MSD algorithm to exploit a highly performance two-level decoding architecture. In general case, the *levelSuffixSize* which indicates the codeword length of *level_suffix* is equal to *suffixLength*. Consequently, the start point of next level codeword in the bitstream can be decided as soon as the *level_prefix* decoding has finished. Moreover, the adjustment of *levelCode* in the second part is only applied to the first *level* of the residual block. It means that those two special conditional branches can be skipped in the second *level* decoding. Base on these two features, we propose a delay balanced two-level decoding (DBTLD) architecture that efficiently shortens the critical path in comparison to traditional design that cascades two *level* decoders directly.

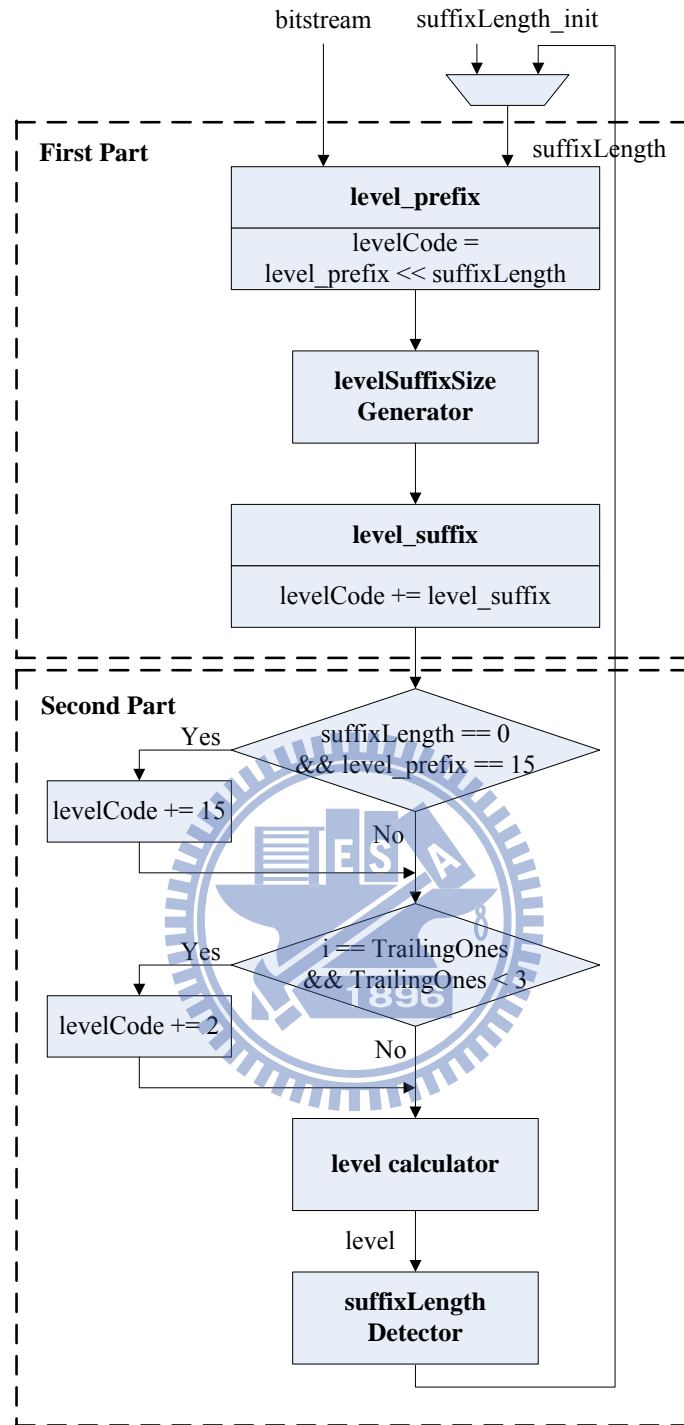
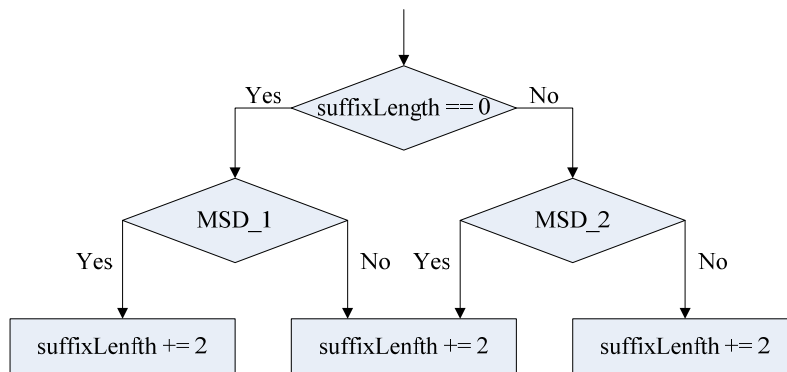


Figure 16. Original *level* decoding procedure defined in H.264/AVC standard.

TABLE 9. THRESHOLD VALUE FOR SUFFIXLENGTH TRANSITION

Current <i>suffixLength</i>	Threshold value to modify <i>suffixLength</i>
0	0
1	3

2	6
3	12
4	24
5	48
6	N/A



MSD_1: $(I == \text{TrailingOnes} \ \&\& \ \text{TrailingOnes} < 3 \ \&\& \ \text{level_prefix} > 3) \ || \ (\text{level_prefix} > 5)$

MSD_2: $(I == \text{TrailingOnes} \ \&\& \ \text{TrailingOnes} < 3 \ \&\& \ \text{suffixLength} == 1 \ \&\& \ \text{level_prefix} > 1) \ || \ (\text{level_prefix} > 2)$

Figure 17. MSD decoding procedure.



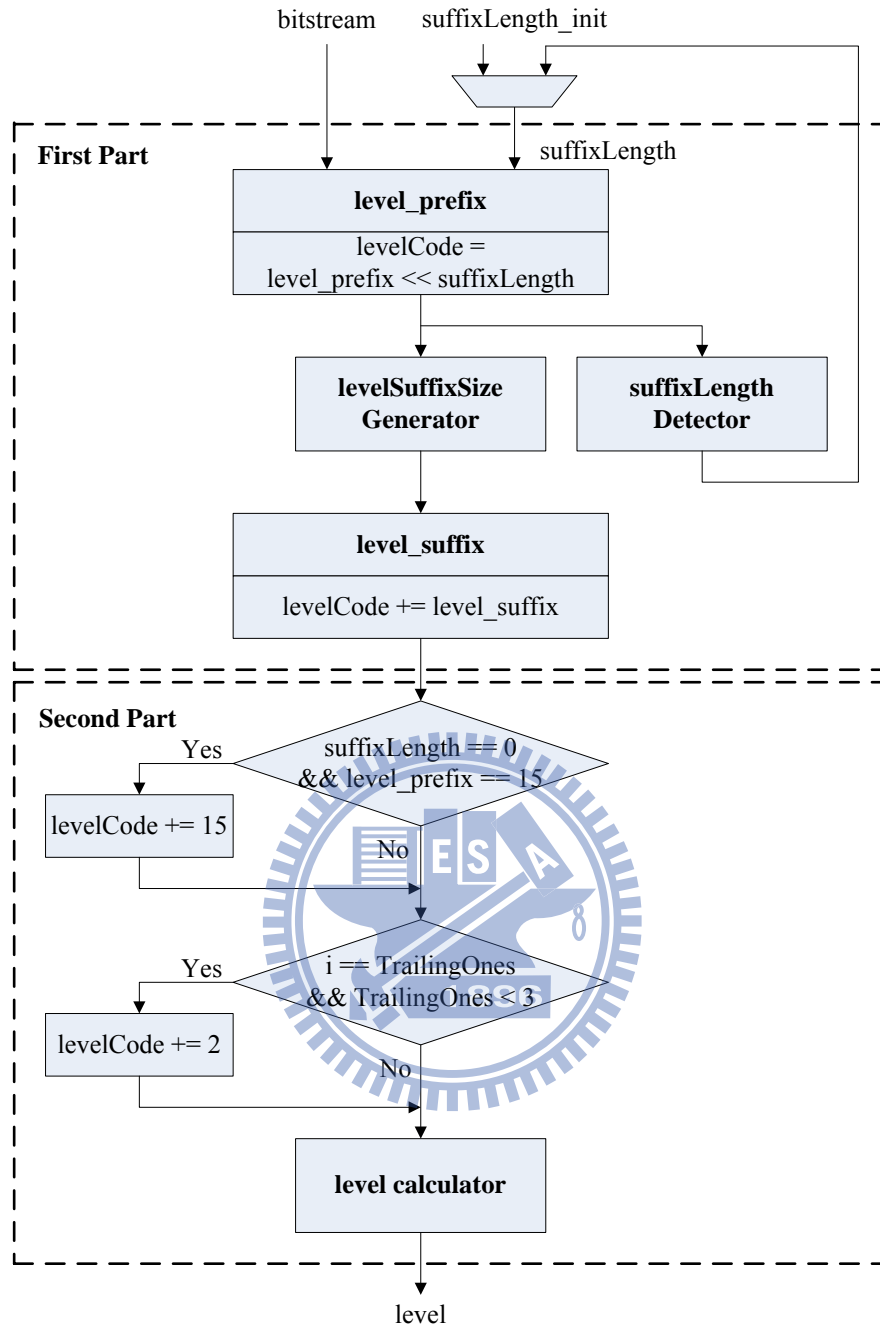


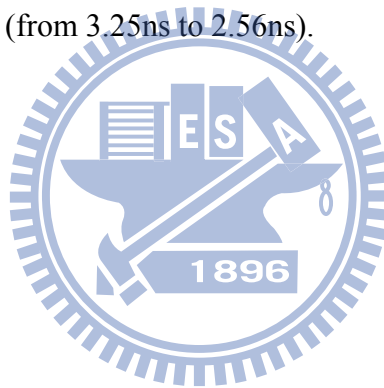
Figure 18. Modified *level* decoding procedure with MSD algorithm.

4.1.2 Proposed Delay Balanced Two-level Decoder Architecture

Fig. 19 shows the block diagram of proposed DBTLD architecture. The second *level* decoding process is designed for the general case that *levelSuffixSize* is equal to *suffixLength*. Since the codeword length of first *level* can be determined immediately after the *level_prefix* is decoded, and the examination process of *levelCode* increment

is unnecessary for the second *level* decoding process, a balanced structure can be obtained.

The first *level* decoding process is the same as Fig. 18 shows. For bitstream supplying for the second *level* decoding process, the input bitstream is shifted according to *suffixLength* and *level_prefix_1*. Afterward, instead of generating *levelSuffixSize_2*, the *level_suffix_2* is parsed directly by fetching the output of first *suffixLength_1* detector (SD₁) which is referred to the MSD algorithm. Finally, without checking the two special cases for increasing *levelCode_2*, the *level* mapping process is performed straight. Compared to the conventional approach of cascading two MSD algorithm based *level* decoders, the critical path delay of proposed DBTLD engine is improved by 21% (from 3.25ns to 2.56ns).



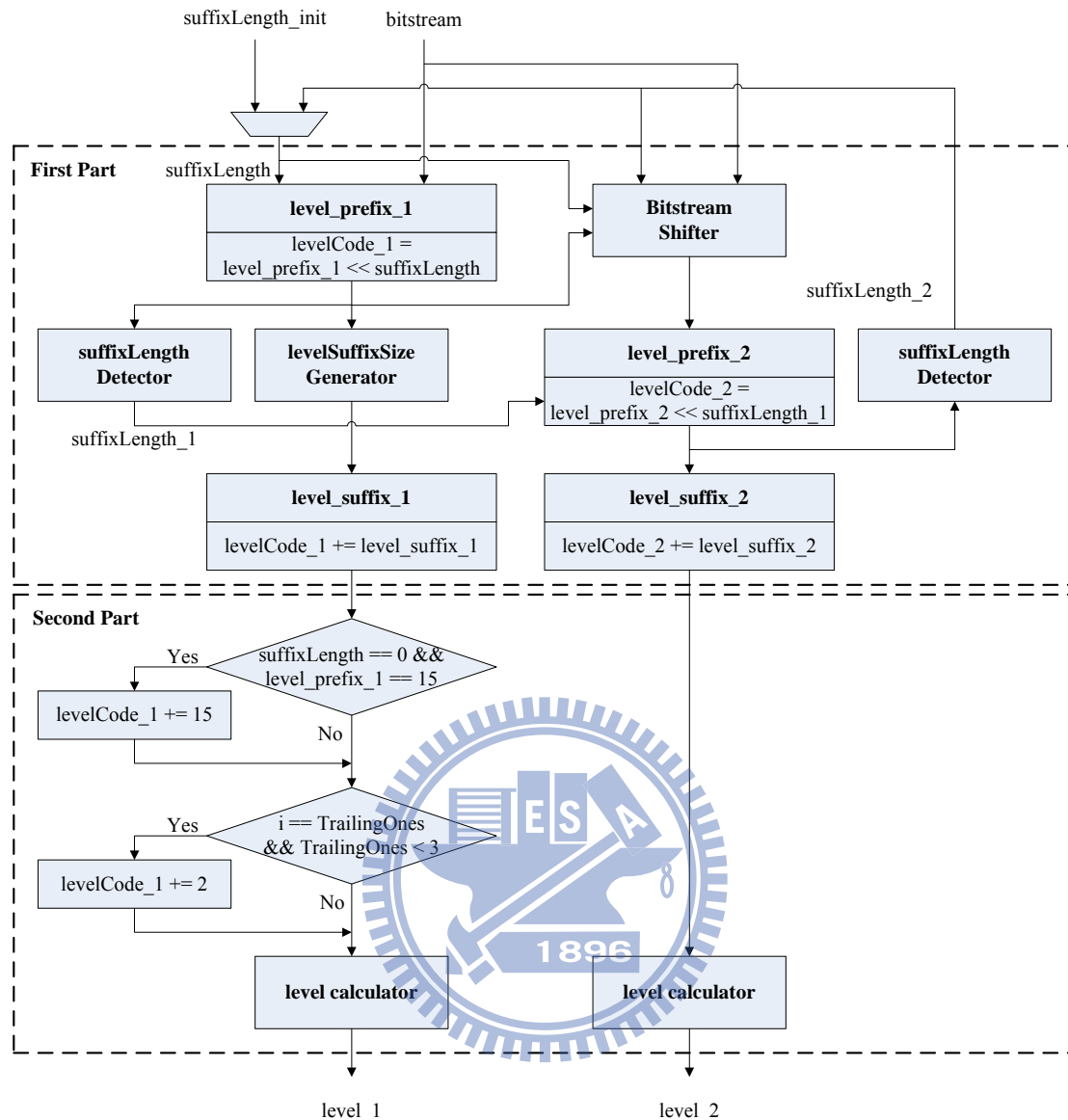


Figure 19. Proposed delay balanced two-level decoding architecture.

4.1.3 CAVLC Decoding Architecture Design

Based on the DBTLD engine, the CAVLC decoding architecture is designed as shown in Fig. 20. In the *trailing_ones_sing_flag* decoding unit, all sign flags are scanned in one cycle. After *level* decoding procedure is done, all nonzero coefficients are stored in a 16-entry deep and 13-bit wide output buffer. Finally, in the *run_before* decoding unit, whenever a *run_before* symbol is decoded, the corresponding *level* is transmitted to its actual position in the output buffer. Since only one output buffer is

used instead of storing *level* and *run_before* information separately, to regularize the data transmission of output buffer, the prediction-based *run_before* look-up table combination method [7] is employed that two *run_before* symbols are decoded in one cycle except when only one *run_before* symbol left. Fig. 21 shows the architecture of residual block reconstruction. After *TrailingOnes* and *levels* are pushed in the output buffer in order, in each cycle, one or two *level* symbols are moved to their final locations respectively depending on the *coeffsLeft* and *zerosLeft* information. The movement starts from the last coefficient and ends until no more *run_befores* are decoded. The parameters *coeffsLeft* denotes the remaining number of nonzero coefficients needs to be moved, and *zerosLeft* represents the remaining number of zeros to be decoded. Table 10 shows an example for the reconstruction process. In the beginning, all nonzero coefficients are arranged in order, output buffer index 0 to (*TotalCoeffs* - 1). After *total_zeros* is decoded, coefficients are moved to the indices which are calculated as (*coeffsLeft* + *zerosLeft* - 1) in reverse order, and the value of the original position of the moved coefficient is replaced by 0. In this example, first, the last coefficient 1 is moved to index 8 (6 + 3 - 1), and the coefficient -1 is moved to index 6 (5 + 2 - 1). In the next cycle, only the one *run_before* symbol is valid since no more zeros left to be decoded, and the coefficients -2 is moved to index 4 (4 + 1 - 1).

To further accelerate the decoding procedure, skipping mechanism is employed to remove redundant decoding processes:

- 1) Zero block skip: When *TotalCoeffs* is equal to 0, the remaining decoding processes are skipped since nonzero coefficients do not exist in the block.
- 2) Level skip: When *TotalCoeffs* is equal to *TrailingOnes*, the *level* decoding procedure is skipped since there has no nonzero coefficients left to be decoded.
- 3) Total zeros skip: When *TotalCoeffs* is equal to maximum number of

coefficients (*maxNumCoeff*), the *total_zeros* decoding procedure and *run_before* decoding procedure is bypassed because there are no zero coefficients to be decoded.

- 4) Run skip: When *total_zeros* is equal to 0 or *TotalCoeffs* is equal to 1, *run_before* decoding procedure is not necessary.

Moreover, in the CAVLC decoding procedure, because *coeff_token*, *trailing_ones_sing_flag*, *level*, *total_zeros*, and *run_before* decoding units are not performed simultaneously, only one of them is designated to work in each cycle, to save power consumption, idled units are turned off by functional gating.

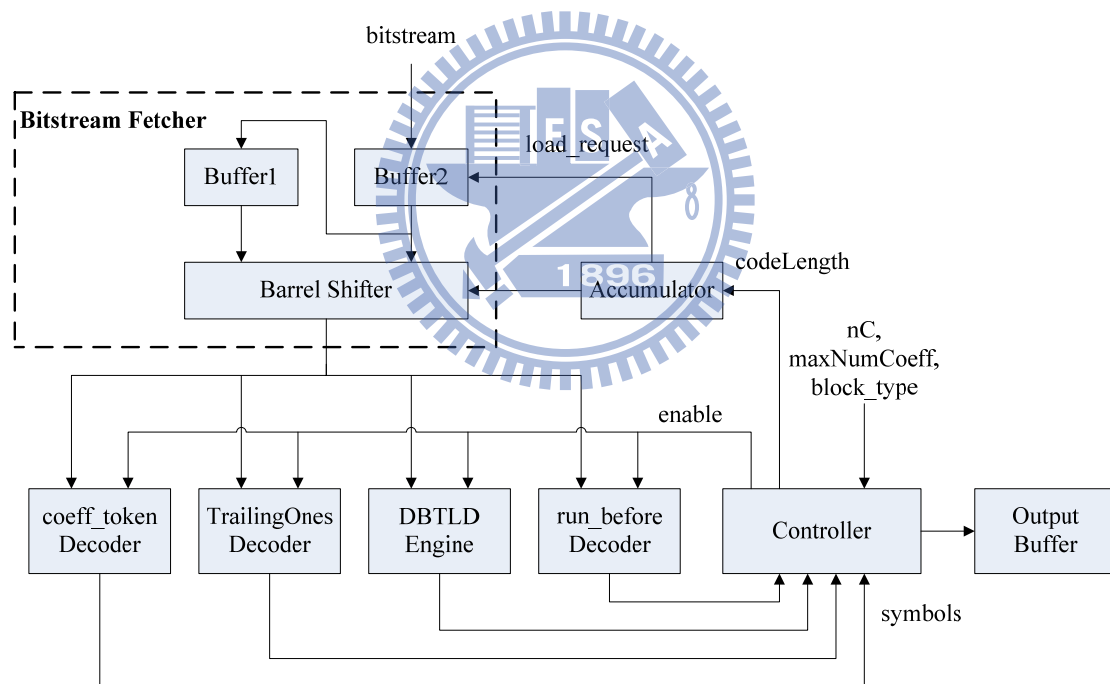


Figure 20. Proposed CAVLC decoder.

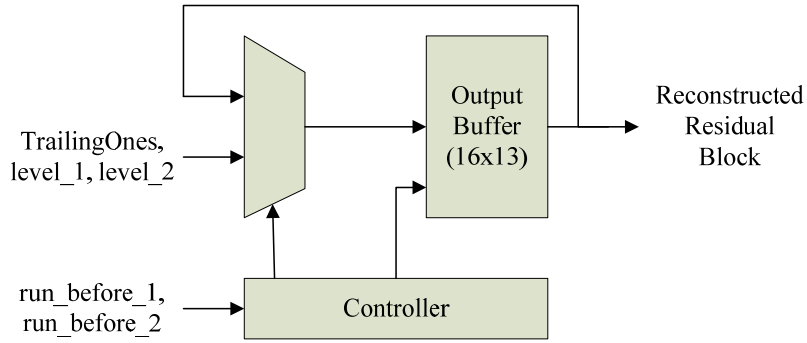


Figure 21. Residual block reconstruction architecture.

TABLE 10. EXAMPLE OF RESIDUAL BLOCK RECONSTRUCTION PROCESS

Decoded Symbol	<i>coeffsLeft</i>	<i>zerosLeft</i>	Output Buffer															
			4	3	2	-2	-1	1	0	0	0	0	0	0	0	0	0	0
<i>total_zeros</i> = 3	x	x	4	3	2	-2	-1	1	0	0	0	0	0	0	0	0	0	
<i>run_before_1</i> = 1 <i>run_before_2</i> = 1	6	3	4	3	2	-2	0	0	-1	0	1	0	0	0	0	0		
<i>run_before_1</i> = 1 <i>run_before_2</i> = x	4	1	4	3	2	0	-2	0	-1	0	1	0	0	0	0	0		

4.1.4 Experimental Results

Table 11 shows the decoding performance of the proposed CAVLC decoder for different video sequences. To compare with previous works fair, we use the same testing environment that all the sequences with resolution of QCIF (176 x 144) are intra coded. The RTL simulation result shows in the low bit-rate coding like high QP or simple image, since the residual block is very sparse, Lee's design [9] which only focus on boosting *run_before* decoding procedure can achieve higher decoding speed. However, in the high bit-rate coding, the demand for high decoding speed is actually necessary, our proposed design that takes both *level* and *run_before* decoding procedures into consideration prevails over other existing designs.

The synthesis results of the proposed CAVLC decoder and a comparison of

hardware cost and decoding speed with other existing work are shown in Table 12. The proposed CAVLC decoder is synthesized with UMC 90nm. We enhance the throughput by exploiting multi-symbol decoding scheme for both *level* and *run_before* symbols while allowing the maximum working frequency to be about 390 MHz with 13.88k gate count. Lin’s design [4] has minimum hardware cost, however, its decoding speed of the two main critical loops, *level* decoding procedure and *run_before* decoding procedure that dominate the overall decoding performance, is only one symbol per cycle, which is merely half in comparison to our design. By applying the prediction-based *run_before* look-up table combination method [7], two *run_bofore* symbols can be decoded in each cycle. Furthermore, with the DBTLD engine, not only two *level* symbols can be decoded at the same cycle, but also 21% critical path delay is saved in comparison to the traditional two-level decoder. Table 13 shows the maximum frame rates (frames per second) for different Level limits defined in the H.264/AVC standard. According to the definition and the throughput of our design, we list the minimum working frequency requirement of Level in Table 14. The result shows that our proposed CAVLC decoder can achieve real-time decoding for all Level conditions.

TABLE 11. COMPARISON OF CAVLC DECODING PERFORMANCE

Video Sequence	QP	Bitrate(Mbps)	Average cycle/MB			
			Proposed	Yu [7]	Lee [9]	Tsai [6]
Akiyo	28	0.59	44	50	N/A	39
	20	1.13	75	93	N/A	N/A
	12	2	117	154	N/A	143
Foreman	28	0.83	58	68	N/A	N/A
	20	1.76	116	151	N/A	N/A
	12	3.12	182	259	N/A	N/A
Mobile	28	2.21	145	194	135	150

	20	3.66	203	300	211	N/A
	12	5.32	233	367	264	241
News	28	0.83	58	70	49	N/A
	20	1.53	95	125	87	N/A
	12	2.58	141	195	138	N/A
Stefan	28	1.5	102	133	97	106
	20	2.58	150	214	154	N/A
	12	3.94	188	282	204	201
Average			127.13	177	148.8	146.67
Reduction (%)				28.18	14.56	13.32

TABLE 12. CAVLC DECODER IMPLEMENTATION RESULT COMPARISONS OF DIFFERENT DESIGNS

Specifications	Proposed		Lin [4]	Yu [7]	Lee [9]	Alle [5]	Tsai [6]
Technology	90nm	0.18um	0.18um	0.18um	0.13um	0.13um	0.18um
Max. Frequency	385 MHz	193 MHz	213 MHz	125 MHz	125MHz	250MHz	160 MHz
Area: Logic Part (gate count)	13,544	14,373	6,771	13,192	15,602	17,202	13,189
Area: Memory Part (bits)	W/O		W/O	W/O	W/O	5,120	W/O
Average cycle/MB	127.13		N/A	177	148.8	N/A	146.67

TABLE 13. MAXIMUM FRAME RATES FOR SOME EXAMPLE FRAME SIZES

Level			1	1b	1.1	1.2	1.3	2	2.1	2.2	3	3.1	3.2	4	4.1	4.2	5	5.1
Max MBs/frame			99	99	396	396	396	396	792	1620	1620	3600	5120	8192	8192	8704	22080	36864
Format	Resolution (W x H)	MBs Total																
SQCIF	128x96	48	30.9	30.9	62.5	125.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0
QCIF	176x144	99	15.0	15.0	30.3	60.6	120.0	120.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0
QVGA	320x240	300	-	-	10.0	20.0	39.6	39.6	66.0	67.5	135.0	172.0	172.0	172.0	172.0	172.0	172.0	172.0
525 SIF	352x240	330	-	-	9.1	18.2	36.0	36.0	60.0	61.4	122.7	172.0	172.0	172.0	172.0	172.0	172.0	172.0
CIF	352x288	396	-	-	7.6	15.2	30.0	30.0	50.0	51.1	102.3	172.0	172.0	172.0	172.0	172.0	172.0	172.0
525 HHR	352x480	660	-	-	-	-	-	-	30.0	30.7	61.4	163.6	172.0	172.0	172.0	172.0	172.0	172.0
625 HHR	352x576	792	-	-	-	-	-	-	25.0	25.6	51.1	136.4	172.0	172.0	172.0	172.0	172.0	172.0
VGA	640x480	1200	-	-	-	-	-	-	-	16.9	33.8	90.0	172.0	172.0	172.0	172.0	172.0	172.0
525 4SIF	704x480	1320	-	-	-	-	-	-	-	15.3	30.7	81.8	163.6	172.0	172.0	172.0	172.0	172.0
525 SD	720x480	1350	-	-	-	-	-	-	-	15.0	30.0	80.0	160.0	172.0	172.0	172.0	172.0	172.0
4CIF	704x576	1584	-	-	-	-	-	-	-	12.8	25.6	68.2	136.4	155.2	155.2	172.0	172.0	172.0
625 SD	720x576	1620	-	-	-	-	-	-	-	12.5	25.0	66.7	133.3	151.7	151.7	172.0	172.0	172.0
SVGA	800x600	1900	-	-	-	-	-	-	-	-	-	56.8	113.7	129.3	129.3	172.0	172.0	172.0
XGA	1024x768	3072	-	-	-	-	-	-	-	-	-	35.2	70.3	80.0	80.0	172.0	172.0	172.0
720p HD	1280x720	3600	-	-	-	-	-	-	-	-	-	30.0	60.0	68.3	68.3	145.1	163.8	172.0
4VGA	1280x960	4800	-	-	-	-	-	-	-	-	-	-	45.0	51.2	51.2	108.8	122.9	172.0
SXGA	1280x1024	5120	-	-	-	-	-	-	-	-	-	-	42.2	48.0	48.0	102.0	115.2	172.0
525 16SIF	1408x960	5280	-	-	-	-	-	-	-	-	-	-	-	46.5	46.5	98.9	111.7	172.0
16CIF	1408x1152	6336	-	-	-	-	-	-	-	-	-	-	-	38.8	38.8	82.4	93.1	155.2
4SVGA	1600x1200	7500	-	-	-	-	-	-	-	-	-	-	-	32.8	32.8	69.6	78.6	131.1
1080 HD	1920x1088	8160	-	-	-	-	-	-	-	-	-	-	-	30.1	30.1	64.0	72.3	120.5
2Kx1K	2048x1024	8192	-	-	-	-	-	-	-	-	-	-	-	30.0	30.0	63.8	72.0	120.0
2Kx1080	2048x1088	8704	-	-	-	-	-	-	-	-	-	-	-	-	-	60.0	67.8	112.9
4XGA	2048x1536	12288	-	-	-	-	-	-	-	-	-	-	-	-	-	-	48.0	80.0
16VGA	2560x1920	19200	-	-	-	-	-	-	-	-	-	-	-	-	-	-	30.7	51.2
3616x1536	3616x1536	21696	-	-	-	-	-	-	-	-	-	-	-	-	-	-	27.2	45.3
3672x1536	3680x1536	22080	-	-	-	-	-	-	-	-	-	-	-	-	-	-	26.7	44.5
4Kx2K	4096x2048	32768	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	30.0
4096x2304	4096x2304	36864	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	26.7

TABLE 14. WORKING FREQUENCY FOR DIFFERENT LEVEL CONDITIONS

Level	Max. MBs/frame	Max. MB Processing Rate (MBs/s)	Working Frequency
1	99	1,458	0.19 MHz
1b	99	1,458	0.19 MHz
1.1	396	3,000	0.39 MHz
1.2	396	6,000	0.77 MHz
1.3	396	11,880	1.52 MHz
2	396	11,880	1.52 MHz
2.1	792	19,800	2.52 MHz
2.2	1,620	20,250	2.58 MHz
3	1,620	40,500	5.15 MHz
3.1	3,600	108,000	13.73 MHz
3.2	5,120	216,000	27.46 MHz
4	8,192	245,760	31.25 MHz
4.1	8,192	245,760	31.25 MHz
4.2	8,704	522,240	66.4 MHz
5	22,080	589,824	74.95 MHz
5.1	36,864	983,040	125.13 MHz

4.2 Proposed CABAC Decoder

Since it is obvious that the main obstacle to adopting pipelining scheme for CABAC decoder comes from data hazards, the design of pipelining stages shall be considered carefully. We are concerned about whether there are factors that dominate the decoding performance. If the answer is affirmative, we can adjust our design to those cases for achieving better decoding performance. Consequently, first of all, we investigate the characteristics of SE parsing flow and bin distribution among SEs.

4.2.1 Analysis

The SE parsing flow is mainly dependent on conditional branches as illustrated in Fig. 22. Branches denoted by “*” indicate that the condition of branch and the current SE value are independent. In other words, the next SE type to be decoded

right after current SE can be decided before the current SE decoding is completed. Therefore, for this kind of branches, the context models used for decoding the next SE can be prepared in advance to prevent pipelining stall. However, most branches denoted by “#” are dependent on the current SE value. Not until the current SE value is ascertained can the next SE type be determined.

Table 15 – Table 23 list the analyzed results of bin distribution based on the video sequences with HD 1920x1080, 4:2:0 color format and frame rate 30 fps encoded by H.264/AVC reference software JM 12.2. From the statistic, we can observe that the proportion of `significant_coeff_flag` and `last_significant_coeff_flag` can reach up to 50% of total bins. Furthermore, the SE switching rate (number of decoded SEs / number of decoded bins) is about 68% in average (see Table 24 – Table 26), and over 90% of SE switches comes from the significance map. Consequently, it is apparent that how to deal with `significant_coeff_flag` and `last_significant_coeff_flag` is the key to solve the problem invoked by data hazards.

Fig. 23 shows the architecture of our proposed CABAC decoder design. In our architecture, we divide the chain of operations into two stages, modified context model selection (MCS) stage and TSBAD stage. MCS stage contains CS and CL. TSBAD stage includes a two-symbol decoding engine and CU. The detailed description of the architecture is presented in the following subchapter.

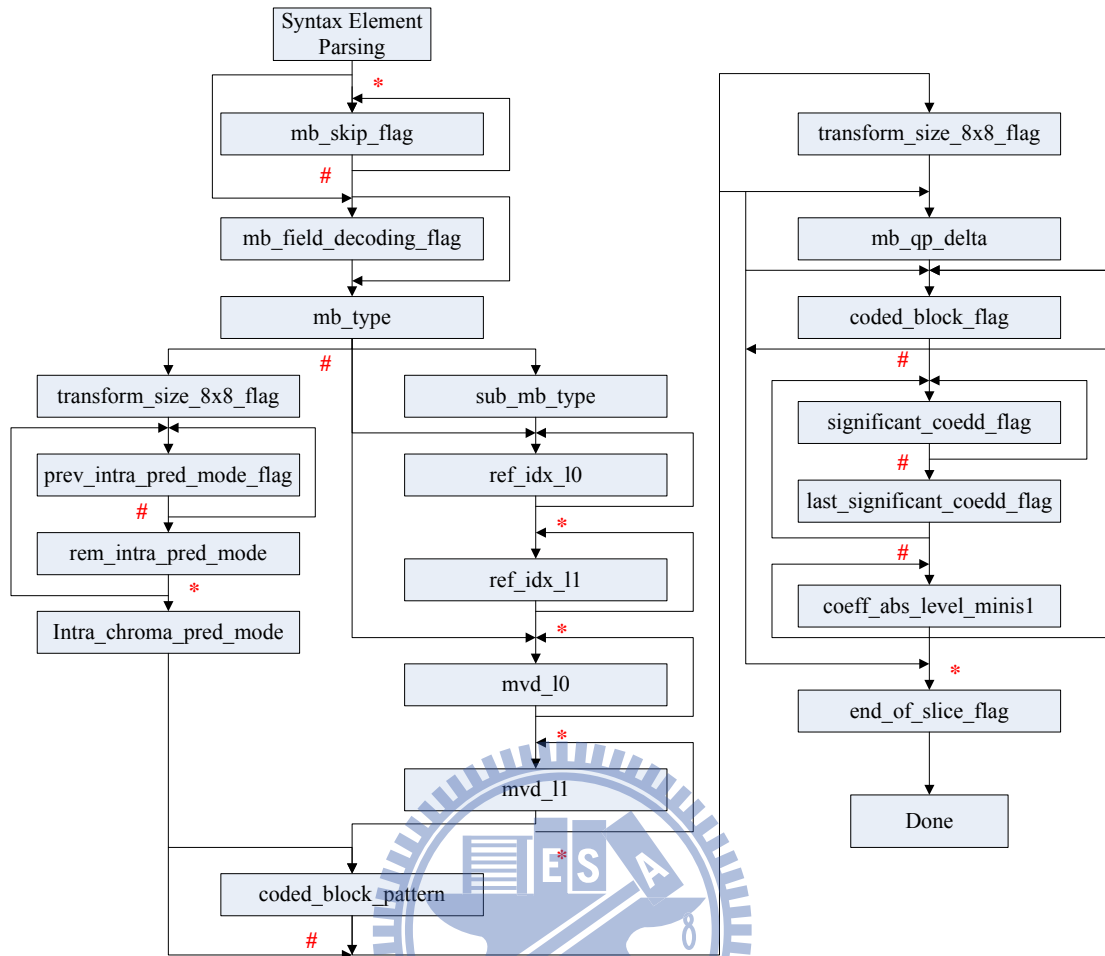


Figure 22. SE parsing flow for the H.264/AVC.

TABLE 15. STATISTICAL RESULT OF BIN DISTRIBUTION WITH I CODING STRUCTURE AND QP28

Syntax Element	Video Sequence (I_QP28)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	2.48	0.88	7.12	1.88	2.15	0.77	2.55
mb_skip_flag	0	0	0	0	0	0	0
intra_pred_mode	9.93	8.24	11.52	11.19	12.18	6.42	9.91
mvd	0	0	0	0	0	0	0
coded_block_pattern	5.04	3.91	6.75	4.6	5.82	3.3	4.9
coded_block_flag	5.91	2.96	3.9	3.17	5.35	4.5	4.3
significant_coeff_flag	28.64	29.85	19.26	28.12	19.63	29.78	25.88
last_significant_coeff_flag	12.5	14.09	12.01	13.79	13.51	14.52	13.4
coeff_abs_level_minus1	30.33	36.78	30.68	32.96	35.87	37.9	34.09

TABLE 16. STATISTICAL RESULT OF BIN DISTRIBUTION WITH I CODING STRUCTURE AND QP20

Syntax Element	Video Sequence (I_QP20)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	0.49	0.27	0.84	0.43	0.56	0.26	0.48
mb_skip_flag	0	0	0	0	0	0	0
intra_pred_mode	4.94	4.24	6.69	5.38	6.36	3.48	5.18
mvd	0	0	0	0	0	0	0
coded_block_pattern	2.02	1.54	3.4	1.74	2.89	1.37	2.16
coded_block_flag	3.58	2.93	3.28	3.23	4.53	2.89	3.41
significant_coeff_flag	37.78	33.84	39.29	35.31	24.12	31.05	33.57
last_significant_coeff_flag	13.62	14.04	11.7	14.26	14.3	14.83	13.79
coeff_abs_level_minus1	35.77	41.9	31.83	38.18	44.78	44.98	39.57

TABLE 17. STATISTICAL RESULT OF BIN DISTRIBUTION WITH I CODING STRUCTURE AND QP12

Syntax Element	Video Sequence (I_QP12)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	0.15	0.11	0.23	0.17	0.19	0.11	0.16
mb_skip_flag	0	0	0	0	0	0	0
intra_pred_mode	3.18	3.37	3.62	3.26	3.86	2.57	3.31
mvd	0	0	0	0	0	0	0
coded_block_pattern	0.78	0.64	0.89	0.67	0.96	0.61	0.76
coded_block_flag	2.2	2.02	2.44	2.03	2.5	1.81	2.17
significant_coeff_flag	31	27.31	36.17	29.41	31.98	25.74	30.27
last_significant_coeff_flag	15.92	15.32	15.62	15.83	13.76	14.87	15.22
coeff_abs_level_minus1	46.1	50.73	40.27	48.09	45.92	53.78	47.48

TABLE 18. STATISTICAL RESULT OF BIN DISTRIBUTION WITH IPPP CODING STRUCTURE AND QP28

Syntax Element	Video Sequence (IPPP_QP28)	Average
----------------	----------------------------	---------

	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	(%)
mb_type	5.59	1.58	8.8	7.35	6.16	3.77	5.54
mb_skip_flag	2.87	0.7	4.11	5.47	6.05	1.76	3.49
intra_pred_mode	6.76	7.92	3.67	4.82	3.7	1.83	4.78
mvd	8.11	0.86	21.6	12.92	16.13	15.4	12.50
coded_block_pattern	8.54	3.98	9.78	11.33	10.83	6.87	8.56
coded_block_flag	5.33	2.86	2.22	3.82	3.86	4.78	3.81
significant_coeff_flag	20.62	29.28	14.36	17.91	14.88	24.77	20.3
last_significant_coeff_flag	10.3	13.75	7.97	8.19	8.48	10.83	9.92
coeff_abs_level_minus1	24.88	35.82	19.15	19.65	20.49	25.63	24.27

TABLE 19. STATISTICAL RESULT OF BIN DISTRIBUTION WITH IPPP CODING STRUCTURE AND QP20

Syntax Element	Video Sequence (IPPP_QP20)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	1.49	0.54	2.51	1.51	3.65	0.95	1.78
mb_skip_flag	0.59	0.25	0.92	0.57	1.64	0.34	0.72
intra_pred_mode	2.75	4.17	3.16	0.99	1.44	0.7	2.2
mvd	3.8	0.1	11.69	5.46	11.98	4.79	6.3
coded_block_pattern	3.04	1.56	4.46	2.87	6.93	1.95	3.47
coded_block_flag	4.79	2.92	3.92	4.86	5.39	4.03	4.32
significant_coeff_flag	39.71	33.49	32.7	45.85	26.74	39.4	36.32
last_significant_coeff_flag	12.06	13.99	10.88	11.24	11.08	13.8	12.18
coeff_abs_level_minus1	29.48	41.75	26.58	24.53	27.17	32.71	30.37

TABLE 20. STATISTICAL RESULT OF BIN DISTRIBUTION WITH IPPP CODING STRUCTURE AND QP12

Syntax Element	Video Sequence (IPPP_QP12)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	0.37	0.21	0.4	0.38	0.59	0.32	0.38
mb_skip_flag	0.14	0.1	0.16	0.13	0.2	0.12	0.14
intra_pred_mode	1.97	3.36	3.42	1.49	1.54	0.88	2.11
mvd	0.77	0.01	0.68	1.11	3.98	1.45	1.33
coded_block_pattern	0.86	0.64	0.93	0.79	1.23	0.71	0.86

coded_block_flag	2.22	2.01	2.47	1.96	3.84	1.65	2.36
significant_coeff_flag	34.46	27.3	36.83	34.92	38.19	31.13	33.81
last_significant_coeff_flag	16.51	15.29	15.72	17.02	14.71	16.61	15.98
coeff_abs_level_minus1	42.05	50.58	38.63	41.61	34.62	46.61	42.35

TABLE 21. STATISTICAL RESULT OF BIN DISTRIBUTION WITH IBBBP CODING STRUCTURE AND QP28

Syntax Element	Video Sequence (IBBBP_QP28)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	7.34	3.83	10.22	8.38	6.29	4.31	6.73
mb_skip_flag	2.86	0.69	4.26	5.09	6.1	1.82	3.47
intra_pred_mode	6.25	7.34	3.96	4.89	4.39	2.42	4.88
mvd	6.74	2.02	20.35	13.92	14.21	14.23	11.91
coded_block_pattern	9.09	3.89	10.46	12.53	11.31	6.37	8.94
coded_block_flag	5.62	3.04	2.67	3.74	3.9	5.01	4
significant_coeff_flag	19.09	28.07	12.62	15.94	13.92	22.98	18.77
last_significant_coeff_flag	10.03	13.29	7.37	7.46	8.3	10.82	9.55
coeff_abs_level_minus1	25.29	34.66	18.18	18.32	20.69	26.46	23.93

TABLE 22. STATISTICAL RESULT OF BIN DISTRIBUTION WITH IBBBP CODING STRUCTURE AND QP20

Syntax Element	Video Sequence (IBBBP_QP20)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	2.37	1.45	4.1	2.48	3.95	1.39	2.62
mb_skip_flag	0.67	0.25	1.04	0.74	1.71	0.35	0.79
intra_pred_mode	3.19	4.02	2.98	1.6	2.12	1.32	2.54
mvd	3.39	0.33	13.41	6.74	11.05	4.47	6.57
coded_block_pattern	3.24	1.55	4.77	3.2	6.44	1.99	3.53
coded_block_flag	5.33	2.9	4.48	5.59	5.49	4.3	4.68
significant_coeff_flag	35.68	33.01	28.04	40.48	23.73	35.81	32.79
last_significant_coeff_flag	12.16	13.85	10.62	11.12	11.25	13.87	12.15
coeff_abs_level_minus1	31.07	41.4	26.44	24.97	29.29	34.86	31.34

TABLE 23. STATISTICAL RESULT OF BIN DISTRIBUTION WITH IBBBP CODING STRUCTURE AND QP12

Syntax Element	Video Sequence (IBBBP_QP12)						Average (%)
	Pedestrian_area	Riverbed	Rush_hour	Station2	Sunflower	Tractor	
mb_type	0.68	0.6	0.98	0.66	1.03	0.55	0.75
mb_skip_flag	0.14	0.1	0.16	0.14	0.22	0.12	0.15
intra_pred_mode	1.95	3.35	2.82	1.2	1.58	1.16	2.01
mvd	0.78	0.04	1.33	1.36	3.27	1.51	1.38
coded_block_pattern	0.87	0.64	0.9	0.82	1.32	0.72	0.88
coded_block_flag	2.58	2.03	2.61	2.05	4.42	1.77	2.58
significant_coeff_flag	34.38	27.19	36.79	35.89	36.93	30.72	33.65
last_significant_coeff_flag	16.39	15.24	15.65	16.84	14.58	16.14	15.81
coeff_abs_level_minus1	41.49	50.31	37.94	40.34	35.51	46.69	42.05

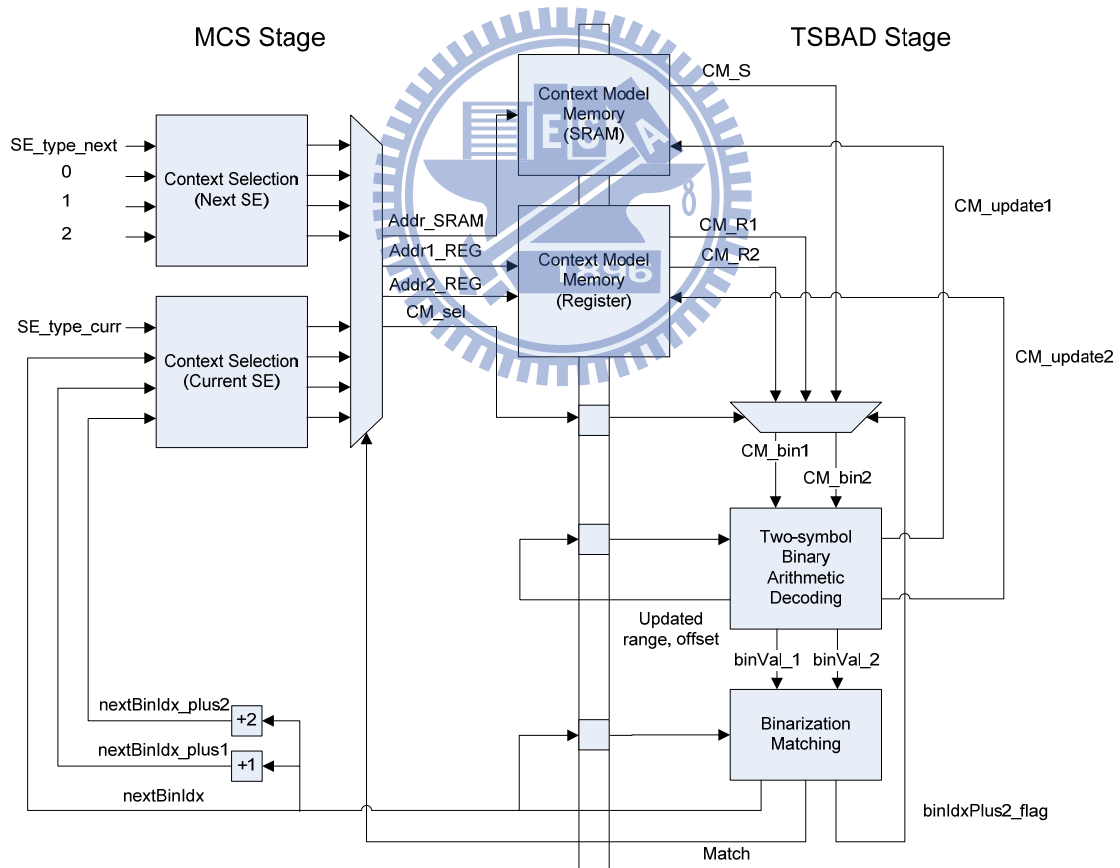


Figure 23. Proposed CABAC decoder architecture.

4.2.2 MCS Stage

The main idea of MCS stage is to select CMs for decoding the next two bins. To simplify and regularize the MCS process, we restrict the two-symbol decoding to a single SE only so that the bin index of the first bin is always even and the second bin is always odd for all SEs. This restriction also matches the property of SE parser that can only parse SEs one by one. As a result, the assignment of CMs to next two bins is regular, and the calculation of CM addresses becomes much simpler. However, this restriction still results in drastic performance degradation due to frequent syntax element switching. To reduce the performance degradation while avoiding being burdened with hardware cost overhead, we propose an approach to predict the type of next SE. Since the high correlation between the features of image in spatial domain, the value of current SE is predictable by referring to the neighboring SEs. Thus by assuming that the value of current SE is the same as its latest value, we can effectively predict what type of SE is coming next. With the proposed scheme, the penalty of prediction miss is merely one cycle as illustrated in Fig. 24. Benefited by the proposed prediction-based method, we can achieve about 80% prediction accuracy in average as shown in Table 24 – Table 26. Note that Hit Rate = (number of prediction hits) / (number of decoded SEs).

To further improve the accuracy of prediction, we merge all symbols of the significance map which is composed of `significant_coeff_flag` and `last_significant_coeff_flag` as an individual SE by exploiting their decoding regularity since `significant_coeff_flag` and `last_significant_coeff_flag` account for over 90% of prediction miss. As a result, predictions for the branches right after `significant_coeff_flag` and `last_significant_coeff_flag` are not necessary anymore. Compared with the predictor which does not perform SE merging, the combination of SE merging method and prediction-based method can achieve about 17% higher prediction accuracy in average, as shown in Table 24 – Table 26. Moreover, for high

bit-rate coding such as QP equaling to 12, the prediction accuracy can reach over 99%.

After applying the SE merging method, the bin index transition of significance map can be summarized in Table 27, and the binarization matching condition becomes when current bin is `last_significant_coeff_flag` and its bin value is 1, or the current bin index meets the final bin index ($((binIdx \% 2 = 1) \&\& (binVal = 1)) \parallel (binIdx = numCoeff - 1)$). From this table, we can observe that only one case that $nextBinIdx$ equals to $(binIdx + 2)$ takes place when current symbol is `significant_coeff_flag` and its bin value is 0 ($(binIdx \% 2 = 0) \&\& (binVal = 1)$). As a result, CM selection and assignment for significance map can still only depend on the bin index of next two bins.

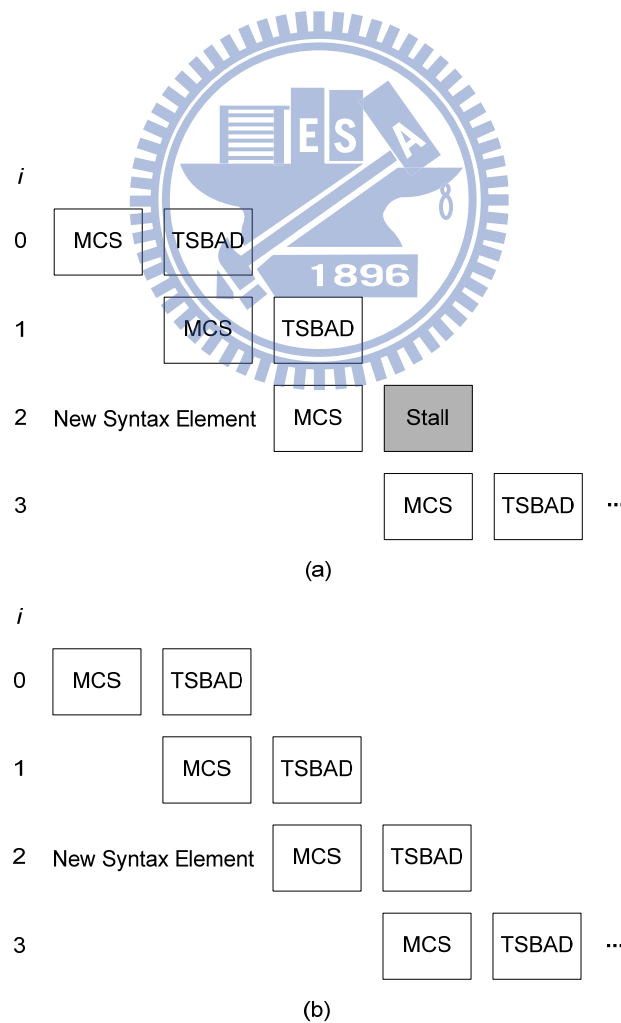


Figure 24. Pipeline scheduling of (a) prediction miss and (b) prediction hit.

TABLE 24. IMPROVEMENT OF PREDICTION ACCURACY USING THE PROPOSED METHODS WITH I CODEING STRUCTURE

Video Sequence (I)	Q P	SE Switching Rate (%)	Hit Rate (%)	
			Without SE Merging	With SE Merging
Pedestrian_a	28	72.89	79.08	95.66
	20	74.06	80.53	98.22
	12	67.88	80.73	99.51
Riverbed	28	70.04	80.89	97.7
	20	68.98	82.08	98.8
	12	62.66	82.54	99.71
Rush_hour	28	65.66	81.45	95.67
	20	73.78	82.68	97.71
	12	72.96	78.95	99.39
Station2	28	71.29	79.01	97.14
	20	72.15	81.07	98.53
	12	65.76	81.66	99.66
Sunflower	28	65.71	81.32	95.42
	20	64.1	83.3	97.52
	12	65.33	82.65	99.55
Tractor	28	71.01	80.12	97.13
	20	67.39	82.81	99.26
	12	59.61	83.53	99.84
Average		68.4	81.36	98.13

TABLE 25. IMPROVEMENT OF PREDICTION ACCURACY USING THE PROPOSED METHODS WITH IPPP CODEING STRUCTURE

Video Sequence (IPPP)	Q P	SE Switching Rate (%)	Hit Rate (%)	
			Without SE Merging	With SE Merging
Pedestrian_a	28	65.95	79.47	92.65
	20	75.38	79.48	96.6
	12	72.24	79.09	99.39

Riverbed	28	69.36	80.85	97.53
	20	68.73	81.98	98.72
	12	62.66	82.54	99.7
Rush_hour	28	55.23	79.8	91.63
	20	67.77	79.39	95.57
	12	73.98	78.6	99.29
Station2	28	63.73	79.67	91.67
	20	79.36	78.2	96.77
	12	73.15	79.17	99.53
Sunflower	28	61.66	79.77	90.46
	20	67.01	77.8	93.09
	12	74.95	78.02	98.76
Tractor	28	64.46	77.5	93.13
	20	75.01	78.08	97.78
	12	67.86	81.16	99.71
Average		68.81	79.48	96.22

TABLE 26. IMPROVEMENT OF PREDICTION ACCURACY USING THE PROPOSED METHODS WITH IBBBP CODEING STRUCTURE

Video Sequence (IBBBP)	Q P	SE Switching Rate (%)	Hit Rate (%)	
			Without SE Merging	With SE Merging
Pedestrian_a	28	64.11	79.65	92.14
	20	72.68	79.4	96.11
	12	72.3	78.93	99.38
Riverbed	28	67.09	80.66	97.24
	20	67.87	81.97	98.69
	12	62.47	82.54	99.7
Rush_hour	28	54.4	80.68	90.98
	20	64.16	79.35	94.71
	12	73.73	78.37	99.23
Station2	28	61.45	80.58	91.88
	20	75.94	78.23	96.03
	12	73.82	78.84	99.53
Sunflower	28	61.46	79.96	89.99
	20	64.56	78.7	93.08

	12	73.89	78.59	98.61
Tractor	28	63.18	78.02	93.08
	20	72.16	79.07	97.71
	12	66.78	81.31	99.72
Average		67.34	79.71	95.99

TABLE 27. BIN INDEX TRANSITION RELATION IN SIGNIFICANCE MAP

Current Flag	Bin Value	Next Flag	Next <i>binIdx</i>
SIG[<i>i</i>]	0	SIG[<i>i</i> +1]	<i>binIdx</i> + 2
SIG[<i>i</i>]	1	LAST[<i>i</i>]	<i>binIdx</i> + 1
LAST[<i>i</i>]	0	SIG[<i>i</i> +1]	<i>binIdx</i> + 1
LAST[<i>i</i>]	1	X	X

a. *i* denotes scanning position

b. SIG denotes significant_coeff_flag

c. LAST denotes last_significant_coeff_flag

To sum up, for successive two bins, the position of the second bin in a SE may be *binIdx_plus1* (*binIdx* + 1) or *binIdx_plus2* (*binIdx* + 2). It means that by giving two possible CMs, the second bin can be decoded according to the necessary CM chosen by its actual bin index. Furthermore, by means of the prediction-based mechanism, the CMs of predicted next SE and the CMs of current SE can be calculated in parallel. In the end that the value of current SE is confirmed, if the actual result matches what we presume, the CABAC decoder can keep processing without stall. Otherwise, the pipeline has to be stalled for recalculating the context models of next SE. Therefore, we employ two CS modules to calculate SRAM memory address (*Addr_SRAM*) and Register memory addresses (*Addr1_REG* and *Addr2_REG*) in parallel, one for current SE and another one for predicted next SE. For the CS module of next SE, only bin indices 0, 1, and 2 are taking into account, since in the next cycle, it will be transfer into current SE, and thus the calculation for bin indices which are larger than 2 is

redundant. As a result, instead of doubling the hardware to satisfy the requirement that calculating CM memory addresses for current SE and next SE at the same time, unnecessary calculation in the prediction module is removed and the hardware cost overhead is thus suppressed. Finally, the result of BM will determine which one is chosen for CL. Furthermore, because the CM provided for the first bin decoding (CM_{bin1}) may come from the SRAM or the Register port 1 (CM_S or CM_{R1}), and CM provided for the second bin decoding (CM_{bin2}) may come from the Register port 1 or the Register port 2 (CM_{R1} or CM_{R2}), an additional selective signal (CM_{sel}) which is SE-dependent is also transmitted from MCS stage to TSBAD stage.

4.2.3 Context Model Memory Design

In the proposed MCS stage design, to reach the destination of loading 3 specific CMs and storing updated CMs in the same cycle, the design of CM memory shall be considered carefully. On the premise that one clock domain is used, the first way to implement CM memory is to use single-port SRAM. The advantage of single-port SRAM is its low hardware cost. However, single-port SRAM can not perform read operation and write operation simultaneously. Therefore, the operations of CL and CU have to be separated, which results in extra one cycle. Yi *et al.* [11] proposed a context model reservoir (CMR) structure to resolve the conflict between CL and CU caused by structural hazard. CMR is a cache-like structure. Several context models that are probably used are cached in CMR. This allows the decoder to postpone CU and enables the parallel processing of CS and CL. Although the CMR structure is effective, the decoding is stalled for two cycles when CMR switching takes place.

Another way to implement CM memory is to use dual-port SRAM. The hardware cost of dual-port SRAM is higher in comparison to single-port SRAM. In

spite of the advantage that the read operation and write operation can be performed in the same cycle, only one context model can be loaded at every access. Consequently, one single dual-port SRAM can not meet our requirement. A Context Table Reallocation Scheme is presented in [14] to read two CMs at once by dividing the CM memory into two parts: a General Context Memory and Extended Context Memory. However, it does not always work since the reallocation is only designed for specific SEs.

Storing all context models in register is the most convenient way to implement context model memory due to the access of register is extreme free. Nevertheless, the expense of hardware cost is too high. Thus, we propose a more suitable approach to implement the CM memory with hardware cost consideration while maintaining the decoding performance. In the proposed flow, because the two-symbol decoding procedure is restricted to a single SE only, the CL for some SEs are simple such as flag-type SE that only one context model (CM_{bin1}) is necessary for TSABD and the other context model (CM_{bin2}) is redundant. For example, there are three candidate CMs used for decoding $transform_size_8x8_flag$. However, only one of them is necessary for TSBAD since $transform_size_8x8_flag$ is composed of one single bin. Thus, the CL for the second bin can be skipped, and only one CM for the first bin has to be concerned. For this type of SEs, a dual-port SRAM is sufficient to support CL and CU. However, for the other SEs like significance map, the CL is much more complicated. When decoding significance map, the next two bins to be decoded may be two $significant_coeff_flag$ ($SIG[i], SIG[i + 1]$), one $significant_coeff_flag$ and one $last_significant_coeff_flag$ ($SIG[i], LAST[i]$), or one $last_significant_coeff_flag$ and one $significant_coeff_flag$ ($LAST[i], SIG[i + 1]$). Therefore, two CMs of $significant_coeff_flag$ CM set and one CM of $last_significant_coeff_flag$ CM set have to be loaded from CM memory concurrently; moreover, two of them must be updated

and write back. Because of the limitation of number of port of SRAM, it is impossible to realize the desired purpose by a dual-port SRAM. It seems that all register based memory is the only solution.

Fortunately, for the different complexities of CL, it is reasonable to load CMs from different sources and assign them to TSBAD stage according to the SE type and the bin indices of next two bins. As a result, we reorganize the 459 CMs by applying the following principle. For every set of CMs, if two CMs of each set are never used for TSBAD simultaneously, it is stored in dual-port SRAM; otherwise, it is stored in registers. For instance, to satisfy the requirement for loading three CMs (one for `last_significant_coeff_flag` and two for `significant_coeff_flag`) from the CM memory and perform storing operation in the same cycle, `significant_coeff_flag` CM set can be stored in register while `last_significant_coeff_flag` CM set can be stored in SRAM as illustrated in Fig. 25. Guided by the principle, the organization of CM memory is listed in Table 28 and Table 29. After memory addresses are derived, one CM is loaded from SRAM and two CMs are loaded from register at the same time. Our proposed hybrid CM memory, about half is dual-port SRAM and half is register, not only avoids structural hazards caused by CM reading and writing but also reduces the hardware cost overhead significantly in comparison to the implementation of all register approach.

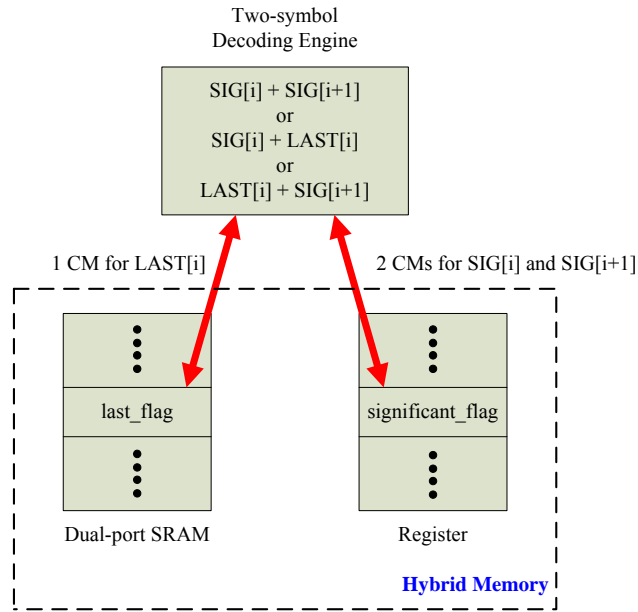


Figure 25. Memory operation in the significance map decoding process.

TABLE 28. CONTENT OF SRAM AFTER REORGANIZATION OF OUR PROPOSAL

Address	CM Index	Syntax Element
0-2	0-2	mb_type (SI)
3-5	11-13	mb_skip_flag (P/SP)
6-8	24-26	mb_skip_flag (B)
9-11	70-72	mb_field_decoding_flag
12-31	85-104	coded_block_flag
32-171	166-226, 338-398, 417-425, 451-459,	last_significant_coeff_flag
172-201	227-231, 237-241, 247-251, 257-261, 266-270, 426-430,	coeff_abs_level_minus1 (First bin)
202-204	399-401	transform_size_8x8_flag

TABLE 29. CONTENT OF REGISTER AFTER REORGANIZATION OF OUR PROPOSAL

Address	CM Index	Syntax Element
0-7	3-10	mb_type (I)
8-14	14-20	mb_type (P/SP)
15-17	21-23	sub_mb_type (P/SP)
18-26	27-35	mb_type (B)
27-30	36-39	sub_mb_type (B)
31-44	40-53	Mvd
45-50	54-59	ref_idx
51-54	60-63	mb_qp_delta
55-58	64-67	intra_chroma_pred_mode
59	68	prev_intra_pred_mode_flag
60	69	rem_intra_pred_mode
61-72	73-84	coded_block_pattern
73-224	105-165, 277-337, 402-416, 436-450,	significant_coeff_flag
225-253	232-236, 242-246, 252-256, 262-265, 271-275, 431-435,	coeff_abs_level_minus1 (First bin excluded)

4.2.4 TSBAD Stage

In the TSBAD stage, first, CM_{bin1} and CM_{bin2} provided for the first bin decoding and the second bin decoding, respectively, is chosen by the selective signal (CM_{sel}). Afterward, in the bin decoding procedure, the updated CMs ($CM_{update1}$ and $CM_{update2}$) are written back into CM memory, and the decoding parameters, interval range and coding offset, are refreshed. Eventually, the values of two bins are passed to the BM module to derive the value of SE and check whether the current SE decoding is done or not.

Following the two-symbol binary arithmetic decoding engine, the final step of this stage is the BM process that maps the constructed binary sequence to nonbinary value. Therefore, the main critical path of this stage occurs in bin value decision of TSBAD engine. In the binary arithmetic decoding procedure, two parameters should be derived and delivered to decode the next bin. One is the updated range and the other is the updated offset. In the traditional TSBAD engine, where two BADs are cascaded directly, the inter-bin dependency of range (R) and offset (O) leads to an unavoidably long critical path. In order to improve decoding performance, a new mathematical transform method for TSBAD procedure is proposed to shorten the critical path. In this thesis, only regular decoding is discussed since implementation of bypass and terminate decoding is much simpler.

According to the H.264/AVC standard, the bin value decision is dependent on O_{LPS} . If O_{LPS} is negative, the $binVal$ is identified as MPS; otherwise, the $binVal$ is identified as LPS. For O_{LPS} to be calculated, a sequential procedure is defined in the standard like Fig. 26(a) shows. To obtain R_{MPS} , it is necessary to run through a 256-to-1 multiplexer first and then do the subtraction. However, a mathematical reordering method [15] can be adopted as follows:

$$O_{LPS} = O - R_{MPS} = O - (R - R_{LPS}) = (O - R) + R_{LPS} \quad (3)$$

In Eq. (3), although $R_{MPS} = R - R_{LPS}$ can not be obtained until R_{LPS} is selected by accessing the look-up table, however, since both R and O are ready in the beginning, the computation of $(O - R)$ and the table look-up for R_{LPS} can be operated in parallel. As a result, benefited by the calculation reordering, a balanced structure can be utilized for reducing the delay of bin value decision process as depicted in Fig. 26(b).

We extend the concept of Eq. (3) to two-symbol two-stage computation.

According to Eq. (3), we perform the mathematical transform for the second bin decision process as shown in Fig. 27, where R'_{LPS} and O'_{LPS} represent R_{LPS} and O_{LPS} of the second stage, respectively. For the reason that O_{LPS} and $(O - R)$ are already calculated in the first bin decision process, the delay of a subtractor can be further eliminated. Note that O_{LPS} and $(O - R)$ have to be shifted 1 to 7 bits according to R_{LPS} since there is a renormalization process between the first bin and the second bin decision procedure.

Fig. 28 shows the detailed architecture of proposed TSBAD engine. In the first bin decision scheme, state index (*stateIdx*) and MPS value (*valMPS*) are extracted from *CM_bin1*. The parameters with word “renorm” denote that they are left-shifted by the renormalization process. The shift amount of MPS case is 0 or 1 depending on the most significant bit of R_{MPS} , whereas the shift amount of LPS case lies in the range 1 to 7. To pass the shifted O_{LPS} and $(O - R)$ to the second bin decision scheme as soon as possible, a table-driven selector is utilized to derive the shift amount of LPS case. In the second bin decision scheme, both cases for previous bin being MPS and LPS are calculated in parallel. With regard to *CM_bin2*, it has to be set to the updated *CM_bin1* when *CM_bin2* and *CM_bin1* are the same. For the reason that the second bin decision process is a parallel working, on the premise that knowing what previous bin is, instead of waiting the updated value of *CM_bin1* is determined, we can access R_{LPS} table immediately and the delay of 64-to-1 multiplexer can be eliminated thus. By using this feature, four possible LPS intervals are selected while performing the first bin decoding procedure. As a result, the main critical path of the second bin decoding is a 4-to-1 multiplexer and an adder. Finally, the value of the second bin is chosen by the most significant bin of O_{LPS} in the first bin part. Note that the updating of R , O , and CM in the second bin decision scheme is not depicted since it is similar to the one in the first bin decision scheme. With the proposed mathematical transform

method, the critical path delay of TSBAD engine is further improved by 28% (from 3.14ns to 2.26ns) compared with the traditional TSBAD engine.

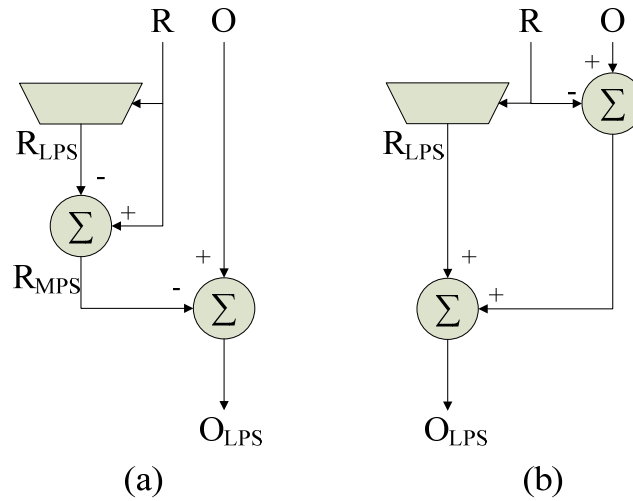


Figure 26. Mathematical reordering. (a) $O - (R - R_{LPS})$. (b) $(O - R) + R_{LPS}$.

if previous bin is MPS then

$$\begin{aligned} O'_{LPS} &= (O_{MPS} - R_{MPS}) + R'_{LPS} \\ &= (O - R_{MPS}) + R'_{LPS} \\ &= O_{LPS} + R'_{LPS} \end{aligned}$$

else

$$\begin{aligned} O'_{LPS} &= (O_{LPS} - R_{LPS}) + R'_{LPS} \\ &= (O - R + R_{LPS} - R_{LPS}) + R'_{LPS} \\ &= (O - R) + R'_{LPS} \end{aligned}$$

Figure 27. Mathematical transform for the second bin decision process.

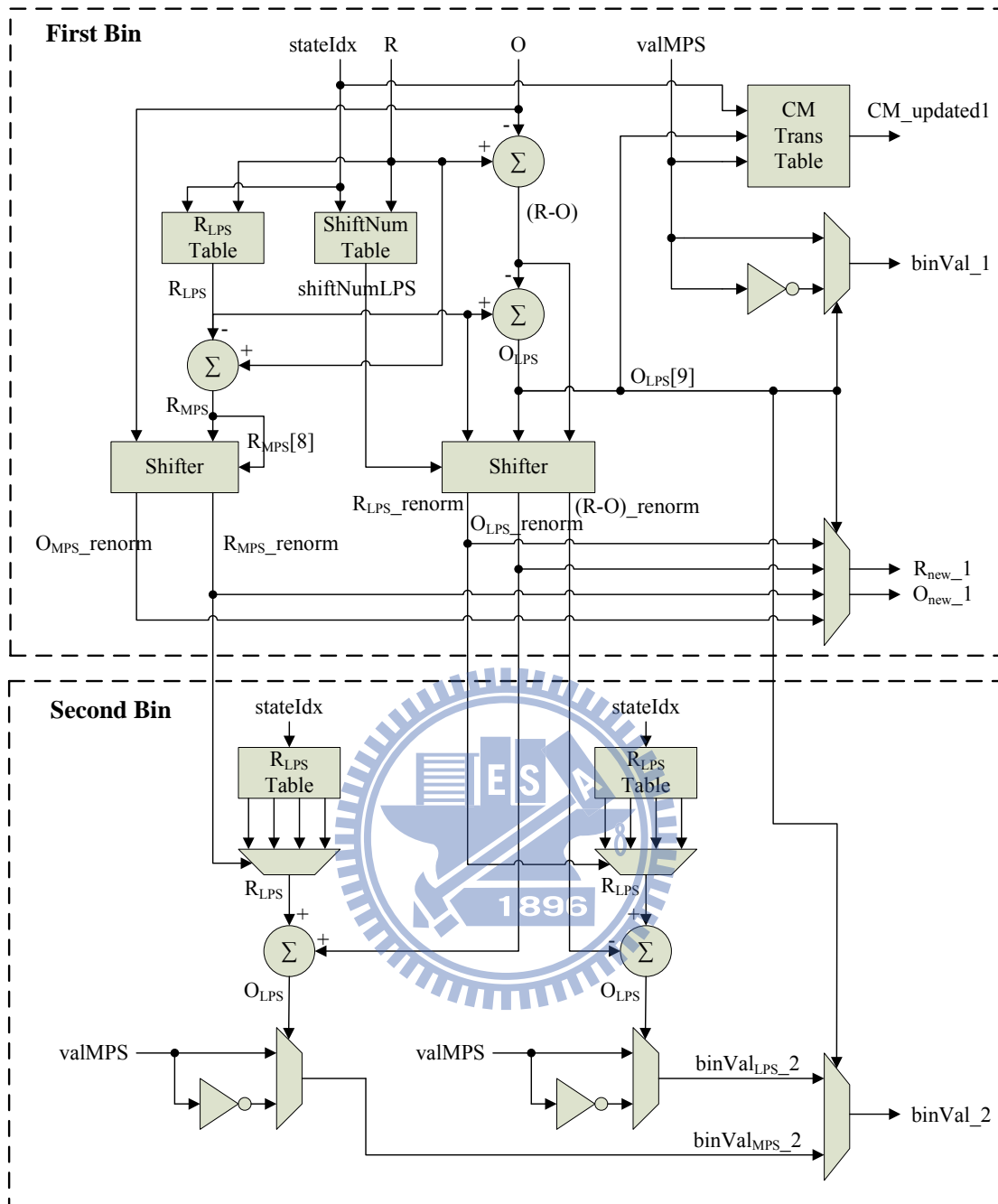


Figure 28. Architecture of proposed two-symbol arithmetic decoding engine.

4.2.5 Experimental Results

Table 30 – Table 32 show the decoding performance of the proposed architecture for different video sequences with different coding structure and QP. All the sequences with resolution of HD 1920x1080, 4:2:0 color format and frame rate of 30 fps are encoded by H.264 reference software JM 12.2. With the prediction-based

mechanism and SE merging method, the RTL simulation result shows that the proposed design can decode 1.71 bins per cycle in average with the drop in decoding speed between optima and actuality under 0.1 bins per cycle. Furthermore, for high bit-rate coding such as QP equaling to 12, the actual decoding speed almost reaches optimal decoding speed.

According to the maximum macroblock processing rate (MB/s) constrain of specified Level defined in the standard, the minimum working frequency requirement for different Level in listed in Table 34. The result shows that our proposed CABAC decoder can support Level 5.1real-time decoding.

The synthesis results of the proposed architecture and a performance comparison with previous works are shown in Table 33. By applying the mathematical transform method, the proposed architecture can efficiently reduce the critical path delay and allows the maximum working frequency to be about 264 MHz. The throughput of the proposed design is 451.4 Mbins/sec in average, which is superior to other existing designs. Although Lin’s design [13] can achieve higher average bin/cycle; however, it requires roughly two times area overhead when compared to our design. With the proposed hybrid CM memory architecture, the total gate count of our design is 42.37k, which achieves 48.6% hardware cost reduction in comparison to the all register based architecture.

TABLE 30. CABAC DECODING PERFORMANCE OF THE PROPOSED ARCHITECTURE WITH I CODING STRUCTURE

Video Sequence (I)	QP	Bitrate (Mbps)	Throughput (bin/s)	Decoding Cycle	Penalty	Optimal Decoding Speed (bin/cycle)	Actual Decoding Speed (bin/cycle)
Pedestrian_area	28	16.26	21,186,892	12,624,603	670,968	1.678	1.594
	20	49.82	66,633,779	36,914,531	876,602	1.805	1.763

	12	138.03	182,974,773	98,927,974	605,139	1.85	1.838
Riverbed	28	26.42	34,409,411	19,380,322	553,485	1.775	1.726
	20	70.29	94,251,652	51,595,049	781,238	1.827	1.8
	12	170.71	229,357,197	123,838,778	421,887	1.852	1.846
Rush_hour	28	8.78	11,213,016	6,741,175	318,614	1.663	1.588
	20	28.12	38,699,528	21,545,044	654,886	1.796	1.743
	12	111.16	151,595,001	81,372,224	679,297	1.863	1.848
Station2	28	19.7	24,381,140	13,942,414	496,822	1.749	1.689
	20	58.64	77,158,534	42,358,072	816,608	1.822	1.787
	12	152.41	203,271,631	109,701,718	451,180	1.853	1.845
Sunflower	28	17.12	21,086,788	12,373,604	634,349	1.704	1.621
	20	39.64	49,516,552	28,037,895	786,138	1.766	1.718
	12	109.78	148,622,187	80,223,469	433,019	1.853	1.843
Tractor	28	32.62	41,408,756	23,643,415	844,159	1.751	1.691
	20	80.45	104,694,644	57,634,937	522,491	1.817	1.8
	12	177.02	236,538,032	127,715,701	228,696	1.852	1.849
Average						1.79	1.76

TABLE 31. CABAC DECODING PERFORMANCE OF THE PROPOSED ARCHITECTURE WITH IPPP CODING STRUCTURE

Video Sequence (IPPP)	QP	Bitrate (Mbps)	Throughput (bin/s)	Decoding Cycle	Penalty	Optimal Decoding Speed (bin/cycle)	Actual Decoding Speed (bin/cycle)
Pedestrian_area	28	6.24	8,243,604	5,096,931	399,656	1.617	1.5
	20	29.54	39,925,740	22,531,263	1,021,723	1.772	1.695
	12	125.65	167,323,465	90,263,065	734,579	1.854	1.839
Riverbed	28	25.6	33,793,858	19,033,689	577,882	1.775	1.723
	20	69.18	93,001,166	50,914,635	815,699	1.827	1.798
	12	170.25	229,139,086	123,698,211	431,383	1.852	1.846
Rush_hour	28	4.23	5,759,310	3,571,634	266,172	1.613	1.501
	20	19.46	25,637,361	14,569,250	769,343	1.76	1.671
	12	108.29	148,073,162	79,495,164	777,936	1.863	1.845
Station2	28	2.97	4,323,558	2,776,874	229,535	1.557	1.438
	20	29.02	41,714,033	23,398,501	1,070,737	1.783	1.705
	12	131.7	176,574,564	95,089,541	602,285	1.857	1.845
Sunflower	28	2.9	3,913,585	2,522,353	230,268	1.552	1.422

	20	11.51	14,473,353	8,612,806	670,697	1.68	1.559
	12	87.76	116,421,680	63,687,024	1,082,948	1.828	1.797
Tractor	28	10.6	13,428,740	7,986,431	594,672	1.681	1.565
	20	52.77	70,588,463	38,960,452	1,173,367	1.812	1.759
	12	155.65	203,467,876	109,714,968	405,875	1.855	1.848
Average						1.75	1.69

TABLE 32. CABAC DECODING PERFORMANCE OF THE PROPOSED ARCHITECTURE WITH IBBBP CODING STRUCTURE

Video Sequence (IBBBP)	QP	Bitrate (Mbps)	Throughput (bin/s)	Decoding Cycle	Penalty	Optimal Decoding Speed (bin/cycle)	Actual Decoding Speed (bin/cycle)
Pedestrian_area	28	6.18	7,996,797	4,963,851	403,129	1.611	1.49
	20	26.48	33,865,563	19,296,701	957,390	1.755	1.672
	12	122.27	158,460,245	85,712,717	706,222	1.849	1.834
Riverbed	28	25.95	33,157,233	18,739,109	613,915	1.769	1.713
	20	69.51	90,186,496	49,414,413	799,758	1.825	1.796
	12	170.56	221,804,661	119,855,332	419,247	1.851	1.844
Rush_hour	28	3.96	5,359,183	3,361,522	262,862	1.594	1.479
	20	17.3	21,880,212	12,574,561	742,208	1.74	1.643
	12	106.46	141,397,927	76,048,399	799,083	1.859	1.84
Station2	28	3.09	4,487,035	2,896,824	223,772	1.549	1.438
	20	22.71	30,697,181	17,486,212	924,615	1.756	1.667
	12	124.92	164,313,863	88,365,185	565,193	1.859	1.848
Sunflower	28	2.81	3,747,747	2,425,104	230,656	1.545	1.411
	20	10.95	13,361,034	7,942,264	596,995	1.682	1.565
	12	81.97	106,090,796	58,346,381	1,085,765	1.818	1.785
Tractor	28	10.2	12,561,227	7,479,415	548,993	1.679	1.565
	20	51.34	65,822,524	36,557,868	1,087,536	1.801	1.748
	12	152.29	194,679,968	104,946,453	362,293	1.855	1.849
Average						1.74	1.68

TABLE 33. CABAC DECODER IMPLEMENTATION RESULT COMPARISONS OF DIFFERENT DESIGNS

Specifications	Proposed	Lin [13]	Chen [14]	Chang [17]
Technology	UMC 90nm	UMC 90nm	0.13um	TSMC 0.18um
Max. Frequency	264 MHz	222 MHz	238 MHz	250 MHz
Gate Count	42,372 ^b	82,445	43,600	35,615
Average bin/cycle	1.71	1.96	1.32	0.64
Throughput ^a (Mbins/sec)	451.4	435.1	314.2	160.0

a. Throughput = (maximum frequency) * (average bin/cycle)

b. Hybrid CM memory included

TABLE 34. WORKING FREQUENCY FOR DIFFERENT LEVEL CONDITIONS

Level	Max. MBs/frame	Max. MB Processing Rate (MBs/s)	Working Frequency
4	8192	245,760	44 MHz
4.1	8192	245,760	44 MHz
4.2	8704	522,240	92 MHz
5	22080	589,824	104 MHz
5.1	36864	983,040	173 MHz

Chapter 5 EXTENDING TOWARDS SVC

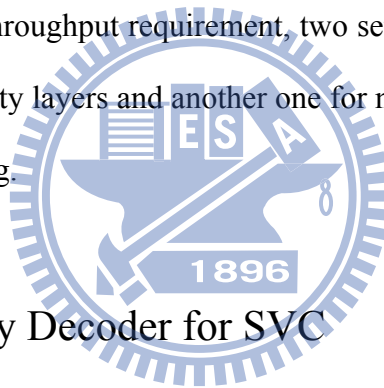
Recently, Scalable Video Coding (SVC), the next-generation video coding standard inherited from the H.264/AVC has been standardized [18]. It provides spatial scalability, temporal scalability, and quality scalability by transmitting a single bitstream containing subset bitstreams which can be transmitted and decoded partially depending on the transmission environments and decoding capabilities of endpoints such as video devices with different screen resolution and power limitation. Relative to the scalable profiles of prior video coding standards, the increased degree of scalability supported by SVC achieves significant improvements in coding efficiency and provides enhancement to transmission and storage applications. However, the throughput requirement for entropy decoder becomes stricter. As a result, further search for a suitable entropy decoder design for SVC is necessary.

5.1 Design Target and Design Challenges

The parsing procedure of SVC is more complex than of H.264/AVC, moreover, to support quality scalability, the two entropy decoding cores CAVLC decoder and CABAC decoder we design for H.264/AVC have to be modified. In SVC, two approaches are specified to provide SNR scalability: coarse-grain quality scalability (CGS) and medium-grain quality scalability (MGS). For CGS coding, quality refinement is achieved by applying different quantization parameter to quality enhancement layer, and the differences between transform coefficients are encoded in the slice data. In the definition of reference software, up to 7 CGS layers can be used for SNR scalability. In addition, as difference of QP (DQ) rises, the residual blocks

become denser. Even though the transform coefficients of quality enhancement layer are general small, the large amount still imposes a higher throughput requirement on entropy decoder for SVC than for H.264/AVC. With regard to MGS coding, the transform coefficients can be partition into up to 16 MGS layers to achieve finer granularity. However, the partition of transform coefficients changes the residual block structure. Therefore, additional look-up tables are introduced in CAVLC and VLC to maintain coding efficiency.

Our target is to develop a SVC entropy decoder which can support 3 spatial layers, maximum resolution 1920x1080, 3 temporal layers, maximum frame rate 60 fps, and 3 CGS quality layers real-time SVC decoding at working frequency 135 MHz. To conquer the barrier of throughput requirement, two sets of entropy decoder engine are employed, one for quality layers and another one for non-quality layers. The detail is presented in the following.



5.2 Proposed Entropy Decoder for SVC

Fig. 29 shows the system level architecture of proposed entropy decoder for SVC. Since the context-based adaptive modeling for entropy decoder is limited in a single slice, the two entropy decoding engines can work in parallel. To realize this architecture, we have to distinguish quality enhancement layer bitstream from non-quality enhancement layer bitstream. Fortunately, NAL units (slices) are separated by start code 0x00000001 in H.264. Therefore, we employ the bitstream scanner to quickly detect the start points of quality enhancement layers and non-quality enhancement layers and transmitted the addresses to the bitstream fetchers. Furthermore, to reduce the hardware cost overhead, a simplified CABAC decoder for quality enhancement layer is proposed. In quality enhancement layer, only

quality refinement information exists, while macroblock information is inherited from base layer. As a result, only `mb_skip_flag`, `coded_block_pattern`, `transform_size_8x8_flag`, `mb_qp_delta`, `coded_block_flag`, significance map, and `coeff_abs_level_minus1` SEs have to be decoded when decoding quality enhancement layer. Consequently, to satisfy the strict throughput requirement while maintaining low hardware cost, we propose a simplified CABAC decoder for decoding quality enhancement layer. As shown in Table 35 and Table 36, unused CMs are removed from the CM memory. The complete CM memory used for base layer is shown in Table 37 and Table 38. To further reduce the hardware cost, unnecessary storage of neighboring SEs used for context model selection is also removed. 120x99 bits memory space using for storing upper macroblock information such as `mvd` and `mb_type` can be saved. As to the CALVC decoder, since it is designed for decoding residual block information inherently, no simplification can be performed. Table 39 summarizes the synthesis results of proposed entropy decoder. It was synthesized with UMC 90nm technology with 135MHz. The simplified CABAC decoder can significantly save the memory area that 82.5% hardware cost reduction of memory is achieved in comparison to the original CABAC decoder.

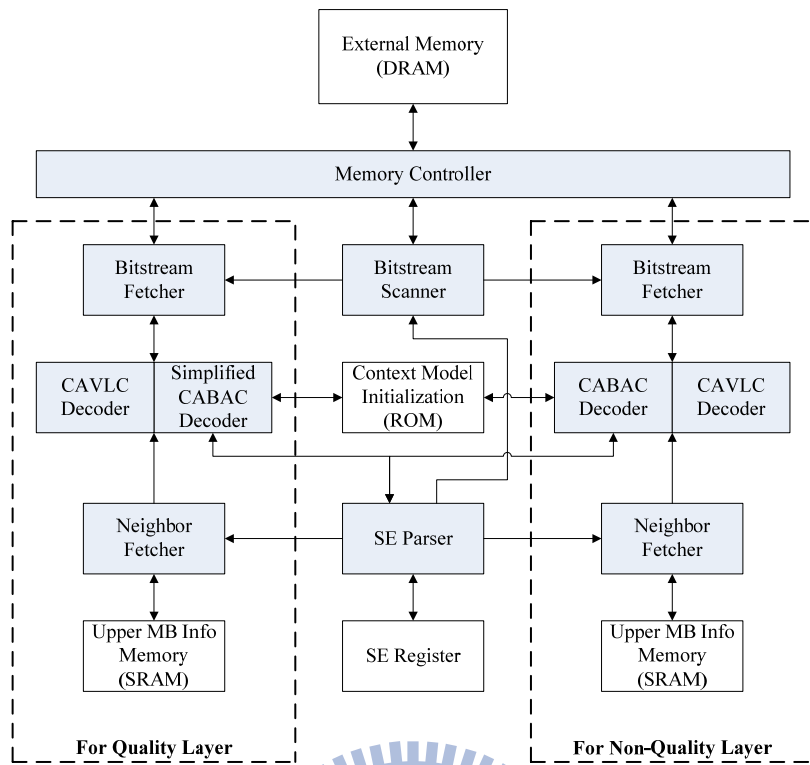


Figure 29. Framework of proposed entropy decoder for SVC.

TABLE 35. CONTENT OF SRAM FOR SVC QUALITY ENHANCEMENT LAYER

Address	CM Index	Syntax Element
0-2	11-13	mb_skip_flag (P/SP)
3-5	24-26	mb_skip_flag (B)
6-25	85-104	coded_block_flag
26-165	166-226, 338-398, 417-425, 451-459,	last_significant_coeff_flag
166-195	227-231, 237-241, 247-251, 257-261, 266-270, 426-430,	coeff_abs_level_minus1 (First bin)
196-198	399-401	transform_size_8x8_flag

TABLE 36. CONTENT OF REGISTER FOR SVC QUALITY ENHANCEMENT LAYER

Address	CM Index	Syntax Element
0-3	60-63	mb_qp_delta
4-15	73-84	coded_block_pattern
16-167	105-165, 277-337, 402-416, 436-450,	significant_coeff_flag
168-196	232-236, 242-246, 252-256, 262-265, 271-275, 431-435,	coeff_abs_level_minus1 (First bin excluded)

TABLE 37. CONTENT OF SRAM FOR SVC BASE LAYER

Address	CM Index	Syntax Element
0-2	0-2	mb_type (SI)
3-5	11-13	mb_skip_flag (P/SP)
6-8	24-26	mb_skip_flag (B)
9-11	70-72	mb_field_decoding_flag
12-31	85-104	coded_block_flag
32-171	166-226, 338-398, 417-425, 451-459,	last_significant_coeff_flag
172-201	227-231, 237-241, 247-251, 257-261, 266-270, 426-430,	coeff_abs_level_minus1 (First bin)
202-204	399-401	transform_size_8x8_flag
205-207	1024-1026	base_mode_flag
208	1027	mation_prediction_flag_10

209	1028	mation_prediction_flag_11
210	1029	residual_prediction_flag

TABLE 38. CONTENT OF REGISTER FOR SVC BASE LAYER

Address	CM Index	Syntax Element
0-7	3-10	mb_type (I)
8-14	14-20	mb_type (P/SP)
15-17	21-23	sub_mb_type (P/SP)
18-26	27-35	mb_type (B)
27-30	36-39	sub_mb_type (B)
31-44	40-53	Mvd
45-50	54-59	ref_idx
51-54	60-63	mb_qp_delta
55-58	64-67	intra_chroma_pred_mode
59	68	prev_intra_pred_mode_flag
60	69	rem_intra_pred_mode
61-72	73-84	coded_block_pattern
73-224	105-165, 277-337, 402-416, 436-450,	significant_coeff_flag
225-253	232-236, 242-246, 252-256, 262-265, 271-275, 431-435,	coeff_abs_level_minus1 (First bin excluded)

TABLE 39. SYNTHESIS RESULTS

Component	Working Frequency	Area: Logic Part (gate count)	Area: Memory Part (bits)
CAVLC Decoder	135 MHz	11,726	5,520
CABAC Decoder	135 MHz	37,885 ^a	14,400
Simplified CABAC Decoder	135 MHz	32,821 ^a	2,520

Neighbor Fetcher	135 MHz	27,723	W/O
Bitstream Scanner	135 MHz	9,248	W/O
Bitstream Fetcher	135 MHz	4,139	W/O
SE Parser	135 MHz	20,722 ^b	16,704

a. Hybrid CM memory included

b. SE Register included



Chapter 6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, to achieve high decoding performance and low hardware cost real-time entropy decoding systems, a high-throughput and fully hardwired entropy decoder for H.264/AVC is proposed. Our proposed entropy decoder architecture makes six main contributions:

- 1) Unlike previous multi-symbol CAVLC decoding architecture, which only accelerate the decoding procedure of *run_before* symbols, our proposed CAVLC decoder can further elevate the throughput by applying the delay balanced two-level decoding (DBTLD) architecture that can decode two *level* symbols in one cycle and shortens the critical path delay by 21% in comparison to the conventional approach of cascading two level decoders, and allows the maximum working frequency to be about 390 MHz.
- 2) To further accelerate decoding procedure, a skipping mechanism is proposed to remove redundant decoding processes and provide an early termination of current residual block decoding procedure. Moreover, in the CAVLC decoding procedure, since only one of *coeff_token*, *trailing_ones_sing_flag*, *level*, *total_zeros*, and *run_before* decoding units is assigned to work in each cycle, idled units are turned off by functional gating to reduce power consumption.
- 3) A fully hardwired CABAC decoder design which combines SE parsing with decoding is proposed. By taking advantage of the characteristics of SE parsing flow and bin distribution among SEs, we design a prediction-based

pipelined architecture to accelerate the CABAC decoding procedure without stall for most case. The prediction hit rate can achieve 96.78% in average and over 99% in high bit-rate coding.

- 4) Our proposed hybrid CM memory architecture not only avoids structural hazards caused by CM reading and writing but also reduces the hardware cost overhead significantly by 48.6% in comparison to the implementation of all register approach.
- 5) With the proposed mathematical transform method, the critical path delay of TSBAD engine is efficiently improved by 28% compared with the traditional TSBAD engine, and allows the maximum working frequency to be about 264 MHz. The throughput of the proposed CABAC decoder can achieve 451.4 Mbins/sec in average.
- 6) We extend our entropy decoder towards SVC extension of H.264/AVC. At the working frequency 135 MHz, our proposed entropy decoder can support 3 spatial layers, maximum resolution 1920x1080, 3 temporal layers, maximum frame rate 60 fps, and 3 CGS quality layers real-time SVC decoding.

6.2 Future Work

High Efficiency Video Coding (HEVC), so-called H.265 is currently under development by Joint Collaborative Team on Video Coding (JCT-VC) of MPEG and VCEG. As a successor to H.264/AVC, HEVC is targeted at next-generation HDTV displays with Super Hi-Vision and aims to reduce bit-rate requirement by half in comparison to H.264/AVC. However the improved coding efficiency usually accompanies with the expense of increased computational complexity. As a result, to

achieve real-time coding system, further search for a hardware-friendly entropy coding algorithm is necessary.



Reference

- [1] “Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec.H.264 jISO/IEC 14496-10 AVC),” in Joint Video Team, Mar. 2003, Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVT-G050.
- [2] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, “Overview of the H.264/AVC Video Coding Standard,” *IEEE Trans. Circuits Syst. Video Technol.*, vol 13, no. 7, pp. 560-576, Jul. 2003.
- [3] D. Marpe, H. Schwarz, and T. Wiegand, “Context-Based adaptive binary arithmetic coding in the H.264/AVC video compression standard,” *IEEE Trans. Circuits Syst. Video Technol.*, vol 13, no. 7, pp. 620-636, Jul. 2003.
- [4] H.-Y. Lin, Y.-H. Lu, B.-D. Liu, and J.-F. Yang, “A Highly Efficient VLSI Architecture for H.264/AVC CAVLC Decoder,” *IEEE Trans. Multimedia*, vol 10, no. 1, pp. 31-42, Jan. 2008.
- [5] M. Alle, J. Biswas, and S. K. Namdy, “High Performance VLSI Architecture Design for H.264 CAVLC Decoder,” in *Proc. IEEE 17th Int. Conf. Application-Specific Systems, Architectures Processors*, Steam-boat Springs, CO, Sep. 2006, pp. 317-322.
- [6] T.-L Fang, “Architecture Design of CAVLC Decoder with Low Power and High Throughput Consideration,” M.S. thesis, Department of Electrical Engineering, National Central University, Jul. 2008.
- [7] G.-S. Yu and T.-S. Chang, “A Zero-Skipping Multi-symbol CAVLC Decoder for MPEG-4 AVC/H.264,” in *Proc. Int. Symp. Circuits Syst.*, Island of Kos, Greece, May 2006, pp. 5583-5586.

- [8] Y.-N. Wen, G.-L. Wu, S.-J. Chen, and Y.-H. Hu, "Multiple-Symbol Parallel CAVLC Decoder for H.264/AVC," in *Proc. 2006 IEEE Asia Pacific Conf. Circuit Syst.*, Singapore, Dec. 2006, pp. 1240-1243.
- [9] G.-G. Lee, C.-C. Lo, Y.-C. Chen, H.-Y. Lin, and M.-J. Wang, "Low Complexity and High Throughput VLSI Architecture for AVC/H.264 CAVLC Decoding," *IET Image Processing*, to be published.
- [10] S.-Y. Tseng, and T.-W. Hsieh, "A Pattern-Search Method for H.264/AVC CAVLC Decoding," in *Proc. 2006 IEEE Int. Conf. Multimedia Expo*, Toronto, ON, Canada, Jul. 2006, pp. 1073-1076.
- [11] Y. Yi and I. C. Park, "High-Speed H.264/AVC CABAC decoding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 4, pp. 490-494, Apr. 2007.
- [12] W. Son and I. C. Park, "Prediction-based Real-time CABAC Decoder for High Definition H.264/AVC," in *Proc. Int. Symp. Circuits Syst.*, Seattle, WA, May 2008, pp. 33-36.
- [13] P.-C. Lin, T.-D. Chuang, and L.-G. Chen, "A branch selection multi-symbol high throughput CABAC decoder architecture for H.264/AVC," in *Proc. Int. Symp. Circuits Syst.*, Taipei, May 2009, pp. 365-368.
- [14] J.-W. Chen, and Y.-L. Lin, "A High-performance Hardwired CABAC Decoder for Ultra-high Resolution Video," *IEEE Trans. Consum. Electron.*, vol. 55, no. 3, pp. 1614-1622, Aug. 2009.
- [15] P. Zhang, "Fast CABAC decoding architecture," *ELECTRONICS LETTERS*, vol. 44, no. 24, Nov. 2008.
- [16] C. H. Kim and I. C. Park, "High Speed Decoding of Context-based Adaptive Binary Arithmetic Codes Using Most Probable Symbol Prediction," in *Proc. IEEE*

ISCAS, Island of Kos, Greece, May 2006, pp. 1707-1710.

[17] Y.-T. Chang, "A novel pipeline architecture for H.264/AVC CABAC decoder," in *Proc. 2008 IEEE Asia Pacific Conf. Circuit Syst.*, Dec. 2008, pp. 308–311.

[18] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 9, pp. 1103-1120, Sep. 2007.



Biographical Notes

姓名：廖元歆

學歷：

高雄市立高雄高級中學 (2001/09 – 2004/06)

國立交通大學電機資訊學士班 (2004/09 – 2008/06)

國立交通大學電子研究所系統組 (2008/09 – 2010/08)

著作：

Yuan-Hsin Liao, Gwo-Long Li, and Tian-Sheuan Chang, “A High Throughput VLSI Design with Hybrid Memory Architecture for H.264/AVC CABAC Decoder,” *in proceeding of IEEE International Symposium on Circuit and System*, pp. 2007-2010, May 2010.

