Architecture and Hardware for a 1 Bin per Cycle Context-Adaptive Binary Arithmetic Coder (CABAC) Encoder

By

RENJIE CHEN

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan M. Baas

Member, Dr. Rajeevan Amirtharajah

Member, Dr. Soheil Ghiasi

Committee in charge
2019

Abstract

Entropy encoding is a key element in H.264/MPEG-4 AVC video coding responsible for the lossless compression of transformed and quantized data. Context-Adaptive Binary Arithmetic Coding (CABAC) is one of the two entropy coding methods available in the H.264 standard. CABAC achieves its high compression ratios by combining arithmetic coding with adaptive context modeling. Compared to the other entropy coding method which is called Context-Adaptive Variable-Length Coding (CAVLC), CABAC achieves a 15% to 19% greater bit-rate reduction. However, the computational complexity of CABAC is much greater than the complexity of CAVLC, so hardware acceleration for CABAC is necessary for real-time high resolution video coding.

This thesis presents a hardware CABAC encoder implementation for the H.264 main profile. The implementation supports all CABAC functions including context initialization, binarization, context modeling and binary arithmetic coding in hardware. The architecture of the CABAC encoder consists of six pipeline stages and is capable of encoding at a rate of 1 bin per cycle where a bin is a binarized symbol encoded in arithmetic coding. In functional simulation, the output generated from the presented CABAC encoder are perfectly matched with the output of the CABAC module in H.264 JM reference software. The CABAC encoder was implemented in a 28 nm FD-SOI CMOS technology and it achieves a throughput of 769 million bins per second which is sufficient to encode real-time 4K UHD (3840×2160) video at 60 frames per second while dissipating an average power of 11.48 mW. The circuit area is 33,411 μm^2 which is a logic gate count of 23.4K equivalent minimum NAND gates.

Acknowledgments

First and foremost, I would like to take this chance to thank Professor Bevan Baas for all of his help, time and guidance throughout my research and graduate career. His insights and advise on this research inspires me exploring widely. With his support, I overcame one after another difficulty throughout the work and eventually make it toward this thesis. He has helped me grow into more skillful and productive researcher in field of VLSI design and signal processing. He has taught me to aim high and stick to my goals. All of this will be beneficial throughout my career and life.

I would also like to thank Professor Soheil Ghiasi and Professor Rajeevan Amirtharajah. Thank your for serving as my thesis committee member and taking time to review the thesis and give me valuable feedback.

I would like to give my special thanks to Christi Tain for her help in physical design of the proposed CABAC encoder and her valuable feedback for my thesis writing. I would also like to thank Shifu Wu for giving me many valuable suggestions in RTL design. I would like to thank Jin Cui and Peiyao Shi for supporting me in my research and also in my life as roommates. I would like to thank Sharmila Kulkarni, Timothy Andreas, Satyabrata Sarangi, Brent Bohnenstiehl and all other members of VLSI Computation Laboratory from whom I receive valuable feedback.

I am especially grateful for my family. Thank you for your encouragement and support throughout all this time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The increasing demand for high definition (HD) and even ultra high definition (UHD) video in real-time broadcasting and streaming via network makes high performance video compression technology important nowadays. The H.264 Advanced Video Coding (AVC) becomes the most commonly used video compression technique standard in recent years due to its high performance in data compression and small loss in video quality. In H.264, the Context-Adaptive Binary Arithmetic Coding (CABAC) is one option for the entropy coding, which is able to reduce the data redundancy significantly based on its statistical property. Compared to the H.264 baseline entropy coding, known as Context-Adaptive Variable-Length Coding (CAVLC), CABAC achieves significantly higher compression ratio especially for video with high resolution. Thus, CABAC is also chosen as the only entropy coding technique for next generation High Efficiency Video Coding (HEVC) standard.

Due to the high computational complexity, hardware acceleration for CABAC is important. Fully hardwired CABAC engine can be utilized as an IP core in System on Chip (SoC) or as an Application Specific Integrated Circuit (ASIC) accelerator in H.264 application system. However, most existing implementation of CABAC are software hardware co-designed, whose processing speed is restrained by the software. The goal of this research is to develop an high-throughput, area-efficient, full-hardwired CABAC encoder for main profile of H.264 standard.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 starts with the background of H.264. A brief introduction of entropy coding concept is given next. Then, syntax element, the data to be encoded in CABAC, is explained by its category. After that, the framework of CABAC encoder is presented. Lastly, a collection of previous existing work of CABAC encoder is introduced with their contribution and trade-offs.

Chapter 3 presents the proposed hardware implementation of the CABAC encoder in detail. Firstly, an overview of the pipelined CABAC encoder architecture is given. Then, the design and implementation of each part in the architecture is elaborated.

Chapter 4 first introduces the H.264 JM reference software which is used as reference for functional verification. The CABAC related configuration of JM encoder is explained. Then the detailed methodology, test bench design and test vector extraction in verification process are elaborated.

Chapter 5 presents the synthesis and physical design result for the proposed CABAC implementation. Then analysis of throughput, circuit area and power is given. Lastly, comparison with previous existing work is presented.

Finally, Chapter 6 gives an overall summary of the thesis and proposes future objectives of this research.

Chapter 2

Overview of H.264 Standard and CABAC

2.1 Background on H.264

2.1.1 Overview of H.264

In digital video compression codec, the encoder encodes the raw video sequence into compressed data file for efficient data storing and transferring through network, while the decoder extracts the video sequence from the compressed file for video playback. Nowadays, the data size of the video sequence has grown significantly. The video that has 1080p (1920×1080) or even 4K (3840×2160) resolution is quite commonly used in consumer media streaming. Therefore, high compression ratio with minimum quality loss is sought for video compression technology.

The H.264/AVC standard provides encoding techniques that has high compression ratio with great graphic quality. The H.264 standard is developed jointly by Moving Picture Experts Group (MPEG) and Video Coding Experts Group (VCEG). The standard is described by the International Telecommunication Union Telecommunication (ITU-T) H.264 standard documentation [1]. The H.264 standard is also known as the Part 10 of the MPEG-4.

The coding path of H.264 encoder is shown in Fig. 2.1. The encoding process of H.264 consists of four main stages: prediction, transform, quantization and entropy coding. First, in the prediction procedure, predicted signals of one video picture is generated by either intra or inter prediction. In intra prediction, the predicted signal is produced based only on previous coded

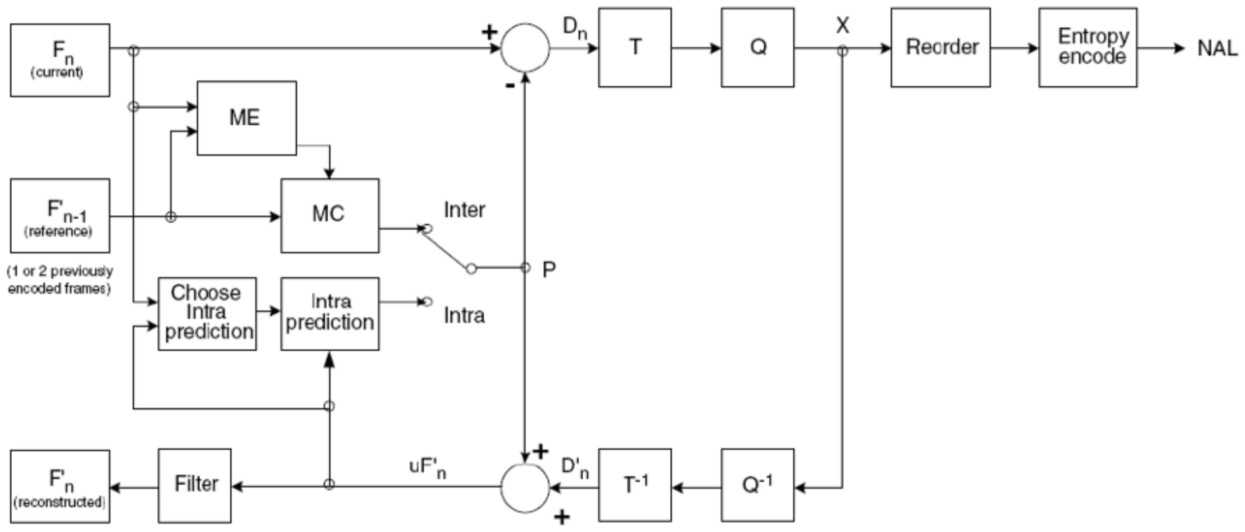


Figure 2.1: Coding path of H.264 encoder [2]

samples from the same picture. In inter prediction which is also known as motion estimation, the prediction is completed by referring to samples in previous coded pictures and following pictures in the video sequence. After prediction, the difference of predicted signal and original picture, known as the residual data, is passed to the transform unit. By integer Discrete Cosine Transforming (DCT), the samples in the residual data are transformed into coefficients. Then, the coefficients are quantized. Lastly, the quantized coefficients are transferred to entropy encoder, in which the coefficients together with parameters of prediction, transform and quantization, are coded into bit stream. In addition, there is a feedback path that reconstructs the picture by dequantizing and inversely transforming the quantized coefficients. The reconstructed picture is utilized as reference in prediction process.

The H.264 standard includes seven profiles that specify sets of capabilities for H.264 codec targeting specific applications. Baseline profile aims low-cost applications with limited computing resources, and only basic H.264 coding function is supported in baseline profile. Main profile is intended as the mainstream consumer profile for broadcast and storage applications. Extended profile is intended as the streaming video profile, with relatively higher compression ratio and extra support for robustness. The three major profiles in H.264 with their specific supported functional features are shown in Fig. 2.2. Other profiles proposed in H.264 standard includes high profile, high 10 profile, high 4:2:2 profile and high 4:4:4 profile. Besides profiles, the levels in H.264 standard ranging from 1.0 to 5.1 define sets of constraints which indicate coding performance requirements of

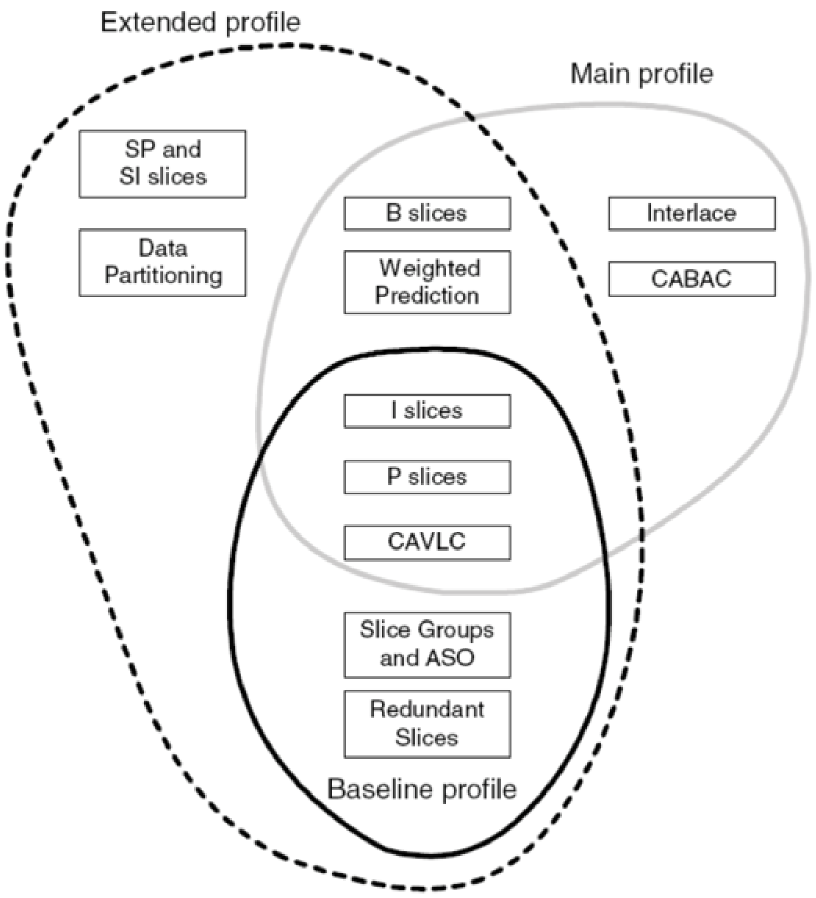


Figure 2.2: Major profiles in H.264 standard [2]

a profile, such as maximum bit-rate, frame rate, resolution, etc.

2.1.2 General Video Coding Concepts in H.264

Pixel Sampling

A digital video is composed of a sequence of pictures. Each picture consists of series of sampling points which are also called pixels. The traditional video sampling uses the red, green and blue (RGB) color space, where every pixel is represented by three numbers indicating the proportion of red, green and blue color in the pixel, respectively. Instead of RGB, a more efficient scheme of pixel sampling based on the luminance and color difference (chrominance), known as YCbCr (or YUV), is used in H.264. In the YCbCr scheme, one luminance component Y, and two chrominance components Cb and Cr are used for representing a pixel. Y, Cb and Cr can be derived from traditional RGB sampling components by equation [2] as follow:

$$\begin{aligned} Y &= 0.299 \times R + 0.587 \times G + 0.114 \times B \\ Cb &= 0.564 \times (B - Y) \\ Cr &= 0.713 \times (R - Y) \end{aligned} \tag{2.1}$$

Because human visual system is more sensitive to luminance than chrominance [2], the YCbCr sampling can be optimized by reducing the number of chrominance components in representing a pixel. The reduction of resolution in chrominance is called chroma sub-sampling. In H.264 baseline, main and extended profile, a popular 4:2:0 chroma sub-sampling scheme is adopted, in which every four pixels are sampled to four luminance components Y and two chrominance components (one Cb and one Cr). The 4:2:0 scheme is illustrated in Fig. 2.3.

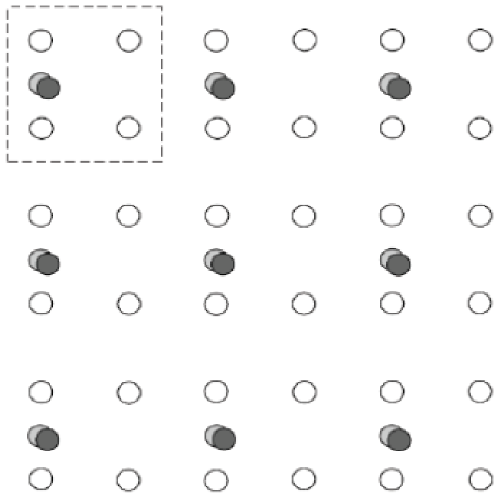


Figure 2.3: 4:2:0 chroma sub-sampling [2]

Some other chroma sub-sampling schemes are supported in high profile of H.264. For example, 4:2:2 chroma sub-sampling scheme includes four Y components, two Cr and two Cb for every four pixels. In 4:4:4 scheme, no chroma is sub-sampled, and every pixel is represented by one Y, one Cb and one Cr.

Group of Pictures

In digital video sequence, one picture is also called a frame. In H.264, five different types of frames are defined in the coded video sequence: I frame, B frame, P frame, SI frame and SP frame. I frame is the intra prediction frame. P frame is the prediction frame generated by inter prediction, it uses one or more previous coded I frames as the prediction reference. B frame is called Bi-direction prediction frame which is inserted between I frame and P frame or between two P frames. B frame is produced by referencing both previous coded I/P frames and following P frames in inter prediction. SI and SP are special frames for bit-rate adaptive control and error resilience, which are supported only in extended profile of H.264. According to [3], I-B-P-B is a typical sequence order of coded frames in H.264. Sequence like I-B-P-B is also called a group of picture (GOP). Each GOP contains one I frame at the beginning followed by several P frames and B frames. The number of P frames followed by I frame and number of B frames inserted can be configured in a GOP. In general, more B and P frames included in a GOP leads to higher bit-rate reduction but also higher computational workload in prediction process.

Macroblock Partition

In H.264 standard, a frame can be divided into one or more sections called slice. Slice also has five types according to type of the frame it belongs to. Furthermore, H.264 codec is a block-based system, in which one slice is composed of square blocks called macroblock. Each macroblock contains information for 16×16 (256) pixels.

Based on the 4:2:0 YUV sampling scheme applied in H.264, the luma (luminance) and chroma (chrominance) information contained in each 16x16 macroblock forms one 16×16 luma block, one 8×8 Cb block and one 8×8 Cr block, respectively. In intra prediction process, they can be partitioned to 4×4 luma blocks and 4×4 chroma blocks, respectively. The overall partitioning from video sequence to luma and chroma blocks in H.264 is shown in Fig. 2.4.

In the process of motion estimation (inter prediction), the 16×16 macroblock can be partitioned into 16×8, 8×16, 8×8 partitions. A 8×8 partition is also called a sub-macroblock which can be further divided into 8×4, 4×8, 4×4 sub-partitions. Different partition schemes in motion estimation is illustrated in Fig. 2.5.

Interlaced Sampling

Sampling a video sequence to a series of complete frames is called progressive sampling. Alternatively, a video can also be sampled to a series of field, which is known as interlaced sampling.

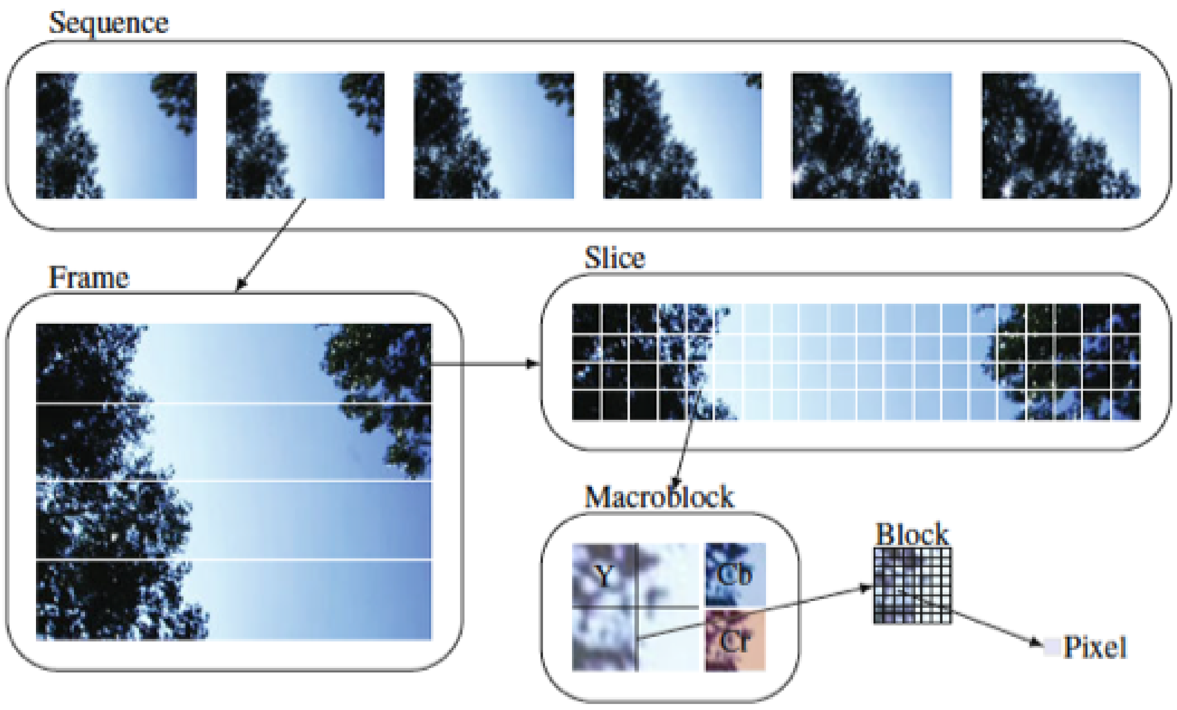


Figure 2.4: Video element partitioning from sequence to macroblock [3]

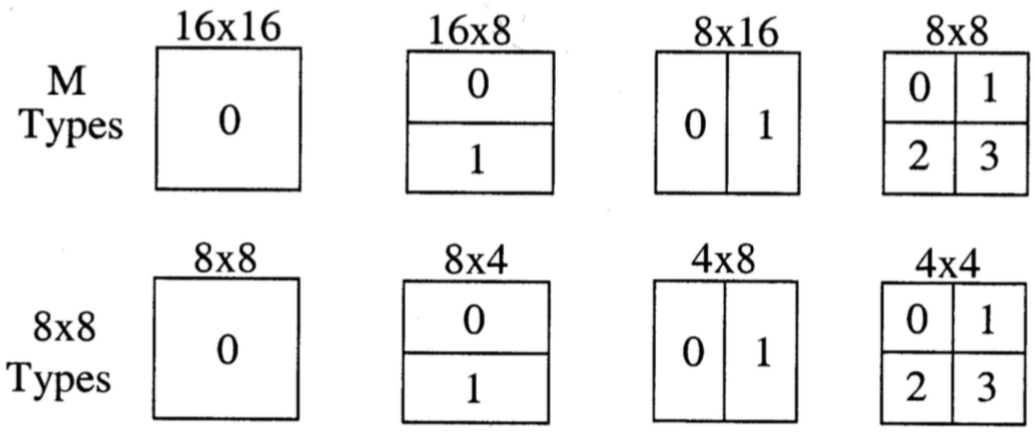


Figure 2.5: Macroblock partition for motion estimation in H.264 [4]

In interlaced sampling, one picture in the video sequence is sampled to one top field and one bottom field. As illustrated in Fig. 2.6, the top field contains macroblocks in odd-numbered rows while the bottom field contains macroblocks in even-numbered rows. To be sampled in interlaced mode, the height of picture in video sequence must be multiple of 32 (double height of a macroblock).

To be capable of encoding both frame and field sequence, two coding mode is applied in H.264 standard: field coding and frame coding. In H.264 standard, the decision of field/frame

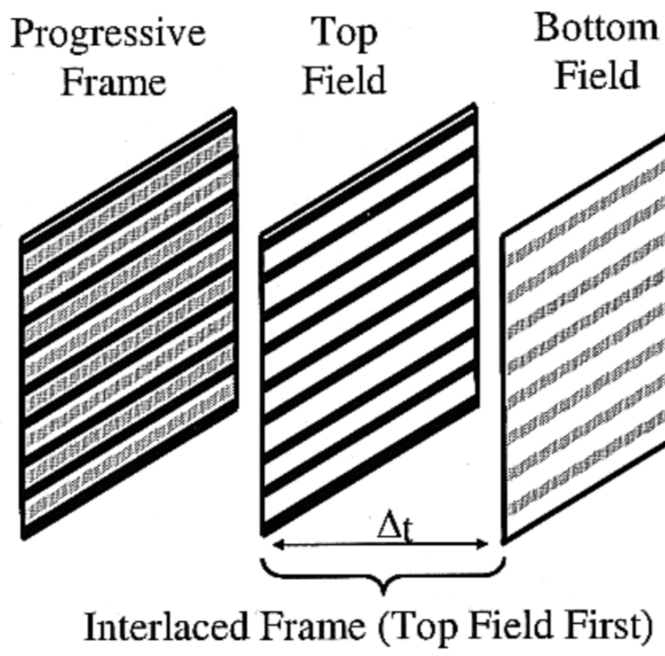


Figure 2.6: Progressive and interlaced frames and fields in H.264 [4]

coding mode can be fixed for the whole sequence or adaptive for each picture in the sequence. The picture-level adaptive coding is referred to as picture-adaptive frame/field (PAFF) coding. Furthermore, macroblock-adaptive frame/field (MBAFF) coding is also supported in H.264 standard. In MBAFF coding mode, the top and bottom fields are combined as a frame in which every pair of two vertically adjacent macroblocks (one in the top field and one in the bottom field) is coded in either frame coding mode or field coding mode. The macroblock pairs processed in field and frame coding mode are referred to as field macroblock pair and frame macroblock pair, respectively.

2.2 Background on Entropy Coding

Entropy coding is a lossless compression technique which reduces the redundancy of the input data based on its statistical property. Commonly used entropy coding techniques includes Run Length Coding (RLC), Huffman coding and arithmetic coding. In the standard of H.264, CAVLC is used for baseline profile, while CABAC is applied for main profile, extended profile and high profile.

The CAVLC is adopted for encoding the zig-zag order transformed residual coefficients, as well as the prediction modes of intra prediction and the motion vectors of inter prediction. In CAVLC, Huffman coding is applied for coding transform coefficients while Exponential Golomb coding is used for encoding the prediction modes and motion vectors [5].

In CABAC, a special scheme of binary arithmetic coding is utilized for encoding the semantics carried by syntax elements (SE) from previous H.264 coding procedures. The semantics carried by syntax elements include the type of macroblock, prediction modes in intra prediction, reference index and motion vector difference in inter prediction, parameters for quantization, residual data parameters and coefficients. CABAC is capable of encoding both binary and non-binary syntax elements. Non-binary syntax element is binarized before being coded in arithmetic coding. In the binary arithmetic coding process, a probability model, known as the context model, is adaptively selected based on local coding context. The coding context includes previous coded information in current coding macroblock, as well as the information of neighboring macroblocks. The adaptive selection of context model allows more accurate probability modeling in CABAC compared to conventional arithmetic coding scheme. As a result, high compression performance can be achieved in CABAC. Furthermore, CABAC is a multiplication-free coding system. Instead of using multiplication, the interval division and probability updating process in arithmetic coding is implemented by table looking up algorithm based on quantized probability states and quantized interval range. Therefore, the computation in CABAC is accelerated compared to conventional multiplication-based arithmetic coding. The detailed coding process of CABAC is elaborated in section 2.4.

According to the coding performance evaluation in [6], CABAC achieves average of 15% to 19% bit rate reduction over CAVLC in test of different format video sequences. The bit rate reduction over CAVLC is more significant when the definition of the video sequence is higher. Therefore, CABAC is a promising entropy coding option when considering the increasing demand for high resolution video nowadays.

2.3 Data to Process in CABAC: Syntax Element

In CABAC coding process, there are total 18 different types of syntax elements which can be divided into 5 categories based on the carried semantics. The categories of syntax element are listed in Table 2.1. The semantics specified by each syntax element is explained in this section.

mb_type: specifies macroblock type. The macroblock type in I slice is specified based on the partition scheme of the macroblock in intra prediction. The macroblock type in B slice and P

Table 2.1: Category of syntax element

Category	Syntax Element
Macroblock Type	mb_type
	sub_mb_type
Inter Prediction	mvd_lX
	ref_idx_lX
Intra Prediction	intra_chroma_pred_mode
	prev_intra4x4_pred_mode_flag
	rem_intra4x4_pred_mode
Residual Data	coded_block_pattern
	coded_block_flag
	significant_coeff_flag
	last_significant_coeff_flag
	coeff_abs_level_minus1
	coeff_sign_flag
Control Flag and Parameter	mb_qp_delta
	mb_field_coding_flag
	mb_skip_flag
	end_of_slice_flag

slice is based on both partition scheme and inter prediction mode of this macroblock.

sub_mb_type: specifies sub-macroblock type. Sub-macroblock is only used in B slice and P slice. The sub-macroblock type is based on both partition scheme and inter prediction mode of this sub-macroblock.

mvd_lX [mbPartIdx][subMbPartIdx][compIdx]: motion vector difference (MVD) is the difference between the motion vector component and its prediction in the process of motion estimation. X in mvd_lX can be either 0 or 1, which designates the reference list used in prediction. list 0 is used for backward prediction while list 1 is used for forward prediction. mbPartIdx and subMbPartIdx specifies the index of macroblock partition and sub-macroblock partition, respectively. compIdx is the motion vector component index. It is assigned to 0 for horizontal vertor, and 1 for vertical vertor. The horizontal and vertical MVD are considered as two different types of syntax element.

ref_idx_lX[mbPartIdx]: specifies the index of reference picture in the reference list for motion estimation. X in Ref_idx_lX is same as the X in MVD. mbPartIdx specifies the index of macroblock partition.

intra_chroma_pred_mode (ICPM): specifies the intra prediction mode for chroma information in a macroblock.

prev_intra4x4_pred_mode_flag and **rem_intra4x4_pred_mode**: specifies intra4×4-prediction mode for each 4×4 luma block. prev_intra4x4_pred_mode_flag is set equal to 0 when there is no rem_intra4x4_pred_mode syntax element in this macroblock.

coded_block_pattern (CBP): specifies which of the four 8x8 luma blocks and two 8x8 chroma blocks contains non-zero transform coefficients. The four bits for luma blocks in CBP are referred to as CBP-Luma while the two bits for chroma blocks in CBP are referred to as CBP-Chroma.

coded_block_flag (CBF): is set equal to 0 when a transform block contains no non-zero transform coefficients. It is set equal to 1 when the block contains at least one non-zero transform coefficients.

significant_coeff_flag[scanningPos] (SCF): equals 0 when the transform coefficient level at current scanning position is equal to zero. It is set equal to 1 when this position has non-zero level value.

last_significant_coeff_flag[scanningPos] (LSCF): is equal to 1 when the following scanning positions within this transform block has all zero values. It is equal to 0 when there is at least one non-zero value in the following positions.

coeff_abs_level_minus1[scanningPos]: is the absolute value of the transform coefficient level value minus one when the this position has non-zero value.

coeff_sign_flag[scanningPos]: specifies the sign of the transform coefficient level value at this position.

mb_qp_delta (QPD): specifies the difference between the QP used in current macroblock and previous macroblock. The QPD of the first macroblock in each slice specifies the difference between the first macroblock QP and the slice QP.

mb_field_coding_flag: is set equal to 0 when current macroblock pair is a frame coding macroblock pair, and 1 when it is field coding macroblock pair.

mb_skip_flag: specifies if current macroblock is skipped.

end_of_slice_flag: is always the final syntax element within one macroblock. It equals 0 when this macroblock is not the final macroblock in a slice. It equals 1 when it is the final

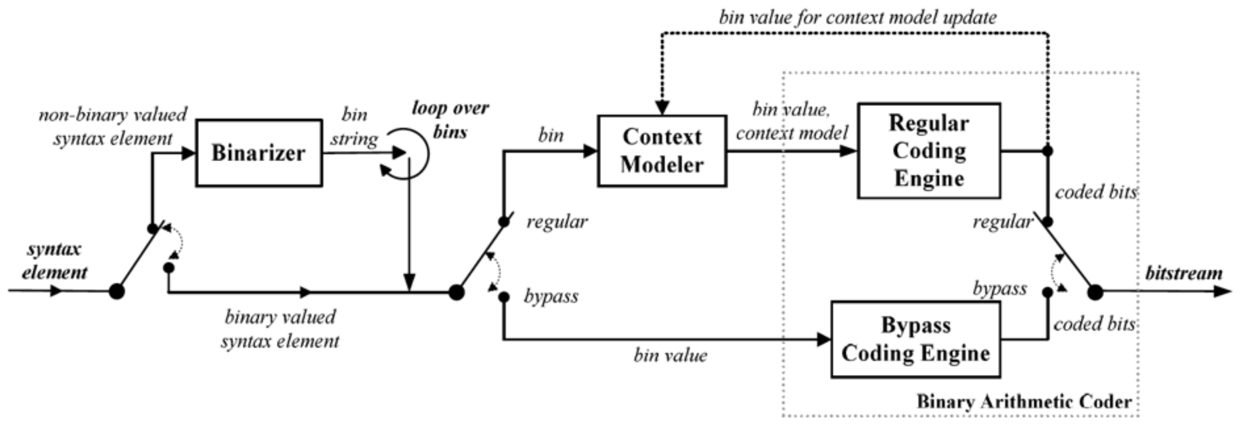


Figure 2.7: Coding path of CABAC encoder [7]

macroblock of a slice.

2.4 Overview of CABAC

The CABAC coding has three elementary procedures: binarization, context modeling (CM) and binary arithmetic encoding (BAE). The coding path of CABAC encoder is illustrated in Fig. 2.7

Binarization is a data pre-processing procedure for BAE. After binarization, non-binary input syntax elements are coded into a string of binary symbols. The binarized symbol, referred to as Bin, is encoded in the following binary arithmetic coding. In total, five different binarization coding techniques are applied in CABAC: Table mapping, unary coding, truncated unary (TU) coding, fixed-length (FL) coding and unary exponential golomb kth-order (UEGk) coding.

In context modeling procedure, a context model is selected for each bin from previous procedure by specifying a context model index (ctxIdx). For main profile H.264 standard, total 399 context models are used in context modeling. The context model index is calculated by adding up an offset (ctxIdxOffset) and an increment (ctxIdxInc) as shown in Eq. 2.2. For residual data syntax elements, an additional offset is specified by the context block category (ctxBlockCat) for different types of transform block.

$$ctxIdx = \begin{cases} ctxIdxOffset + ctxCatOffset[ctxBlockCat] + ctxIdxInc & \text{, Residual data SEs} \\ ctxIdxOffset + ctxIdxInc & \text{, Other type of SEs} \end{cases} \tag{2.2}$$

Basically, the ctxIdxOffset is determined by the type of syntax element, while the ctxIdxInc is corresponding to each bin of the syntax element after binarization. Derivation of the ctxIdxInc

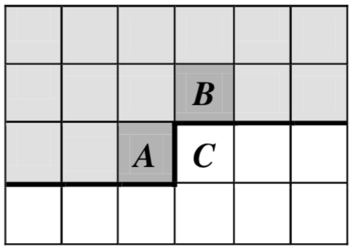


Figure 2.8: Reference syntax element in upper and left neighboring macroblocks [7]

refers to index of the bin in binstring (binIdx). For syntax element mb_type, the calculation of ctxIdxInc is also based on previous coded bin value in the binstring. For residual data related syntax element, the derivation of ctxIdxInc also depends on the accumulated residual data information such as the scanning position and the number of coefficients levels that equal or greater than one. For some other types of syntax element, the derivation of ctxIdxInc for their first bin is based on information of the upper and left neighboring macroblocks or partitions. The two neighboring macroblocks to be referenced are illustrated in Fig. 2.8. In general, for these types of syntax element, two flag signals are calculated according to the syntax element information of the neighboring macroblocks or partitions. Then the ctxIdxInc is computed from these two flags as shown in Eq. 2.3. For different syntax element types, different number of context models are allocated for the coding bin. Hence, different ways of calculating the ctxIdxInc is used. In the first two derivation shown in Eq. 2.3, the ctxIdxInc ranges from 0 to 2 and 0 to 3, respectively. Therefore, 3 and 4 context models can be specified by the ctxIdxInc, respectively. The last derivation in Eq. 2.3 is applied for the two bins of syntax element CBP-Chroma. As the ctxIdxInc ranges from 0 to 7, four context models are allocated for each bin of CBP-Chroma.

$$ctxIdxInc = \begin{cases} FlagA + FlagB \\ FlagA + 2 * FlagB \\ FlagA + 2 * FlagB + ((binIdx == 1)?4:0) \end{cases} \tag{2.3}$$

After context modeling, the bin together with the ctxIdx is passed to the last binary arithmetic coding procedure. In this procedure, the input bin is either symbol 0 or symbol 1. Among all the coded bins, the symbol has higher probability is called most probable symbol (MPS), and the other symbol is called least probable symbol (LPS). In the context model selected by ctxIdx, the value of MPS together with the probability of LPS among all coded bins (P_{LPS}) for the

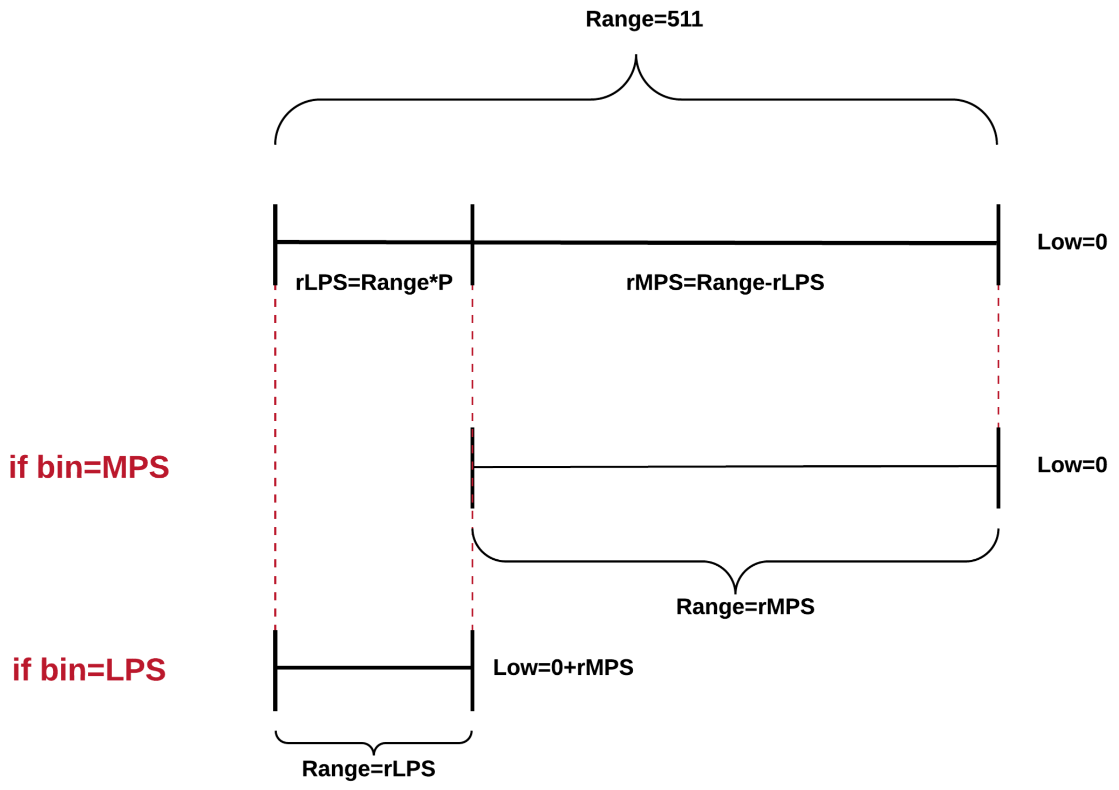


Figure 2.9: Updating of interval Range and Low in binary arithmetic coding

corresponding coding context is records and updated. Therefore, in the 399 context models, 399 pairs of MPS value and P_{LPS} are recorded for different coding context. Usually a memory, referred as to context memory (or context table), is applied for storing the 399 context models in CABAC implementation.

In arithmetic coding, a coding interval is setup and updated based on the probability of MPS and LPS. The code word of arithmetic coding is generated from recursively dividing the interval. As shown in Fig. 2.9, Low represents the end point of the interval while Range specifies the length of the interval. The initial value of Range and Low are 511 and 0, respectively. rMPS and rLPS represents the two corresponding sub-intervals of MPS and LPS, respectively. The interval division process is also illustrated in Fig. 2.9. If input bin is equal to MPS, rMPS is chosen as the new interval, otherwise rLPS is selected. Furthermore, valid value range of the Range is from 256 to 511, inclusive. When the updated Range value is not in the valid range, a renormalization process is invoked. Majority part of code word bits are generated during the renormalization process, and the remainder of code word is generated from the final coding interval in terminate coding process.

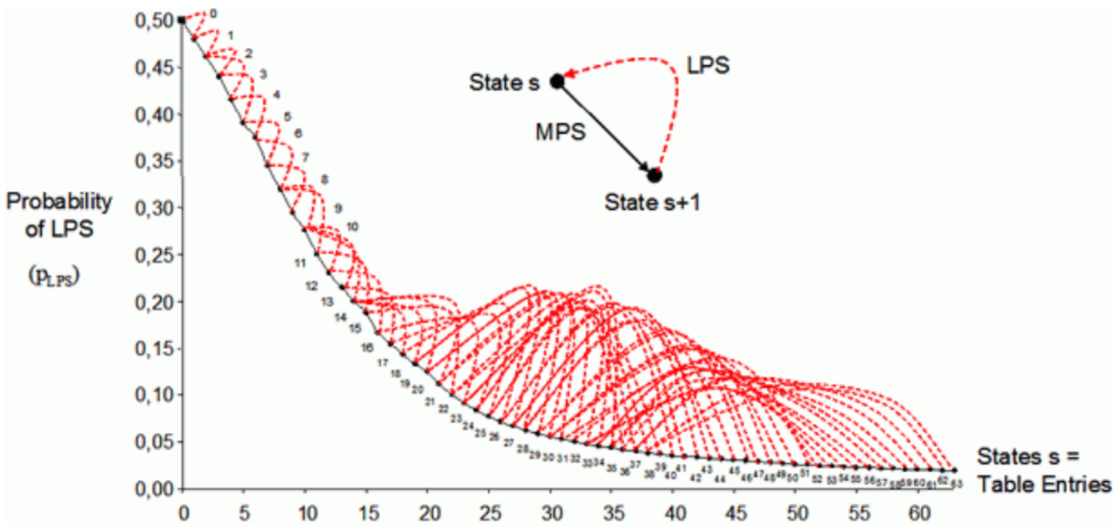


Figure 2.10: Probability state transition for MPS and LPS in BAE [8]

As illustrated in Fig. 2.9, P_{LPS} is required for computing the value of rLPS. In CABAC, the P_{LPS} ranging from 0 to 0.5 is quantized to 64 discrete probability states specified by a index (pstateIdx) ranging from 0 to 63. According to the value of input bin, the transition of probability state in context model is illustrated in Fig. 2.10. Instead of using multiplication, the LPS probability updating in CABAC is based on table looking up, which speeds up the coding process.

The calculation rLPS $=$ Range $\times$ P_{LPS}, presented in Fig. 2.9, includes operation of multiplication. In order to speed up this process in CABAC, the Range is quantized to four R_Q value first. Then, rLPS is quantized to 256 values based on R_Q and pstateIdx (probability state index of P_{LPS}). As a result, computing of rLPS can be simply done by looking up in a two-dimension table, in which R_Q and pstateIdx are the two indices.

The coding process described above is referred to as regular coding mode of binary arithmetic coding in CABAC. There are two alternative coding modes: bypass coding and terminate coding. Bypass coding mode is applied for fast interval division in encoding specific type of syntax elements. In bypass coding, probability of both symbol 0 and 1 are assumed to be 0.5. In the equal probability case, no context model is selected. The update of pstateIdx and MPS in context model is also not necessary. To further simplify the computation in equal probability case, the Range keeps unchanged while the Low is doubled, updated and renormalized based on each input bin. The process of bypass coding is shown in Fig. 2.11 where L represents the Low and R represents the Range. A renormalization process is also included in bypass coding mode.

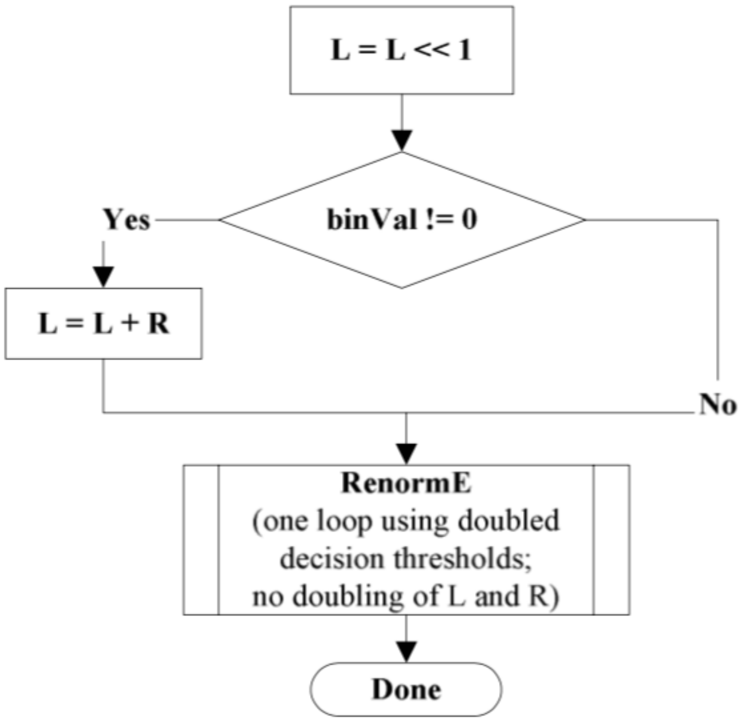

L = L << 1
binVal != 0
Yes
No
L = L + R
RenormE
(one loop using doubled decision thresholds; no doubling of L and R)
Done

Figure 2.11: Bypass coding process in binary arithmetic coding [7]

Terminate coding mode is used for coding syntax element mb_type when the type of macroblock is IPCM and syntax element end_of_slice. In terminate coding, no context model is selected, the LPS is fixed to 1 and the rLPS is fixed to 2. Other than the fixed LPS and rLPS, the following interval division and renormalization process is same as regular coding mode. If the input bin is equal to 1 when terminate coding mode is invoked, the coding process of current slice is going to be terminated after the terminate coding process and additional output bits flushing procedure is invoked at the end of the process.

Furthermore, an initialization procedure is included in CABAC. After coding of one slice is finished, a context model initialization process is invoked, in which all pstateIdx and MPS value for the 399 models are reset to a set of initial values, and the interval parameter Range and Low are also reset to 511 and 0, respectively. The set of initial value for context model is specified by the slice quantization parameter (QP), type of the slice and a context initialization index. As the coding status in binary arithmetic coding is reset and initialized at beginning of each slice, the coding process of each slice is independent in CABAC.

2.5 Previous Work of CABAC Encoder

CABAC encoder of H.264 has been implemented on several different platforms. Software implementation of H.264 includes the JM reference software, X264, etc. Hardware implementations includes designs for ASICs and Field Programmable Gate Arrays (FPGA).

Most existing implementations of CABAC encoder are designed in pipeline architecture. The design of pipeline architecture for CABAC encoder determines the number of bins can be processed every cycle. The bin processing rate together with the maximum clock frequency determines the overall throughput of the implementation. Due to the serial processing nature in the BAE, parallelism under slice level in CABAC is challenging [3]. However, multiple-bin processing is achieved in design [9], [10], [11], [12], [13] and [14]. Furthermore, due to the complexity of context modeling in CABAC, many existing CABAC designs do not include context modeling function implemented in hardware. In these design, CABAC is not fully accelerated by hardware as the process speed is restrained by the processing rate of host processor. In remainder of this section, several typical CABAC encoder architectures of previous work are analyzed based on the symbol processing rate and the function implemented in hardware.

In the designs reported in [15], [16] and [17], only the binary arithmetic encoder (BAE) is implemented in hardware. The binarizer and the context modeler are left for software. In the design of [15], the BAE is divided into six pipeline stages. First four stages are allocated for context memory accessing and interval updating, while the other two stages are designed for bit generation. A single port memory is used for the context memory. Therefore, three different stages are required for reading the memory, calculating new memory data and writing the memory. The architecture is designed for processing one bin per cycle. However, only 0.33 bin per cycle is achieved, due to the data dependency in first four stages.

The pipeline structure of BAE is optimized in [16] by rescheduling the task and applying early fetch of memory data for next incoming bin. The speedup of adopting the early fetch technique is reported as 15%. In addition, a bubble insertion technique is utilized for the pipeline, and a dual-port Static Random-Access Memory (SRAM) is utilized for the context memory. However, the data dependency is still not completely eliminated by the optimization. An overall 0.53 bin per cycle processing rate is achieved in this design.

The implementation in [17] has a similar architecture as [15]. But additional forwarding technique between the first stage and the third stage is applied. As a result, the data dependency is completely removed, and the processing rate of this design achieves 1 bin per cycle. Same as [16], a dual-port SRAM is used in this design.

A fully hardwired implementation of main profile CABAC encoder is presented in [18], the procedures of binarization, calculation of context index and binary arithmetic coding are all implemented in hardware. In addition, the context initialization is implemented, which takes 415 clock cycles to initialize the encoder for each slice. The processing rate of this design is 0.67 bin per cycle.

Fully hardwired CABAC is also implemented in [19] and [10]. In the design of [19], additional rate-distortion optimization (RDO) is implemented, and 1 bin per cycle is achieved in this design. In addition, caches are implemented in context memory to pre-fetch possible context models for upcoming coding process. As a result, the memory access activity is reduced, and the dynamic power is saved by the reduction of switching activity. In [10], some parallelism is achieved: the context memory is rearranged into two memory banks and all the data path following the context memory are doubled for processing two bins in parallel. However, the multi-bin BAE only process two bins in parallel when the ctxIdx for these two bins are the same. Hence, the overall processing rate in [10] is 1.42 bins per cycle. Due to doubling the data path in the BAE, the area cost of this design is increased.

In design [18], [19], [10] and [20], an additional neighbor memory is implemented. The neighbor memory stores the coded macroblock's syntax elements of current slice. Instead of reading from the input, the neighbor macroblock information for context modeling can be accessed from the memory. As a result, no data of neighboring macroblock is extracted and transferred from the host H.264 system. The self-contained neighbor memory makes the CABAC encoder more independent compared to other designs. Better plug-able feature on SoC system of H.264 encoder is also guaranteed by this additional memory. However, the neighbor memory takes large area. For 4K video encoding, at least 34.6K bits additional memory is required. The memory interface logic and buffers also consume large circuit area.

In [14], the processing rate achieves 4 bins per cycle. In this design, binarizer and BAE are implemented while the context modeler is left for software. For multi-bins processing, the SRAM-based context memory is divided into 6 banks. The data path for reading and updating the memory and the interval division is duplicated six times in this architecture. A parallel processing engine is adopted to process residual related syntax elements in parallel. In addition, a syntax element feeding control is also implemented to eliminate bubbles in the pipeline caused by the parallel processing engine. Benefit from all the parallelism optimization, this design achieves higher symbol processing rate compared to other designs. But the area cost and the length of critical path also increase significantly.

Due to the high computational complexity and large size data storing in context modeling, the reported FPGA implementations mainly focus on the process of BAE and binarization. The BAE implementation in FPGA is reported in [21], [15] and [9]. Two CABAC binarizer implemented in FPGA are reported in [22] and [23]. In design [24], the CABAC BAE is implemented in FPGA with an encryption technique appended. The encryption and compression of a symbol is executed simultaneously in the process of interval division and renormalization. After the coding process, the compressed image of the video is also encrypted with no impact on bit-rate.

Chapter 3

Architecture and Circuit Implementation

3.1 Overview of the Proposed CABAC Architecture

As described in section 2.4, the CABAC encoder consists of three elementary components: Binarizer, context modeler (CM), binary arithmetic encoder (BAE). The binarizer encodes each input syntax element into a string of bins while BAE encodes each bin into bit-stream by referring to a context model selected by CM. After coding each bin, BAE updates the context model being referred to, as well as the value of interval Range and Low before processing next bin. Therefore, a serial processing nature is presented in BAE, and the processing rate of the CABAC encoder is determined by BAE, regardless how fast the binarizer and CM are.

A preliminary top-level architecture is shown in Fig 3.1. In order to handle the difference in the processing rate of binarizer and BAE, a Parallel-In-Serial-Out (PISO) buffer is inserted between binarizer and CM, dividing the CABAC encoder into two independent stages. The PISO buffer is a combination of First-in-First-out (FIFO) buffer and PISO shift register. It is able to buffer number of binstrings and shift the bin out one by one. In this architecture, the binarizer in first stage is implemented by combinational logic only. In the second stage, CM calculates the context index (ctxIdx) for each bin from PISO buffer, and packs them together with the coding mode of BAE. Then, the packet of bin, ctxIdx and coding mode is sent to BAE.

The neighbor macroblock information used for context modeling is read from an slice

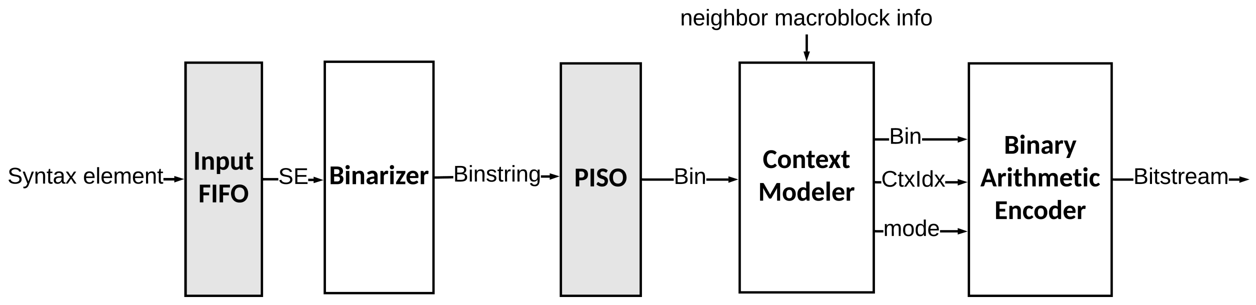


Figure 3.1: Preliminary top-level architecture of the CABAC encoder

memory in the H.264 host processor. The macroblock information stored in this memory is shared by CABAC encoder and other functional units that reference information in neighbor macroblocks, such as intra-prediction and motion estimation.

The Proposed Pipelined Architecture

As shown in Fig. 3.2 , the proposed pipelined architecture of CABAC encoder is derived from the preliminary architecture introduced above. In the proposed architecture, the encoding flow of CABAC is divided into six pipeline stages. The final implementation of CABAC encoder proposed in this thesis is based on this architecture.

At the beginning of first stage, the input FIFO buffers the input syntax element and the corresponding coding information, for example, the index of block/partition that input syntax element belongs to, the prediction direction for motion-estimation related syntax elements and the corresponding syntax element information in neighbor macroblock. A syntax element parser is implemented in the first stage. It parses the input syntax element based on its type and value, and records the syntax element information for current coding macroblock.

Instead of processing the whole context modeling procedure in one pipeline stage, a two-stage context modeling architecture is first introduced in design [19]. In this design, the calculation of context index increment (ctxIdxInc) referencing syntax elements of neighboring macroblock is finished in the first context modeling stage, while the calculation of remaining ctxIdxInc and ctxIdx that refers to result of binarization is finished in the second stage. A similar architecture for context modeling is adopted in the proposed design. As shown in Fig. 3.2, the CM unit is divided into CM1 and CM2, and a FIFO buffer is inserted between the two stages. Different from design [19], the calculation of ctxIdxInc for residual data syntax elements is also included in CM1. In this two-stage

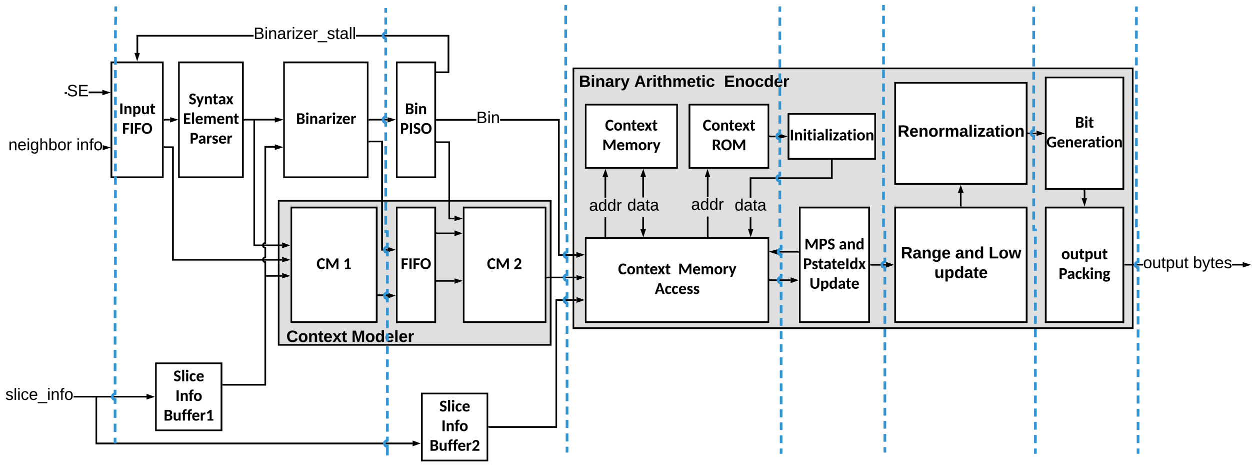


Figure 3.2: High-level pipeline architecture of the proposed CABAC encoder

CM scheme, the reference data for CM1, including the coded syntax elements of current macroblock, the syntax elements of neighboring macroblock and the accumulated information for residual data syntax elements, is not buffered and passed to the second stage. Only the ctxIdxInc, length of binstring and syntax element type are buffered in the FIFO. As a result, the FIFO size between stage 1 and stage 2 are significantly reduced by splitting the CM. The critical path in stage 2 is also reduced because part of the calculation is finished in the previous stage. In addition, the FIFO between CM1 and CM2 is synchronized with the PISO buffer, providing the proper information for each binstring in the PISO buffer.

The BAE is partitioned into six consecutive functional units which are scheduled in four pipeline stages from stage 3 to stage 6. The context memory access unit is in stage 3, reading and writing context model memory with specified ctxIdx when BAE is in regular coding mode. It also accesses the context ROM when BAE is in initialization mode. The context model data MPS and PstateIdx is updated in the following stage 4. In stage 5, the interval registers Range and Low are updated and renormalized. In the final stage, the output bits are generated based on coding mode. Lastly, the generated bits are packed into one or more output bytes by the following output packing unit. It is a natural way to divide the BAE process into pipeline stages as introduced above. Similar procedure division scheme for BAE pipeline structure is presented in design [15], [21], [17], [18], [25], [19] and [10].

The first stage and following stages are relatively independent because of the PISO buffer inserted, and the time point of finish encoding one slice is different at the first and following stages.

Hence, two slice information buffers are adopted, providing the proper slice information for stages before and after the PISO buffer.

The detailed design of Binarizer, CM and BAE is elaborated in the following sections of this chapter.

3.2 Design of Binarizer

3.2.1 Implementation of Binarization Algorithm and Code Format

In the process of binarization, 18 different types of syntax element are encoded into binstring. 5 binarization algorithm and combinations of two different algorithm are adopted in this process. The value range differs from syntax element to syntax element, and the different value range results in diverse port size and code format in each binarization sub-module. Therefore, analysis of value range for each syntax element is important for binarizer implementation. In this section, the implementation of each binarization algorithm together with the value range and code-word format for each syntax element type is described.

Table Mapping

mb_type and sub_mb_type are the two syntax elements that use table mapping for binarization. Value range for both of these two syntax elements is based on the slice type specified. mb_type in P, B slice has a binstring composed by a prefix and suffix. According to the standard [1], a conclusion of the range of value and length of coded bins for mb_type and sub_mb_type is listed in Table 3.1.

Table 3.1: Value range and maximum binstring length for syntax elements using table mapping binarization

Syntax Element	Range of Value	Maximum Length of Binstring (bits)
mb_type (I)	[0, 25]	7
mb_type (P)	[0,3] [5,30]	8
mb_type (B)	[0, 48]	13
sub_mb_type (P)	[0, 3]	3
sub_mb_type (B)	[0, 12]	6

Table mapping binarization for mb_type and sub_mb_type is implemented by five looking up table (LUT) modules. Three LUTs are designed for mb_type in I, P, B slice. The other two LUTs are designed for sub_mb_type in P and B slice. The binstring for each macroblock/sub-macroblock type value and slice type are specified by table 9-26, 9-27 and 9-28 in the standard [1]. 1 to 13 valid bits are generated from mb_type LUTs and 1 to 6 valid bins are generated from sub_mb_type LUTs.

Unary Coding

Unary coded word is consist of number of ones followed by one zero. The number of ones equals the value of input syntax element. mb_delta_qp and ref_idx_lX are the two syntax elements that use unary coding for binarization. The value of ref_idx_lX has two possible ranges based on field/frame coding mode of current macroblock:

$$\begin{cases} 0 \quad to \quad num\_ref\_lX\_idx\_active - 1, & \text{in frame coding mode} \\ 0 \quad to \quad 2 * (num\_ref\_lX\_idx\_active - 1) + 1, & \text{in field coding mode} \end{cases} \tag{3.1}$$

num_ref_idx_lX_active is the number of active reference picture index in the inter-prediction list X. According the slice header semantics specified in standard [1], the number of active index is derived as follow:

$$num\_ref\_idx\_lX\_active = \begin{cases} num\_ref\_frames * 2, & \text{in frame coding mode} \\ num\_ref\_frames, & \text{in field coding mode} \end{cases} \tag{3.2}$$

As the maximum number of reference frames for inter-prediction is 16 in H.264 main profile, in both field and frame coding mode, the ref_idx_lX ranges from 0 to 31, inclusive. Unary coding has a coded bins length that is equal to syntax element value plus one. Therefore, the maximum length of bins for ref_idx_lX is 32 bits.

The signed value of mb_qp_delta is first mapped to an unsigned value by looking up table. Then the mapped value is binarized by unary coding. The range of mb_qp_delta is -26 to 25, inclusive. After mapping, the mapped value ranges from 0 to 52, inclusive. Therefore, the maximum length of binstring for mb_qp_delta is 53 bits.

The Unary binarization module is implemented by barrel shifters. The unary code is generated by following steps. Step1: A 53-bit all one bit-pattern is left-shifted by the value of syntax element. Step2: Inverting all bits. Step3: The pattern is left-shifted again by one bit. After the Unary coding process, 1 to 53 valid bins are generated.

Truncated Unary (TU) Coding

TU coding has a Unary code word that is truncated when the length is larger than the parameter *cmax*. Therefore, the maximum length of TU code is equal to *cmax*. Intra_chroma_pred_mode, suffix of CBP, prefix of MVD and prefix of coeff_abs_level_minus1 are encoded in TU coding binarization. The maximum *cmax* is 14, for encoding coeff_abs_level_minus1 prefix. The implementation of TU coding is same as the Unary code if the syntax element value is smaller than the parameter *cmax*. Otherwise a *cmax*-bit length all-one pattern is selected as the TU code. 1 to 14 valid bins are generated in TU coding.

Fixed length (FL) Coding

The FL code word is basically the fixed-length reverse-order binary representation of the syntax element value. A parameter *cmax* specifies the maximum value to be coded, and the maximum length of the FL coding is derived by:

$$length = ceil(log_2(cmax + 1)) \tag{3.3}$$

An example of FL binarization with length fixed to 3 is given in Table 3.2. rem_intra4x4_pred_mode is encoded by FL coding with *cmax* set to 7. The prefix of CBP binstring is also encoded in FL coding with *cmax* set to 15. Therefore, 1 to 4 valid bins are generated in FL binarization. In addition, the binarization of CBP invokes two types of binarization. The first two bits of CBP value are coded in TU as CBP-chroma suffix, while and the last four bits of CBP value are coded in FL as CBP-luma prefix. The 6-bit CBP, ranging from 0 to 47, inclusive, is coded into 6-bit binstring after the FL and TU binarization.

Table 3.2: An example for fixed-length coding with $cmax = 7$

syntax element Value	Fixed Length Coded Bins
1	100
2	010
3	110
4	001
5	101
6	011
7	111

Unary-Exponential Golomb k-th-Order (UEGk) Coding

UEGk coding bins consist of a TU prefix and EGk suffix. The parameter *cmax* of the TU prefix is specified by parameter *uCoff* in UEGk coding. mvd_lX and coeff_abs_level_minus1 are the two syntax elements binarized by UEGk coding, and they are coded in UEG 3th-order (UEG3) coding and UEG 0th-order (UEG0) coding, respectively.

According to the H.264 standard [1], range of MVD varies in different profiles and levels. In the targeting main profile level 5.1, horizontal MVD component ranges from -2048 to 2048, with -2048 inclusive and 2048 exclusive. Vertical MVD component ranges from -512 to 512, with -512 inclusive and 512 exclusive. With *uCoff* set to 9 and order k set to 3, the MVD binstring has a maximum length of 28 bits, 9 bits for prefix and 19 bits for suffix. The value range of coeff_abs_level_minus1 is determined by the value of transform coefficient levels. The supported coefficient levels ranges from -2^{15} to 2^{15}, inclusive. Hence, coeff_abs_level_minus1 ranges from 0 to $2^{15} - 1$, inclusive. With *uCoff* set to 14 and the order k set to 0, the binstring for this syntax element has a maximum length of 43 bits, 14 and 29 bits for prefix and suffix, respectively.

Compared to other binarization techniques, UEGk coding is relatively more challenging for implementation. The suffix EGk coding algorithm described in the H.264 standard takes variable numbers of loops, which makes finishing UEGk binarizaiton in one pipeline stage not possible. In order to complete the binarization process in one cycle, the loop in the algorithm is unrolled and implemented in pure combinational logic.

For binarization of coeff_abs_level_minus1, the 0th-order EG0 coding is invoked. The EG0 code word is also composed by a prefix and suffix. The prefix is X-bit ones followed by 1-bit zero, where X is determined by the value range of the difference between the absolute value of coefficient level and *uCoff*. This difference also determines the EG0 suffix code word. let *sufS_plus1* be the difference. Then, the value range of *sufS_plus1* can be inferred by the number of leading zeros in *sufS_plus1*. Hence, a special leading zero detecting module is implemented for EG0 coding. The proposed EG0 coding algorithm for hardware is shown in Algorithm 1. Similar UEGk binarization based on leading zero detecting is also reported in design [26], [10], [20] and [23].

Algorithm 1 Pseudo-code of EG0 Coding

```
$sufS\_plus1 = SE\_value - uCoff + 1$        # $uCoff$ =14, $sufS\_plus1$ and $SE\_value$ are both 15-bit
if $sufS\_plus1 == 1$ then
    $EG0\_bins = 0$
    $EG0\_length = 0$        # the UEG0 code has only TU prefix
else
    $N =$ the number of leading zeros in $sufS\_plus1$
    $X = 14 - N$
    $EG0\_bins\_prefix = \{X\text{-bit } 1, 1\text{-bit } 0\}$
    $EG0\_bins\_suffix =$ lower $X$ bits of $sufS\_plus1$
    $EG0\_length = 2X + 1$
end if
```

Table 3.3: Interval division for UEG3 coding

Interval Index	Range of Value
0	9 to 16
1	17 to 32
2	33 to 64
3	65 to 128
4	129 to 256
5	257 to 512
6	513 to 1024
7	1025 to 2047

The MVD invokes the 3rd-order EG3 coding. EG3 code word is composed by three parts: prefix, suffix and a sign bit. For $|SE\_value| < uCoff$, where $uCoff$=9, an all-zero code word is generated. For $|SE\_value|$ in other value ranges , 8 different code formats are generated for the 8 intervals within the value range from 9 to 2047. The interval division is shown in Table 3.3. Similar to EG0, which interval does $|SE\_value|$ falls within can be inferred by the number of leading zeros in $|SE\_value| - 1$. Based on this idea, a leading zero detecting module is also adopted in EG3 coding, and the proposed EG3 coding algorithm for hardware is shown as follow:

Algorithm 2 Pseudo-code of EG3 Coding

```
$abs\_SE\_value\_minus1 = |SE\_value| - 1$    # $abs\_SE\_value\_minus1$ is 11-bit
$signbit = SE\_value[MSB]$
if $|SE\_value| < uCoff$ then
    $EG3\_bins = \{18\text{-}bit\ 0, signbit\}$
    $EG3\_length = 1$
else
    $N =$ the number of leading zeros in $abs\_SE\_value\_minus1$
    $X = 7 - N$    # X indicates that $|SE\_value|$ falls within interval X
    $L = 1 \ll (X + 3) + 1$
    $sufS\_X = |SE\_value| - L$
    $EG3\_bins = \{X\text{-}bit\ 1, 1\text{-}bit\ 0, sufS\_X[X + 3 - 1 : 0], signbit\}$
    $EG3\_length = 2X + 5$
end if
```

Table 3.4: Summary of value range and maximum binstring length for different syntax elements

Syntax Element	Binarization Method	Range of Value	Maximum Length of Binstring
mb_type	Table Mapping	[0, 48]	13
Ref_idx_lX	Unary	[0, 31]	32
Intra_chroma_pred_mode	Truncated Unary	[0, 3]	3
flags	Fixed Length	[0, 1]	1
rem_intra4x4_pred_mode	Fixed Length	[0, 7]	3
mvd_lX	UEG3	[-512, 512) (Vertical) [-2048, 2048) (Horizontal)	28
coeff_abs_level_minus1	UEG0	[0, 2^{15}-1]	43
coded_block_pattern	FL+TU	[0, 47]	6
mb_qp_delta	Mapped Unary	[-26, 25]	53

Summary

The value range and maximum binstring length for each syntax element is listed in Table 3.4 with its binarization methods. The overall value range of syntax element is from -2048 to $2^{15} - 1$, inclusive. A 16-bit signed input port is allocated for the value of syntax element, while a 5-bit input

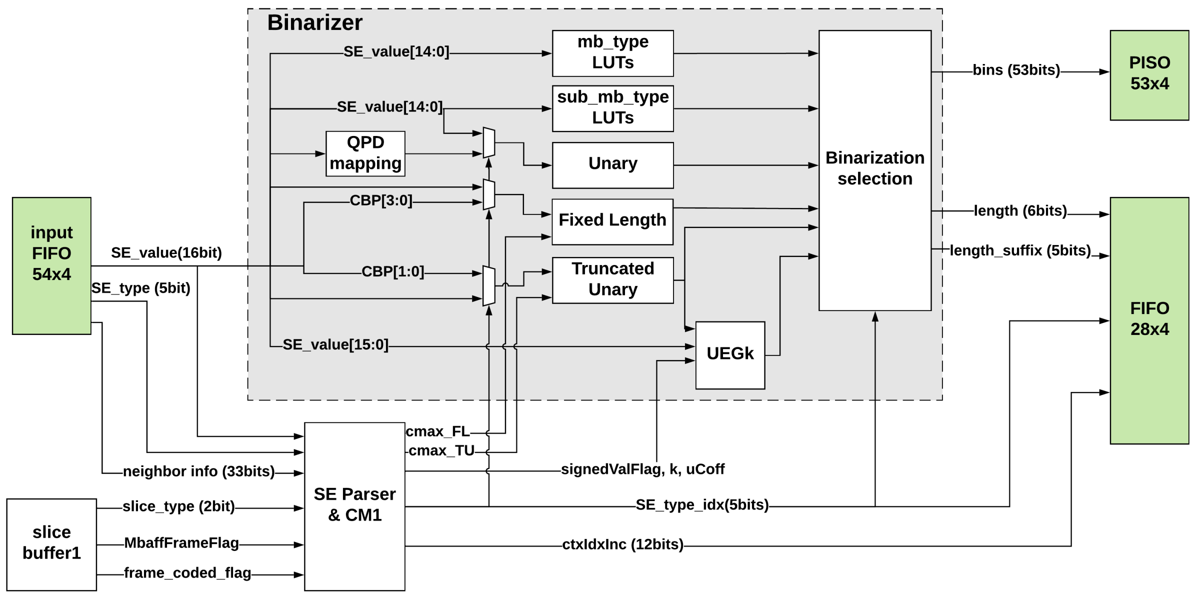


Figure 3.3: Block diagram of the first pipeline stage

port is allocated for the 18 different syntax element types. Also, a 53-bit output port is used for the output bins from binarizer. A 6-bit and 5-bit output ports are allocated for the binstring length and suffix length, respectively.

3.2.2 The Proposed Architecture of Binarizer

The block diagram of the first pipeline stage is shown in Fig. 3.3, with sequential logic between stages marked in green. The major data paths within the binarizer and between the binarizer and syntax element parser are illustrated. The binarizer is implemented by only combinational logic which consists of 6 different binarization modules, one QPD mapping module and a selection module. The parameters of each binarization module are generated from the syntax element parser. Then, each module generates the binstring based on input syntax element value. A syntax element type index from the syntax element parser is used for binarization selection. The concatenation of prefix strings and suffix strings is also done in the selection module by a barrel shifter and an OR gate. After binarization is finished, the 53-bit binstring, together with 6-bit binstring length and 5-bit suffix length that indicates the valid bin location, is written into the following buffers at the end of the first stage.

The syntax element parsing procedure for binarization is basically based on fast LUT as

Table 3.5: Parsing of syntax element for binarization

Syntax Element	SE_type	Param. For Binarizer	SE_type_idx	Description
mb_type	1	-	1	I slice
			3	P slice
			5	B slice
mb_skip_flag	2	cmax_FL=1	7	P slice
			8	B slice
sub_mb_type	3	-	9	P slice
			10	B slice

mvd_lX[][][0] (horizontal)	4	signedValFlag=1
		k=3
		uCoff=9
			11	-

mvd_lX[][][1] (vertical)	5	signedValFlag=1
		k=3
		uCoff=9
			13	-

ref_idx	6	-	15	-
mb_qp_delta	7	-	16	-
intra_chroma_pred_mode	8	cmax_TU=3	17	-
prev_intra4x4_pred_mode_flag	9	cmax_FL=1	18	-
rem_intra4x4_pred_mode	10	cmax_FL=7	19	-
mb_field_coding_flag	11	cmax_FL=1	20	-

coded_block_pattern	12	cmax_FL=15
		cmax_TU=2
			21	-

coded_block_flag	13	cmax_FL=1	23	-
significant_coeff_flag	14	cmax_FL=1	24	Frame coding
			25	Field coding
last_significant_coeff_flag	15	cmax_FL=1	26	Frame coding
			27	Field coding

coeff_abs_level_minus1	16	signedValFlag=0
		k=0
		uCoff=14
			28	-

coeff_sign_flag	17	cmax_FL=1	30	-
end_of_slice_flag	18	cmax_FL=1	31	-

shown in Table 3.5. The parameters including the *cmax* for FL and TU coding and the k, *uCoff*, *signedValFlag* for UEGk coding are generated based on the SE_type (syntax element type). As the binstring selection and concatenation process of some syntax elements is based on the slice type and field/frame mode, an additional index for syntax element type that ranges from 0 to 31 is generated in the syntax element parser. Besides being used in the first stage for binarization, this index is also used for context modeling in both CM1 and CM2. Hence, it is also buffered and passed to next pipeline stage.

3.3 Design of Context Modeler

In the process of context modeling (CM), context index (ctxIdx) for each bin of the binstring is calculated from the ctxIdxInc and ctxIdxOffset. In the first stage, CM1 calculates the ctxIdxInc for those bins referring to the neighboring macroblock information, previous coded information of current macroblock and accumulated residual data information. While in the second stage, CM2 calculates the remaining ctxIdxInc based on bin index (binIdx), previous coded bins and the category of current coding block. Lastly, CM2 generates the ctxIdx by adding up the index increment and offset.

3.3.1 Syntax Element Parsing for Context Modeling

In the syntax element parser, the semantics carried by input syntax elements is also parsed for following context modeling. Several registers are adopted for storing these parsed information for current coding macroblock.

If the input syntax element is mb_type, the type of macroblock is parsed and stored in a 3-bit register. The stored macroblock type includes only Intra16×16, Intra4×4, IPCM, Direct in P/B slice. macroblock types other than these are recorded as other type, as they are not utilized in following CM process. A flag specifying if current coding macroblock is in intra-prediction mode is also generated from the type of macroblock. Furthermore, the macroblock partition mode is parsed and stored in a 2-bit register. In P/B slice, if a macroblock is partitioned into 8×8 sub-macroblocks, sub_mb_type is also parsed. A partition index specifying the location of each sub-macroblock is read in together with sub_mb_type. A 4-bit register is adopted for recording if each sub-macroblock is in direct coding mode. Another 8-bit register stores the partition mode for each sub-macroblock. The partition mode and direct coding mode information of macroblock and sub-macroblock are referenced in following MVD and ref_idx context modeling process.

In addition to mb_type and sub_mb_type, mb_field_coding_flag, mb_skip_flag, mb_qp_delta are also parsed an stored. One bit register is applied for recording if the mb_qp_delta for current macroblock is equal to zero.

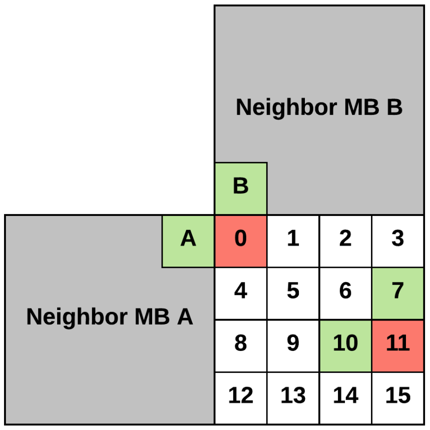
For residual syntax elements, the number of coded coefficient levels that equals one (numLevelEq1) and the number of coded coefficient levels greater than one (numLevelGt1) are the accumulated information which are referenced in computing ctxIdxInc of abs_coeff_level_minus1. Two 4-bit registers are utilized for recording them. CBF indicates the start of coding a transform coefficient block, when both registers are reset to zero. When coeff_abs_level_minus1 is in parsing process, if the syntax element value equals 0, numLevelEq1 counts up by one, otherwise numLevelGt1 counts up by one. Furthermore, a scanning position index is parsed from SCF and LSCF during coding a transform block.

3.3.2 CtxIdxInc Calculation

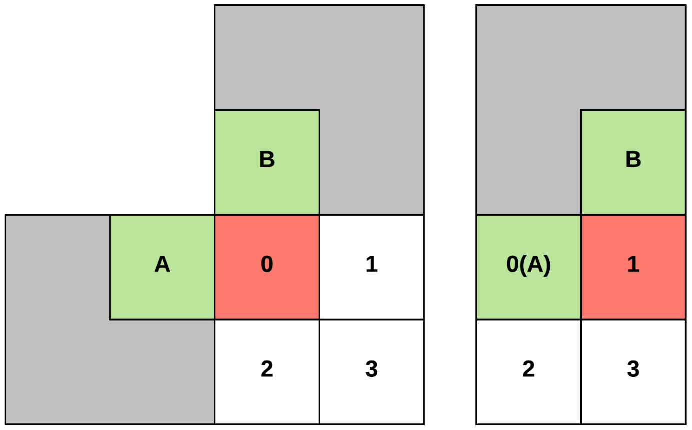
In CM1 (first stage of context modeling) process, ctxIdxInc calculation for some syntax elements is based on the coded syntax elements in neighboring macroblocks, partitions of macroblock, sub-macroblocks or 4×4 blocks. First, two condition flags are calculated for the top and left neighbors, based on the neighbor information. Then, the ctxIdxInc is calculated from the two flags. These syntax elements includes: mb_type, mb_skip, ref_idx_lX, ICPM, mb_field_coding_flag, CBP, MVD and CBF. Among these syntax elements, mb_type, mb_skip, ICPM and mb_field_coding_flag are corresponding to one macroblock. Therefore, they only appear once per macroblock. The neighbor syntax elements they refer to always locate in the top and left neighbor macroblock. The syntax element data of the neighboring macroblock is read from the input FIFO. The ctxIdxInc calculation for these syntax elements is simply based on the algorithm introduced in section 9.3.3.1.1 of the H.264 standard [1].

Syntax elements such as CBP, ref_idx_lX, MVD and CBF correspond to one partition block within the macroblock. The block size varies for different syntax element type and partition mode. In H.264, partitions size includes 16×8, 8×16, 8×8, 8×4, 4×8 and 4×4. Therefore, for these syntax element types, various number of neighbor syntax elements are read in and processed for each macroblock. In ctxIdxInc calculation for these syntax elements, not only the blocks in neighboring top and left macroblock are referred to, the coded blocks within current macroblock are also used as reference. Fig. 3.4 shows some examples of neighbor blocks within and outside current macroblock for 4×4 and 8×8 partition mode.

CBP corresponds to one macroblock, and appears only once per macroblock. However,



(a) 4x4 block neighbors outside/within current MB



(b) 8x8 block neighbors outside/within current MB

Figure 3.4: Neighboring blocks for different macroblock partitions

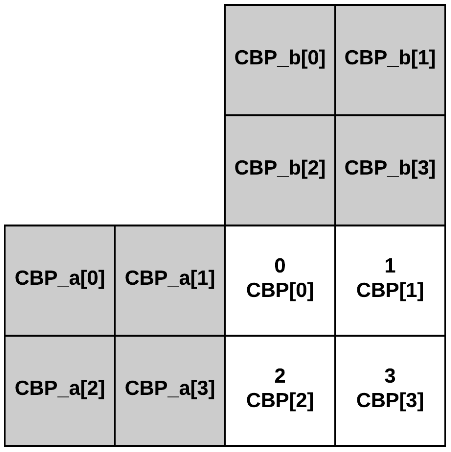


Figure 3.5: Neighboring 8×8 blocks of Coded_Block_Pattern (CBP)

the four CBPLuma (CBP[3:0]) bits within the 6-bit CBP corresponds to the four 8×8 macroblock partitions, respectively. As shown in Fig. 3.5, calculation of the ctxIdxInc for each bin of CBPLuma refers to CBP bits in either neighbor macroblock or current macroblock. The binIdx for binstring of CBPLuma (prefix part) is equal to the index of 8×8 block. In other words, bin with index=X corresponds to block X, which also corresponds to CBP[X]. Therefore, the CBP bit of neighbor can be easily located by binIdx. For example, when processing the CBP bit with binIdx=2 (left-bottom block), the top neighbor CBP bit equals to CBP[0] of current macroblock. The left neighbor CBP bit equals to CBP[3] of neighbor macroblock A, if the neighbor macroblock A is available. A simple LUT is implemented for the index mapping scheme, based on which the condition flags of the two neighbors can be calculated through the algorithm described in section 9.3.3.1.1.4 of the H.264

standard [1]. Modeling of the two remaining CBPChroma bits is simply based on the algorithm in standard, as they only refers to neighboring macroblock. Processing of the 6 CBP bits is in parallel in the proposed design. At the end of CM1 procedure, 6 pairs of neighbors are referenced and 12 condition flags are generated and buffered in the FIFO.

CtxIdxInc Calculation for Ref_idx_lX and MVD

ref_idx_lX and MVD are the two motion estimation syntax elements. They only appear in macroblocks that belong to B slice and P slice. ref_idx_lX corresponds to each 8×8 macroblock partition (sub-macroblock), while MVD corresponds to each partition and sub-partition of macroblock, with size from 16×16 to 4×4. Therefore, in B and P slice, each macroblock is possible to have 1 to 16 MVDs and 4 ref_idx_lX. Each syntax element is read in with the index X and Y specifying the horizontal and vertical location of the block the syntax element corresponds to. The block index of 8×8 and 4×4 are derived as follow:

$$\begin{aligned} block4{\times}4Idx &= X + Y * 4 \\ block8{\times}8Idx &= X + Y * 2 \end{aligned} \tag{3.4}$$

As shown in Fig. 3.6, only blocks at left/top edges of a macroblock refer to neighboring macroblocks for ctxIdxInc calculation, and the blocks at left and top edge are indexed by X=0 and Y=0, respectively. Based on this, a simple neighboring index deriving algorithm for both 8×8 and 4×4 blocks is implemented as shown in Algorithm 3. Two generated flags indicate if the left and top neighbors are in current macroblock.

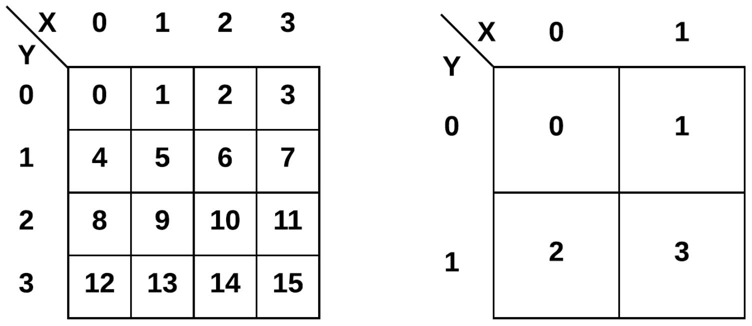


Figure 3.6: Horizontal and vertical index of 8×8 and 4×4 blocks

For ctxIdxInc calculation of ref_idx, coded ref_idx of current macroblock is stored as reference for following ref_idx. The two types of ref_idx are stored: forward-prediction ref_idx and

Algorithm 3 Pseudo-code of Index Derivation in CM1

```
if $X \neq 0$ then
    $left\_X = X - 1$
    $left\_is\_currmb = 1$
else
    $left\_X = X$
    $left\_is\_currmb = 0$
end if
if $Y \neq 0$ then
    $top\_Y = Y - 1$
    $top\_is\_currmb = 1$
else
    $top\_Y = Y$
    $top\_is\_currmb = 0$
end if
```

backward-prediction ref_idx. The direction of prediction is specified by a parameter *ref_list*. For example, ref_idx_l0 corresponds to *ref_list*=0, which designates backward reference list. Furthermore, only ref_idx with block8×8Idx equals to 0, 1, 2 is stored. Because the right-bottom block is never referred to as neighbor for current macroblock. Therefore, in total 2×3 ref_idx are stored for current macroblock. As the condition flag calculation depends only on if ref_idx>1, =1 or =0. The 5-bit ref_idx can be reduced and stored in only 2 bits. Hence, instead of using a 40-bit register for all ref_idx bits in current macroblock, only a 12-bit register is adopted for ref_idx. Apart from neighbor's ref_idx, whether or not the neighbor is coded in direct mode is another reference in this process. If the reference 8×8 block is in current macroblock, the stored sub_mb_type information in syntax element parsing stage is read by the specified neighbor's block8×8Idx. The detailed calculation of condition flag and ctxIdxInc of ref_idx is introduced in section 9.3.3.1.1.6 of the standard documentation.

The ctxIdxInc calculation process of MVD derived from the description in H.264 standard [1] is shown in Algorithm 4 and 5 , where A and B stands for left and top neighbor, respectively. There are possibly four types of MVD within one macroblock. The type is specified by two parameters:

Algorithm 4 Pseudo-code of ctxIdxInc Calculation for MVD (continued on next page)

```
if $block\_A\_available \neq 1$ then
  $A = 0$
else if $compIdx == 1$ && $MbaffFrameFlag == 1$ then
  if $currMB\_field == 0$ && $block\_A\_field == 1$ then
    $A = abs\_mvd\_A$ <<1
  else if $currMB\_field == 1$ && $block\_A\_field == 0$ then
    $A = abs\_mvd\_A$ >>1
  else
    $A = abs\_mvd\_A$
  end if
else
  $A = abs\_mvd\_A$
end if
if $block\_B\_available \neq 1$ then
  $B = 0$
else if $compIdx == 1$ && $MbaffFrameFlag == 1$ then
  if $currMB\_field == 0$ && $block\_B\_field == 1$ then
    $B = abs\_mvd\_B$ <<1
  else if $currMB\_field == 1$ && $block\_B\_field == 0$ then
    $B = abs\_mvd\_B$ >>1
  else
    $B = abs\_mvd\_B$
  end if
else
  $B = abs\_mvd\_B$
end if
```

Algorithm 5 Pseudo-code of ctxIdxInc Calculation for MVD (continued)

```
if $A + B < 3$ then
    $ctxIdxInc = 0$
else if $A + B > 32$ then
    $ctxIdxInc = 2$
else
    $ctxIdxInc = 1$
end if
```

ref_list and *compIdx*. *ref_list* specifies the prediction direction of MVD while *compIdx* specifies it is a horizontal or vertical MVD component. Same as ref_idx, the coded MVD in current macroblock is stored as reference for following MVDs. In the worst case that the macroblock is partitioned into 16 4×4 blocks, 15 blocks' MVDs are stored. Because the block with index=15 is never used as reference in current macroblock. Therefore, 4×15 MVDs are store for one macroblock. Each MVD is represented by 12 signed bits. However, according to Algorithm 5, for large value MVD, only whether or not |MVD|>>1 is larger than 32 affects the result of ctxIdxInc. Therefore only 7 bits is used for storing the coded |MVD|. If |MVD| is larger than 64, value 65 is stored for this MVD. In total four 105-bit (15×7) registers are applied for the four MVD types, respectively. The register size for storing coded MVD is significantly reduced. After processing one MVD, only one of the four MVD registers selected by the 2-bit {*compIdx*, *ref_list*} is updated.

To update and reference the coded MVD in these registers, a partition index is needed for each MVD corresponded partition. In the MVD processing, 4×4 block is the basic unit. Every MVD is read in with its X and Y index ranging from 0 to 3. Then the block4×4Idx is calculated by Eq. 3.4. According to the scheme in JM reference software, the MVD for one partition is assigned with the index of the left-top 4×4 block within the partition, regardless of the partition shape. Some examples of MVD partition index for different partition mode is shown in Fig. 3.7. (a) shows the case that macroblock is divided into two 16×8 partitions. At most two MVDs are read in for this macroblock with block4×4Idx = 0, 8. In (b), the macroblock consist of two 8×16 partitions. At most two MVDs are read in with block4×4Idx = 0, 2. (c) and (d) show the macroblock partitioned into four 8×8 sub-macroblocks. Each sub-macroblock can be partitioned in 8×8, 8×4, 4×8 and 4×4. At most 9 MVDs are read in for the case of (c) and (d), and their block4×4Idx are 0, 2, 6, 8, 9, 10,

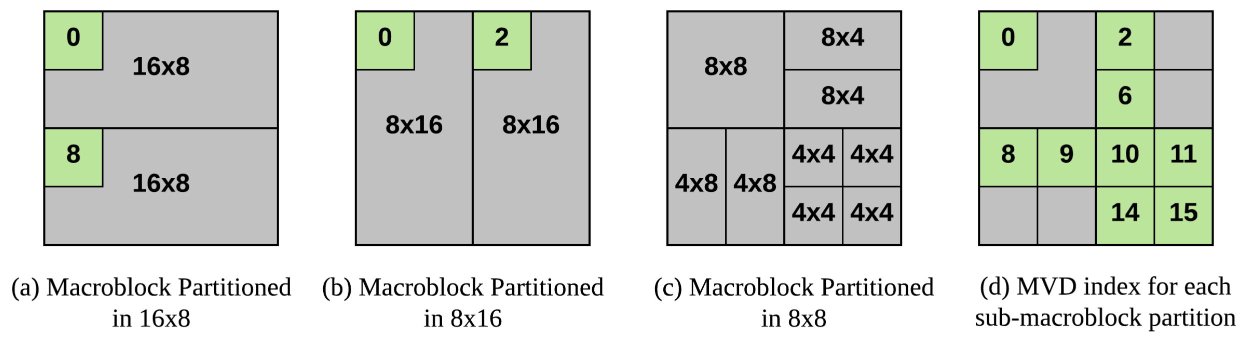

0
16x8
8
16x8
(a) Macroblock Partitioned in 16x8
0
2
8x16
8x16
(b) Macroblock Partitioned in 8x16
8x8
8x4
8x4
4x8
4x8
4x4
4x4
4x4
4x4
(c) Macroblock Partitioned in 8x8
0
2
6
8
9
10
11
14
15
(d) MVD index for each sub-macroblock partition

Figure 3.7: MVD partition index for different macroblock partition mode

11, 14 and 15. It is possible that one or more partitions have no MVD read in. For the absence of MVD in a partition, the MVD for this partition is inferred to be zero. Therefore, all the updated MVD registers are reset to the default zeros when end_of_slice_flag = 0, which indicates the last cycle of coding current macroblock. This explains why the MVD storing circuit is not implemented by a 7×60 bit RAM. Because in a RAM circuit, reset of multiple words cannot be finished in one cycle.

For referencing neighbor MVD within current macroblock, different applied partition mode makes the locating of neighbor partition quite challenging. First, the partition mode for current macroblock is obtained from syntax element parser. If the partition mode is 16×16, then the neighbor MVDs locate in neighboring macroblock A and B. The two MVD value is read directly from input FIFO. If the partition mode is 16×8 or 8×16, as shown in Fig. 3.7 (a) and (b), the top 16×8 block and the left 8×16 is referred to by the following partition. The coded reference MVDs for them are both assigned with block4×4Idx=0 (the left-top block). All the other reference MVDs in 16×8 and 8×16 mode are from external neighbor macroblocks.

For 8×8 macroblock partition mode, the partition mode of an 8×8 partition (sub-macroblock) is required for referencing 4×4 blocks belong to it. For referencing the top and left 4×4 blocks, partition mode of the two corresponding sub-macroblocks can be read from the syntax element parser by specifying its block8×8Idx. The block8×8Idx can be derived from the neighbor 4×4 block's index X and Y as follow:

$$\begin{aligned} subMBtop\_idx = &\quad top\_X[1] + 2 * top\_Y[1] \\ subMBleft\_idx = &\quad left\_X[1] + 2 * left\_Y[1] \end{aligned} \tag{3.5}$$

Two examples are shown in the Fig. 3.8. (a) shows the partition mode for each sub-

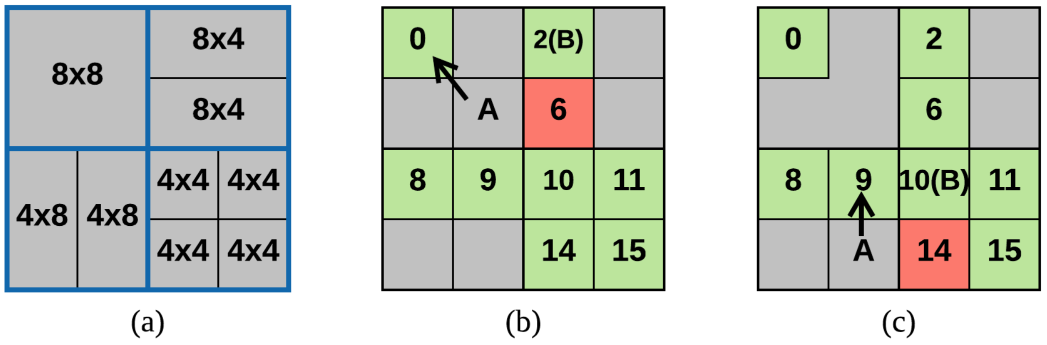


Figure 3.8: Locating neighboring reference MVD in 8×8 partition mode

macroblock. In (b), the current coding MVD is assigned with block4×4Idx=6. The top neighbor is block 2 (block B), which has X=2 and Y=0. From Eq. 3.5, subMBtop_idx=1 is calculated. Hence, the 8×4 sub-macroblock partition mode for top neighbor is read from syntax element parser with the index specified. By the same method, that the left neighbor belongs to the sub-macroblock with subMBleft_idx=0 is derived. The 8×8 partition mode is read for it. In the example of (c), current coding MVD is for block 14. By the same method, the two index are derived: The top neighbor (block B) belongs to sub-macroblock with subMBtop_idx=3. The left neighbor (block A) belongs to sub-macroblock with subMBleft_idx=2. The partition mode for them are 4×4 and 4×8, respectively.

Algorithm 6 Pseudo-code of Index Mapping for Reference MVD

```
if $subMB\_partition\_mode == 8\times 8$ then
    $ref4\times4\_idx = \{X[1], 1'b0\} + \{Y[1], 1'b0\} * 4$
else if $subMB\_partition\_mode == 8\times 4$ then
    $ref4\times4\_idx = \{X[1], 1'b0\} + Y * 4$
else if $subMB\_partition\_mode == 4\times 8$ then
    $ref4\times4\_idx = X + \{Y[1], 1'b0\} * 4$
else if $subMB\_partition\_mode == 4\times 4$ then
    $ref4\times4\_idx = block4\times4Idx$
end if
```

Since the sub-macroblock partition mode is known for the two neighbors, in order to locate the reference MVD for this partition, an index mapping algorithm is implemented. As shown by the arrows in Fig. 3.8 (b) and (c), the block4×4Idx of the actual reference MVD (left-top block) is mapped to the block4×4Idx of the neighbor block A and B. The pseudo-code of the algorithm is shown as Algorithm 6. Based on this mapping method, for the example in Fig. 3.8 (b), the top 8×4 partition's coded MVD is read from block 2, while the left 8×8 partition's coded MVD is read from block 0. In (c), coded MVD for the top 4×4 partition is read from block 10, while the left 4×8 partition's coded MVD is read from block 9.

CtxIdxInc Calculation for CBF and Residual Syntax Elements

CBF is corresponding to each transform coefficient block. The numbers of CBFs within one macroblock is determined by the category of the transform coefficient block, which is specified by the parameter *ctxBlockCat* as shown in Table 3.6. For luma coefficient block, if the block category is luma16×16DC or luma16×16AC, only one CBF for luma appears in each macroblock. If the block category is luma4×4AC, each macroblock has 16 CBFs for luma. Each CBF corresponds to a 4×4 luma block. Furthermore, for the chroma block, the Cb/Cr type of chrominance block is specified by parameter *iCbCr*. If the block category is chromaDC block, each macroblock includes one CBF for Cb and one CBF for Cr. If the category is chromaAC block, each macroblock has 4 Cr CBFs and 4 Cb CBFs. Each CBF corresponds to an 8×8 chroma block. As the right-bottom 4×4 luma block (index=15) and the right-bottom 8×8 chroma block (index=3) are never referred to in the ctxIdxInc calculation, maximum 15 coded CBFs for luma4×4AC and 2×3 coded CBFs for chromaAC are stored as reference. A 15-bit and a 6-bit register are implemented for them, respectively. The locating of reference coded CBFs within the current macroblock for both 8×8 and 4×4 block is simply based on the Algorithm 3 introduced previously. The detailed calculation for the two neighbor condition flags of CBF is elaborated in section 9.3.3.1.1.9 in the standard [1].

Table 3.6: Category of transform block [1]

Category of Block	Maximum Number of Coefficient	ctxBlockCat
Intra16x16_lumaDC	16	0
Intra16x16_lumaAC	15	1
Intra4x4_luma	16	2
chroma_DC	4	3
chroma_AC	15	4

SCF, LSCF and coeff_abs_level_minus1 are the three syntax elements for residual data in H.264. Although ctxIdxInc calculation for them does not refer to neighbor macroblock or partitions, the accumulated residual data information from syntax element parser is referenced in the process.

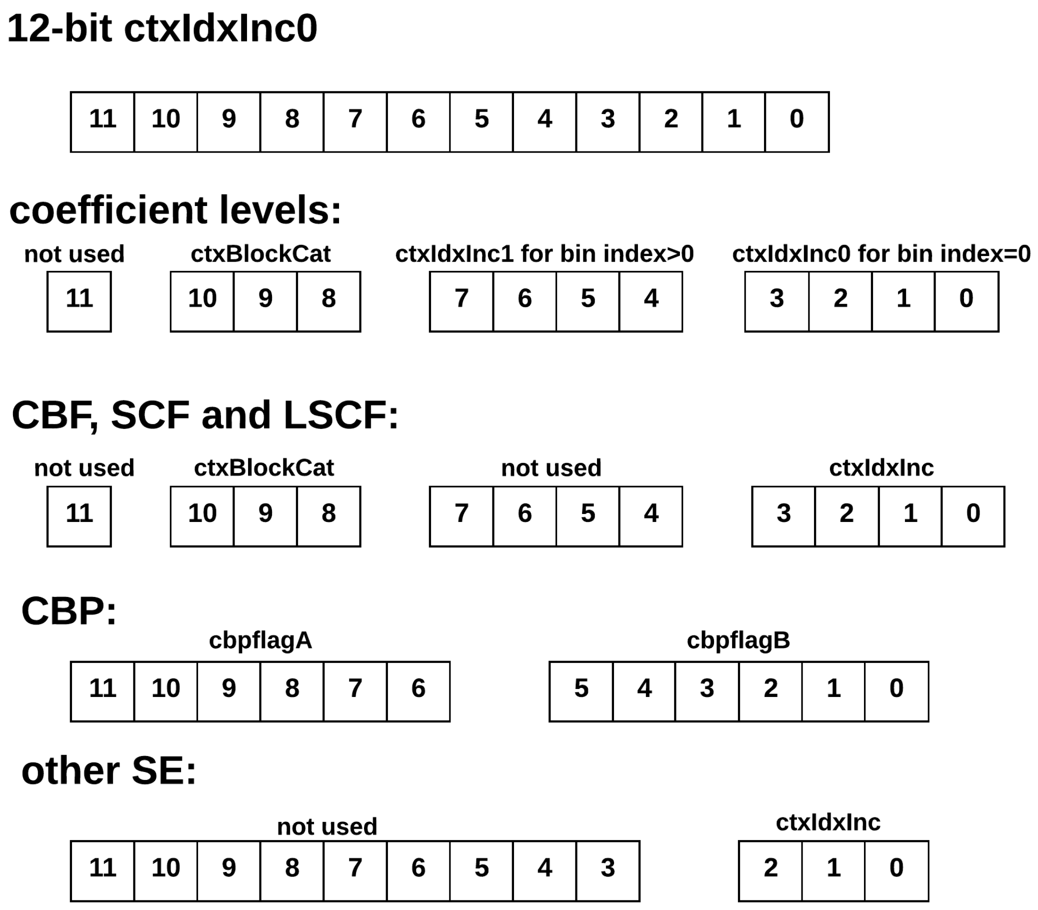


Figure 3.9: The bit packing of ctxIdxInc

Therefore, the ctxIdxInc calculation for residual syntax elements are also implemented in CM1. For LSCF and SCF, the ctxIdxInc is equal to the scanning position (scanningPos) in coefficient block transform. The scanningPos is mapped from the scanning position index parsed in syntax element parser. For ctxIdxInc calculation of coeff_abs_level_minus1, two ctxIdxInc are derived as:

$$\begin{aligned} &\mathit{ctxIdxInc0} = (\mathit{numLevelGt1} \neq 0)?0 : \mathit{Min}(4, 1 + \mathit{numLevelEq1}) \\ &\mathit{ctxIdxInc1} = 5 + \mathit{Min}(4, \mathit{numLevelGt1}) \end{aligned} \tag{3.6}$$

The ctxIdxInc0 and ctxIdxInc1 are for the first bin and the remaining bins, respectively. The *numLevelGt* and *numLevelEq1* are also read from syntax element parser. Additionally, the maximum number of coefficient levels for different block category is also shown in Table 3.6.

The generated ctxIdxInc for different syntax elements are stored in a 12-bit packet called ctxIdxInc0 and buffered to the FIFO between pipeline stage1 and stage2. In stage 2, ctxIdxInc0 is utilized in the calculation of final ctxIdx for each bin in CM2. The packing scheme for different syntax element is shown in Fig. 3.9. For CBF, SCF, LSCF and coeff_abs_level_minus1, the 3-bit parameter *ctxBlockCat* is packed together with the ctxIdxInc. In CM2, *ctxBlockCat* is used for selecting context Index offset for each different block category. For CBP, two 6-bit condition flags are packed instead of the ctxIdxInc. Because the ctxIdxInc for two CBP chroma bins ranges from 0 to 7, while the ctxIdxInc for four CBP luma bins ranges from 0 to 3. The six ctxIdxInc generated in parallel takes 14 bits in total, which increases the cost of the FIFO buffer. Hence, the two 6-bit condition flags is buffered directly, and the ctxIdxInc for each CBP bin is calculated in the following CM2 unit.

3.3.3 Context Index Calculation

The proposed architecture of CM2 (second stage of context modeler) is shown in Fig. 3.10. CM2 is composed by four elementary functional blocks: PISO control unit, BAE coding mode selector and two blocks for generating the ctxIdxInc and ctxIdxOffset, respectively. After the CM2 stage, 1-bit bin, 3-bit BAE coding mode and 9-bit ctxIdx are written into the register buffers for the following BAE stages.

The PISO control unit counts up the number of bins that has been shifted out from current coding binstring. The count number is recorded in a 6-bit register binIdxCnt. Based on the binIdxCnt, length of binstring and length of suffix, the binIdx and suffix_flag are generated. suffix_flag specifies if the current coding bin in CM2 belongs to suffix of the binstring. The control unit also generates the dequeue request signal for the PISO and FIFO buffers at the end of coding each binstring. If a dequeue request is specified, the FIFO shifts out its first data in the queue. Furthermore, the control unit records the bin value with index equals to 1 and 3. These two coded bin value are referred to in the ctxIdxInc selection process for syntax element mb_type and sub_mb_type. In addition, binarizer_stall_flag is forwarded to the input FIFO, which stalls dequeue operation of the input FIFO when the PISO buffer is full.

The selection of ctxIdxOffset for each bin in CM2 depends on the syntax element type, slice type, field/frame coding mode and prefix/suffix part it belongs to. Therefore, a ctxIdxOffset index is calculated by adding up the SE_type_idx and suffix_flag. This index ranges from 0 to 31, inclusive. Then, one 32-entry LUT is implemented for the ctxIdxOffset selection. For CBF, SCF, LSCF and coeff_abs_level_minus1, additional three LUTs are implemented for selecting ctxCatOffset

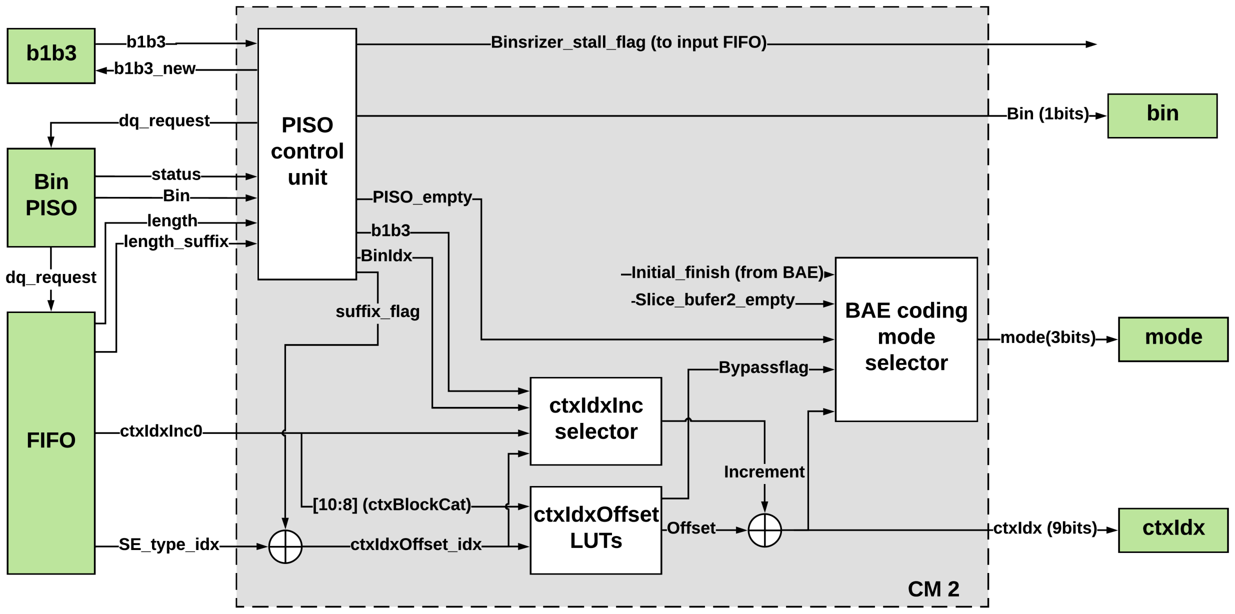


Figure 3.10: The block diagram of context modeling in second pipeline stage

by the parameter *ctxBlockCat*. The ctxCatOffset is added to the ctxIdxOffset. In addition, a bypass flag is generated from the ctxIdxOffset index, specifying bypass mode in BAE.

To generate the corresponding ctxIdxInc for the offset, the same ctxIdxOffset index is specified. Then, the ctxIdxInc for the first bin is read from ctxIdxInc0 which is from the previous CM1 stage. The ctxIdxIncs for remaining bins are selected based on binIdx and the coded bin value.

BAE Coding Mode Selection

The BAE coding mode selection process is shown in the Fig. 3.11. There are four different coding modes. When mode is 3 and 4, the BAE stalls for empty PISO and initialization, respectively. The initialization mode is selected if the initialization process in BAE is not finished or it is the first cycle of a slice in CM2. If initialization mode is selected, the binstring dequeuing and bin shifting operations in the PISO are disabled. Therefore, the first cycle data is hold in CM2 until BAE finishes initialization. The bypass mode is invoked by bypass_flag=1. The terminate mode is invoked by ctxIdx=276. Terminate mode does not necessarily mean the end of slice. Because ctxIdx=276 can be derived also from IPCM macroblock type and end of macroblock. Only ctxIdx=276 with bin=1 indicates the last cycle of current slice in CM2. Furthermore, if the BAE coding mode is not set to initialization, stall, bypass or terminate mode, then it is set to regular coding mode.

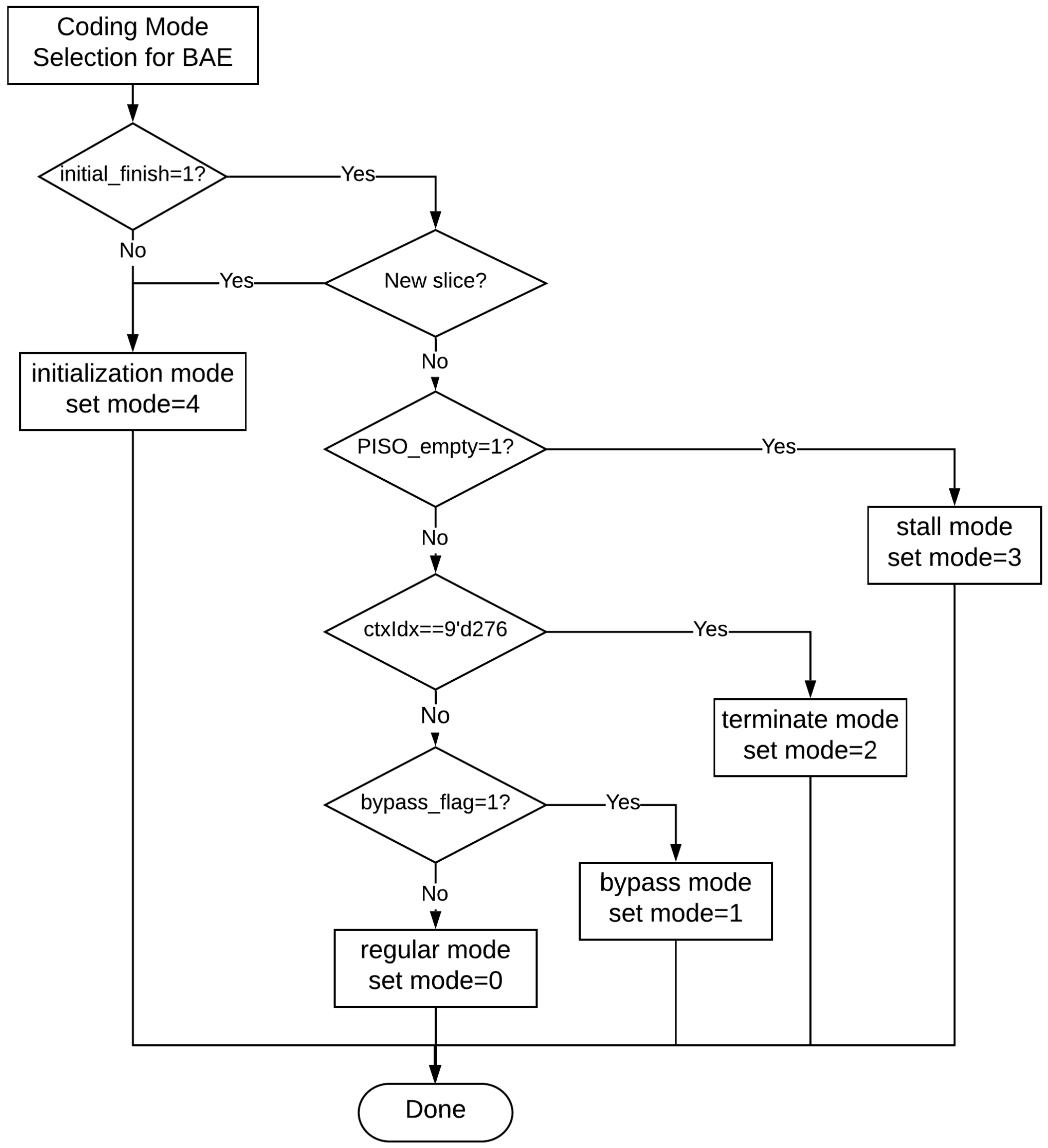


Figure 3.11: The process of coding mode selection for binary arithmetic encoder

3.4 Design of Binary Arithmetic Encoder

BAE is the final component of the CABAC encoder. In the proposed design of BAE, the data path is divided into four pipeline stages. In the first two stages, the context memory is read and update based on the input bin, context index, and coding mode. In the third stage, the interval registers codIRange and codILow for arithmetic coding is updated and renormalized. In the final stage, output bits pattern is generated and packed into one or more bytes output. In this section, the architecture and functional implementation of each stage in the BAE is described.

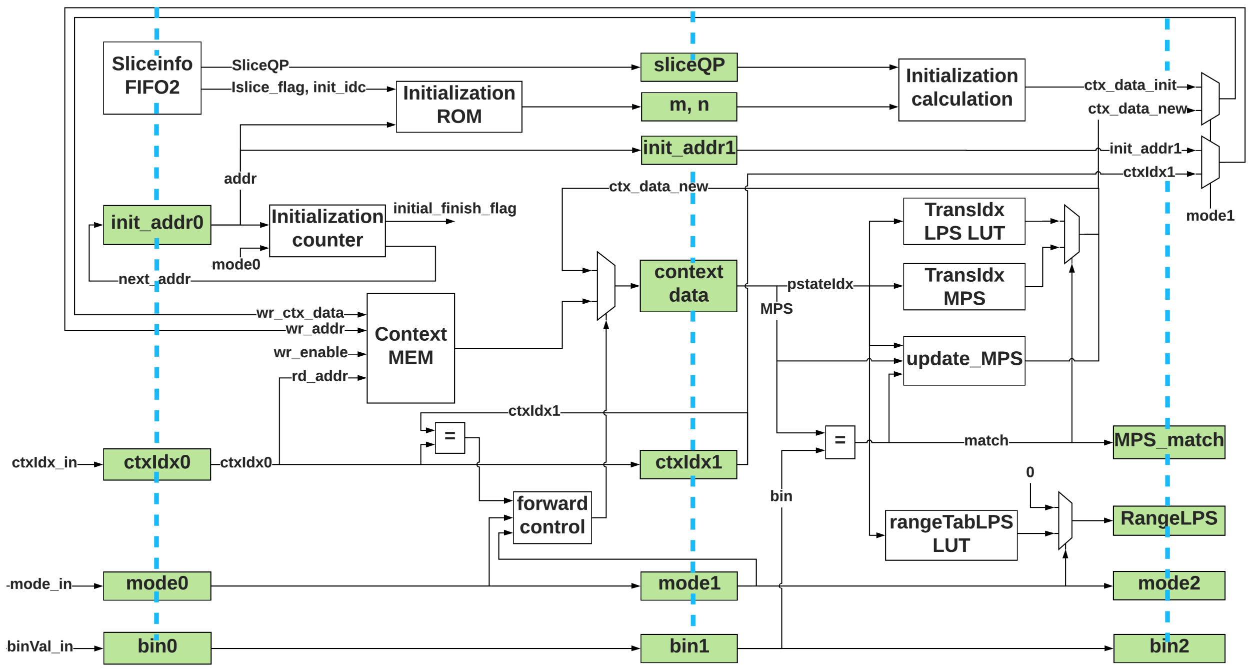


Figure 3.12: The block diagram of first and second stage of BAE. The division of pipeline stages is shown by the dashed lines

3.4.1 Context Memory Access and Update

The Fig. 3.12 shows the major data path of the first two stages of BAE. The input are the bin value, context index and mode. The bin value and mode are buffered in registers bin1, bin2 and mode1, mode2, respectively, for following stages. In the first stage, the two content of context memory: 1-bit MPS and 6-bit pstateIdx are read, and then stored in context data register. In the second stage, the MPS and pstateIdx is updated based on the input bin value. The initialization of the context memory is also implemented in the first two stages. The detailed implementation of context data access, update and initialization is elaborated in the following subsections.

Context Memory

The 7-bit×399 words context memory is implemented by a flip-flop-based dual port RAM, which allows reading and writing of the memory at the same cycle. In regular coding mode, the read and write address for the memory are specified by the ctxIdx0 and ctxIdx1. If the two consecutive ctxIdx are the same, the data read from memory in first stage is not valid. Because the context data for this address is being updated in the second stage and has not been written into the memory. In order to eliminate the data dependence between the memory read stage and memory update stage, a forwarding technique for the updated context data is implemented when ctxIdx0=ctxIdx1 and both mode0 and mode1 are regular coding mode. As a result, throughput of 1 bin/cycle is achieved in the BAE. Furthermore, if mode_in is not regular coding mode, the register ctxIdx0 holds its previous value. When mode1 is not regular mode, the write enable signal of the memory is set to 0. Therefore, the memory access frequency is reduced for non-regular mode. The dynamic power of the memory is reduced.

Updating of MPS and pstatesIdx

The updating process of MPS and pstatesIdx in second stage, as well as the interval Range and Low updating in third stage, depends on whether or not the bin value matches MPS. If the bin equals to MPS, MPS transition process is invoked, otherwise LPS transition process is invoked.

In the proposed design, the LPS transition process of pstateIdx is implemented by a 64×6 bit LUT circuit, while the MPS transition is implemented by combinational circuits. The transition process for pstateIdx is shown in Algorithm 7. Furthermore, if MPS_match=0 and pstateIdx=0, the new value of MPS is set to 1−MPS, otherwise the value of MPS does not change.

Algorithm 7 Pseudo-code of Updating pstateIdx

```
if $MPS\_match == 1$ then
  if $pstateIdx \leq 61$ then
    $new\_pstateIdx = pstateIdx + 1$
  else
    $new\_pstateIdx = pstateIdx$
  end if
else
  $new\_pstateIdx = transIdxLPS[pstateIdx]$
end if
```

Context Memory Initialization

At beginning of each slice, the context models in the memory are initialized based on the slice parameters. In the proposed design, the context memory initialization is finished in 398 cycles

Table 3.7: Context initialization ROM size for different ctxIdx

Range of ctxIdx	ROM Size
0 to 10	16 bits×11 words
11 to 59	16 bits×49×3 words
60 to 69	16 bits×10 words
70 to 398	16 bits×329×4 words

(ctxIdx=276 is for terminate mode and is skipped for initialization). First, when mode0 is terminate mode with bin0=1, the initialization address is reset to 0. Then, the coding mode is switched to initialization. In initialization mode, the address is counted up by one every cycle. When the address is counted up to 398, an initial_finish flag is set to 1. In the first stage, two signed 8-bit data m and n are read from the context initialization ROM by specifying the address and the Islice_flag, init_idc from slice_info buffer 2. Then, the m, n together with the address and sliceQP are buffered in register for second stage. In the second stage, the initial MPS and pstateIdx is calculated from the m, n and sliceQP. At end of this stage, the initial MPS and pstateIdx are written to the context memory with the writing address specified by the initialization address.

In the proposed implementation, the context ROM is implemented by 4 LUTs as shown in Table 3.7. Each LUT has a different indexing method based on the value of Islice_flag and init_idc. The context ROM contains 16×1484 bits in total.

Algorithm 8 Pseudo-code of Context Initialization

```
$preCtxState1 = m * sliceQP$
$preCtxState2 = preCtxState1[MSB : 4]$
$preCtxState3 = preCtxState2 + n$
$preCtxState = Max(1, Min(126, preCtxState3))$
**if** $preCtxState \le 63$ **then**
  $pstateIdx = 63 - preCtxState$
  $MPS = 0$
**else**
  $pstateIdx = preCtxState - 64$
  $MPS = 1$
**end if**
```

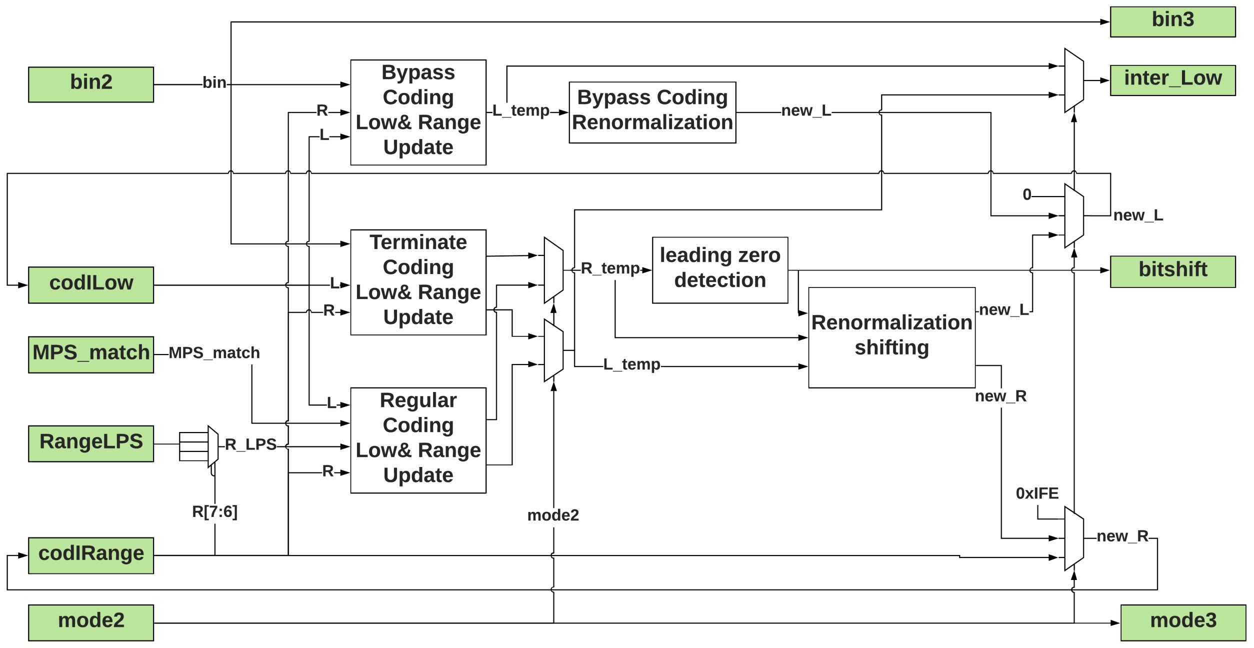


Figure 3.13: The block diagram of third stage in BAE

Algorithm 8 presents the derivation of the MPS and pstateIdx from ROM data m,n and sliceQP. In the algorithm, the intermediate value preCtxState1 is 15-bit signed value, preCtxState2 is 11-bit signed value, preCtxState3 is 12-bit signed value and preCtxState is 7-bit unsigned value.

3.4.2 Interval Update and Renormalization

The Fig. 3.13 shows the proposed design of the stage 3 in BAE. In this stage, the interval value 9-bit codIRange and 10-bit codILow are updated and renormalized through three different coding modes. The major data path includes the three interval updating blocks for bypass mode, terminate mode and regular mode, as well as the two paths of renormalization. The RangeLPS is the quantized rLPS for interval division in regular mode. Four RangeLPS are selected from the RangeLPS table by assigning pstateIdx in the previous stage. Then, one of the four is selected in this stage by the quantized Range R_Q, which is comprised of the second and third most significant bits of codIRange. Furthermore, during initialization of BAE, the initial value of codIRange and codILow is set to 510 and 0, respectively. According to Marpe's definition [7], the codIRange has valid range from 256 to 511, inclusive. However, the initial value (maximum valid value) of codIRange is set to 510 in H.264 standard [1].

The process of updating the codIRange and codILow for different coding mode, as well

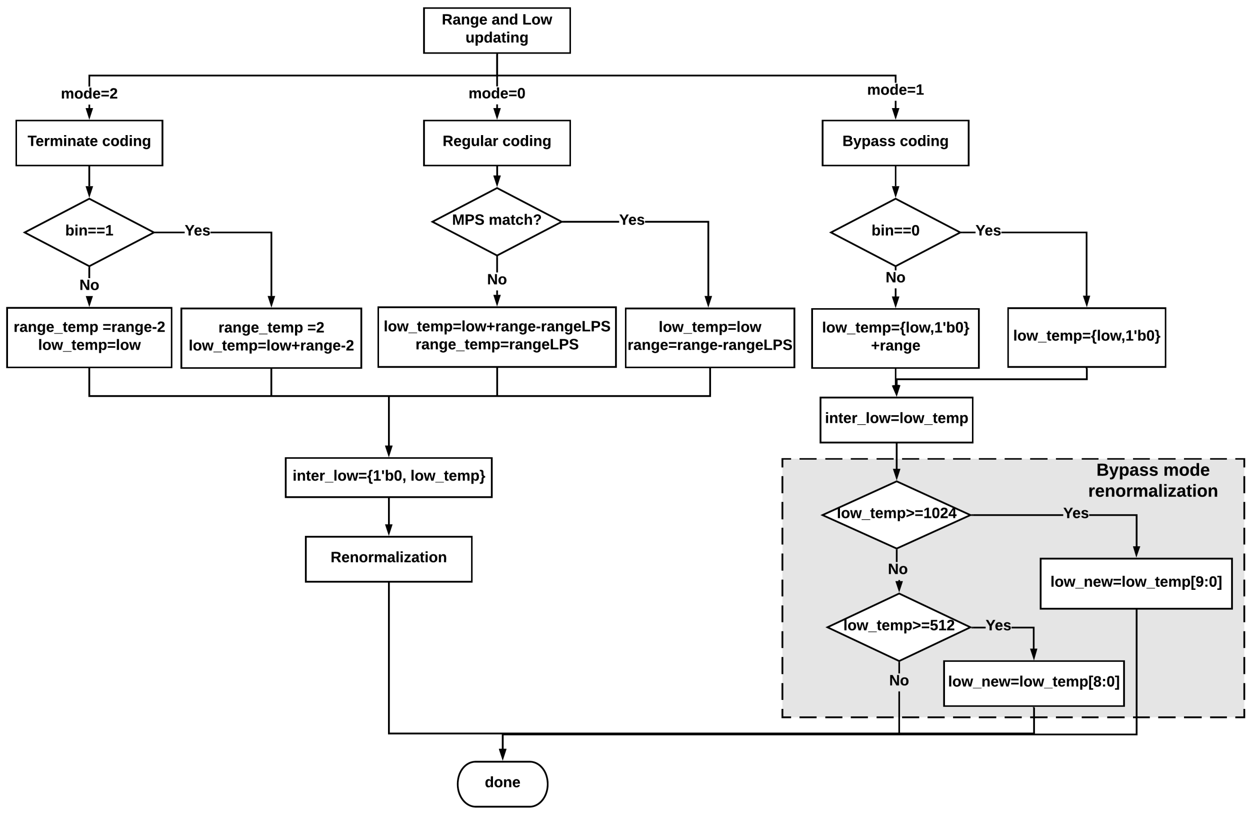


Figure 3.14: The process of updating Range and Low

as renormalization for bypass mode is implemented as shown in Fig. 3.14. The proposed interval updating process is derived from the description in section 9.3.4.1 in the standard [1]. In regular and terminate mode, if the updated codIRange is smaller than 256, it will be renormalized to the valid range from 256 to 510, inclusive. According to the table 9-33 in H.264 standard [1], the RangeLPS ranges from 2 to 240, inclusive. As a result, the pre-renormalized value Range_temp ranges from 2 to 508, inclusive. 9 bits is used for representing Range_temp. Normally in the updating process, the end-point Low keeps its value within 0 to 1023. However, in the special bypass mode, the intermediate Low_temp is possible to exceed 1023 when bin is not equal to 0. Therefore, 11 bits are used for representing Low_temp in bypass coding, while 10 bits are used for Low_temp in regular and terminate mode.

Renormalization

For regular and terminate mode, the renormalization process described in the standard is shown in Fig. 3.15. Because of the variable length of loop delay, the process described in the standard

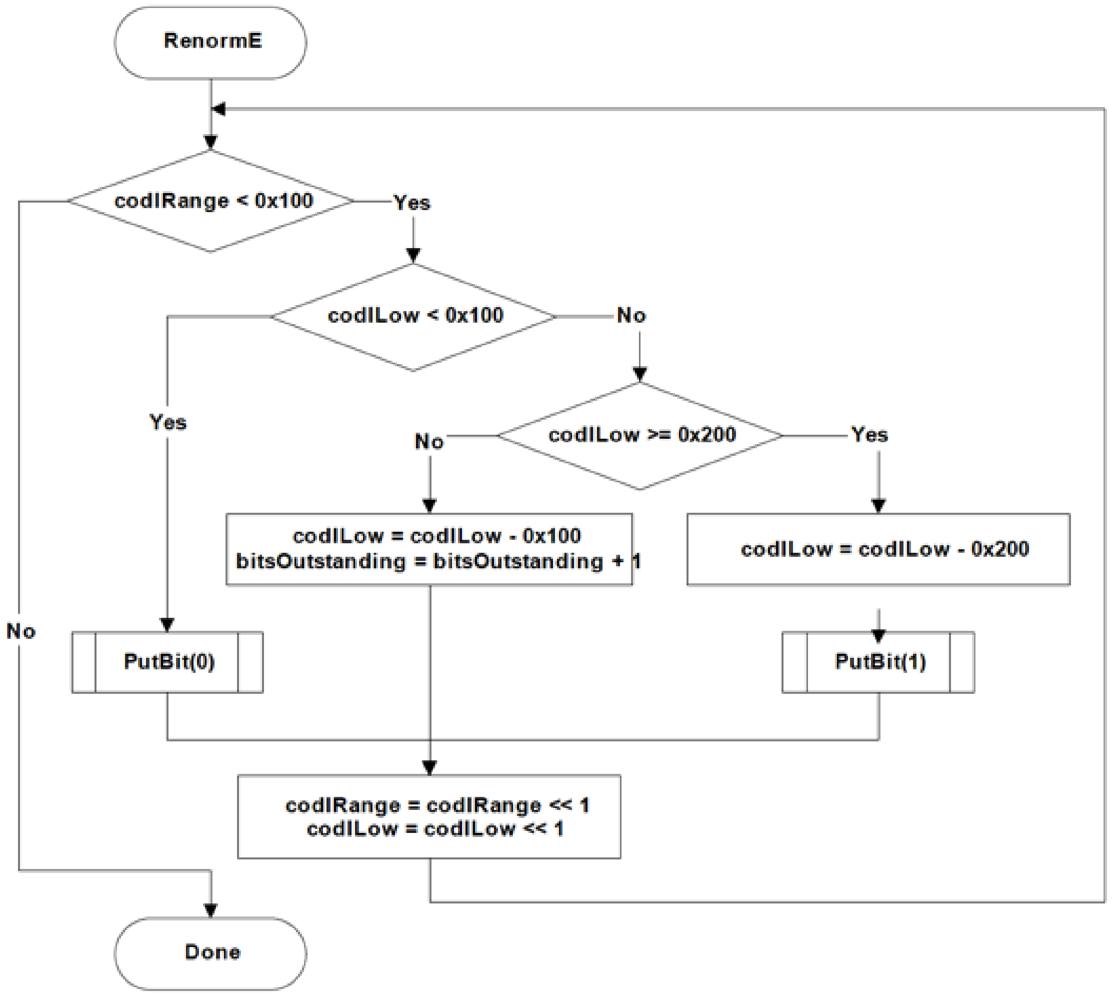


Figure 3.15: The process of renormalization described in H.264 Standard [1]

is not suitable for hardware implementation. In the proposed design the loop of renormalization is unrolled according to the bit-patterns of codIRange and codILow. The 9-bit codIRange is left shifted by one bit in every loop until the MSB is one. Therefore, the number of bits shifted is determined by the number of leading zeros in codIRange before renormalization. The fast renormalizaion of codIRange is implemented by a leading zero detection module and a 9-bit barrel shifter. The number of leading zero is recorded in *bitshift*, which ranges from 0 to 7. For codILow, A method of fast renormalization introduced in design [18] is implemented in proposed design: (1) left-shifting codILow by *bitshift* bit. (2) if the first *bitshift*+1 bits of the pre-shifted codILow are all 1s, set the MSB of renormalized codILow equal to 1. Otherwise, set the MSB equal to 0.

The process of bit generation and outstanding bit update shown in Fig. 3.15 is skipped in this stage. Therefore, the bits being shifted out in the renormalization of codILow is passed to the

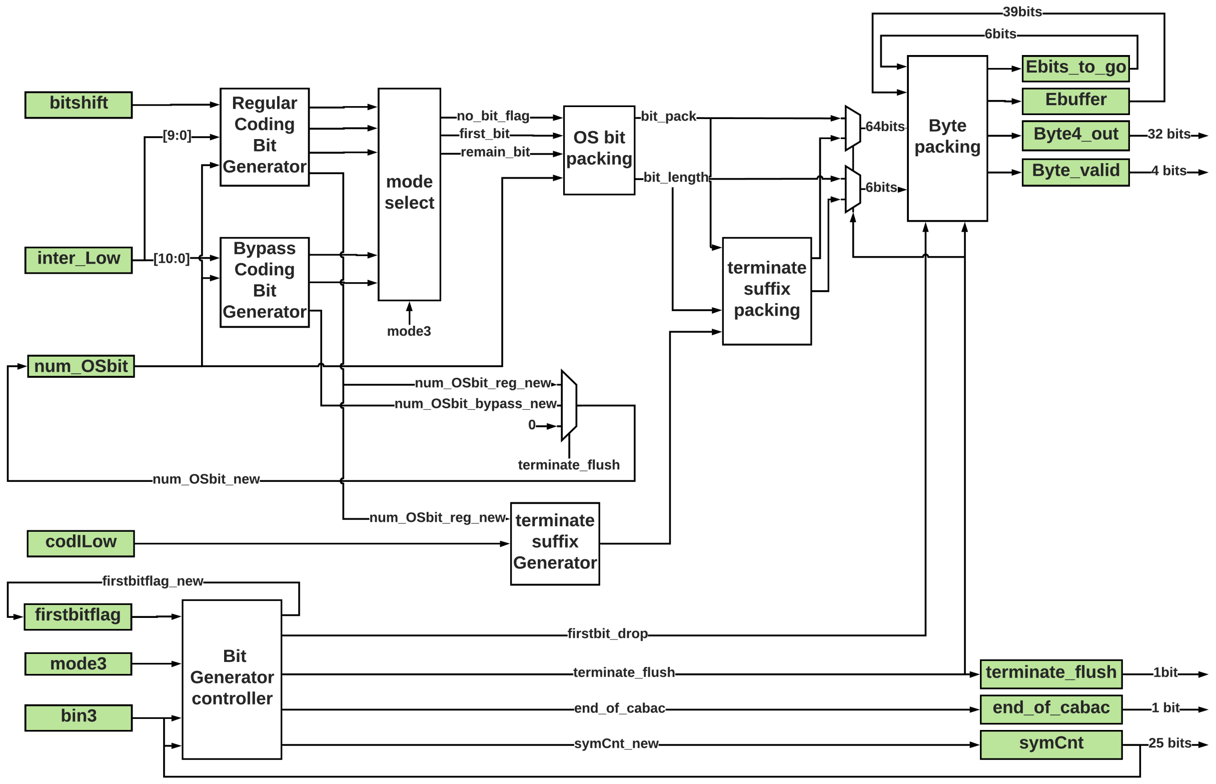


Figure 3.16: The block diagram of last stage in BAE

next stage for following output bit generation process. The 11-bit pre-renormalized value inter_Low together with the 3-bit parameter *bitshift* is buffered for the next stage.

3.4.3 Bit Generation and Packing

The block diagram for the last stage of BAE is shown in the Fig. 3.16. For regular mode, the regular coding bit generator works together with the outstanding (OS) bit packing unit. Bitstring for regular mode is generated based on bitshift, inter_low and number of remaining OS bits. For bypass mode, the bitstring is generated according to the inter_low and reaming OS bit. In terminate mode, if bin equals 1, the bitstring is comprised of a prefix from regular bitstring and a suffix from the terminate bit generator, otherwise, the bitstring for terminate mode is same as regular mode. After the bitstring is generated, they are packed into 0 to 4 bytes and written into a 32-bit output register. a 4-bit byte_valid signal is also generated to indicate the valid bytes in the register. The remaining bits that are not enough to form a new byte are buffered for the next cycle. According to the H.264 standard [1], if a bit is the first generated bit of current slice, this bit is dropped in BAE as it always equals zero. Furthermore, the terminate flush process of slice is invoked by mode3=2 with bin3=1. In this process, number of zeros are appended to the remaining bits to form a new byte. The detailed implementation of bit generation and packing is elaborated in following subsection.

The output register symCnt records the number of symbol (bin) coded in current slice. In the simulation of encoding 4K video at highest bit-rate specified in the standard, the number of symbol encoded in one slice is smaller than 20 million. Therefore, 25 bits is sufficient for the symCnt register. The output flag terminate_flush indicates the last cycle of one slice. if mode3 is equal to 3 for the consecutive three cycles after finish of encoding a slice, the PISO is inferred to be empty for three cycles after end of a slice. Hence, an end_of_cabac flag is set equal to 1.

Bitstring Generation

In order to generate output bitstring in one cycle, a fast bit generation for hardware is implemented in the proposed design. Fig. 3.17 shows the structure of output bitstring for three different coding mode.

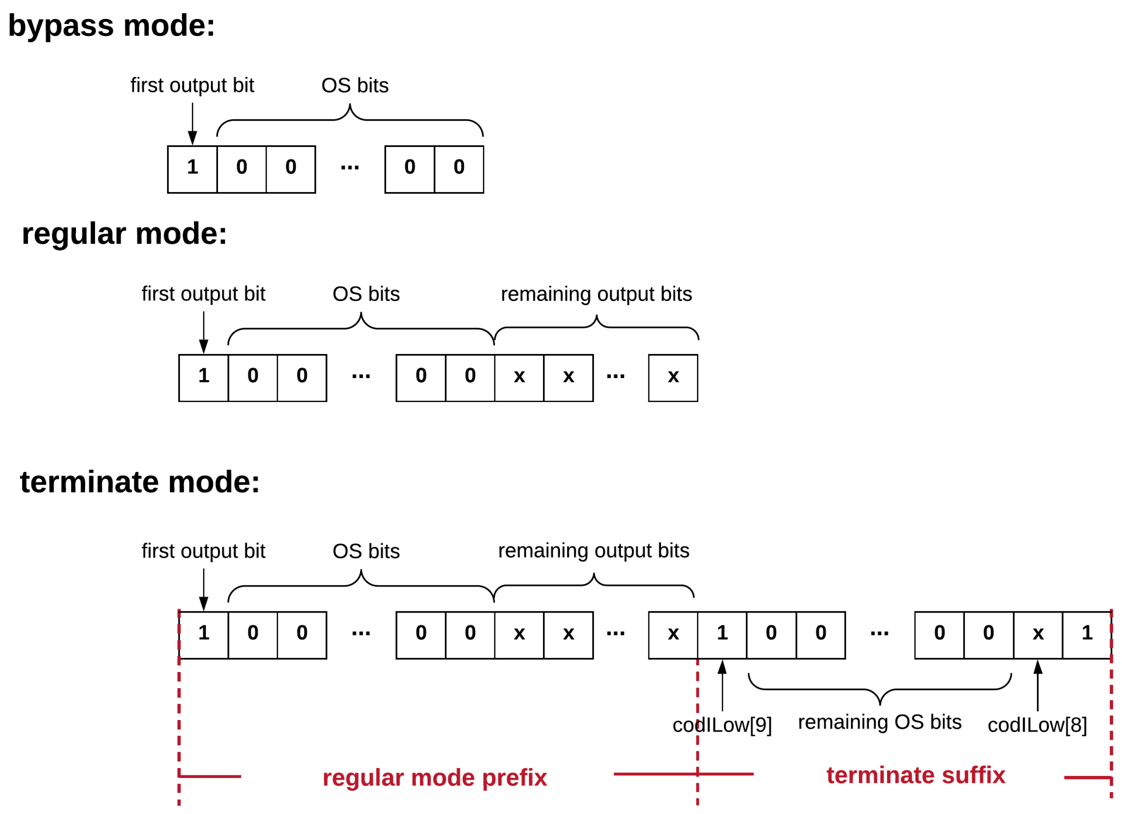


Figure 3.17: Output bitstring structure for different coding mode

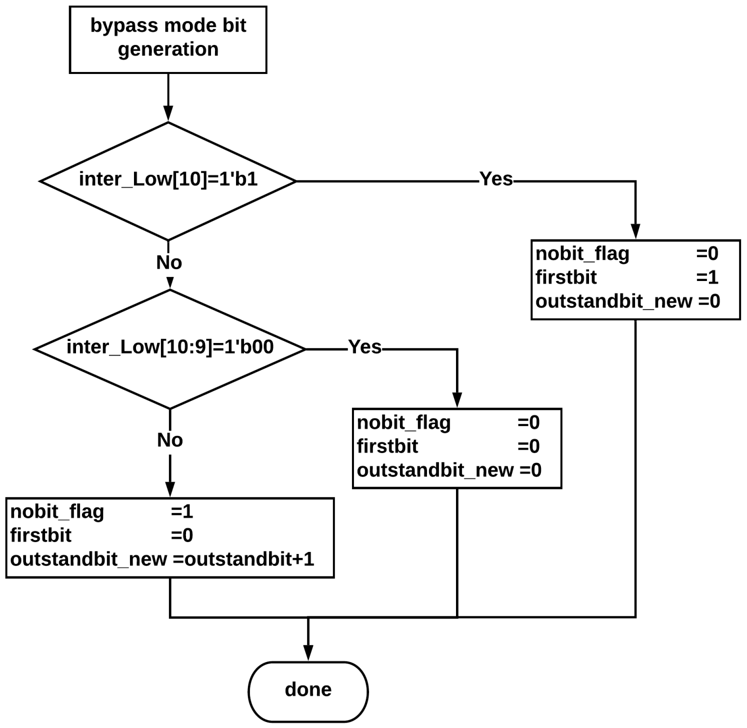


Figure 3.18: The process of bypass mode output bit generation

In bypass mode, the bitstring is comprised of the first output bit and the following OS bits. The number of OS bits is read from the register num_OSbit. The value of OS bits is always the inverted value of the first output bit, which is determined by inter_Low[10:9]. The process is shown in Fig. 3.18. The updated number of OS bit is written to register num_OSbit for next cycle.

In regular mode, additional output bits are appended to the first bit and OS bits. The process of generating the first and remaining bits for regular mode is shown in Fig. 3.19. A special module is implemented for detecting the position of the least significant zero in the first *bitshift*+1 bits of inter_Low[9:0] from MSB to LSB. The bits before the least significant zero including the first output bit and remaining output bits are packed to bitstring in this cycle, as shown in Fig. 3.17. The number of 1s after the least significant zero are recorded as number of OS bits for next cycle. Two special cases are considered: (1) If there is no zero detected in the *bitshift*+1 bits pattern, then the first *bitshift* bits are all output bits for this cycle. The first output bit and *bitshift*-1 remaining output bits are all equal to 1 and the number of OS bits for next cycle is equal to 0. (2) If the least significant zero locates at inter_Low[9] (the MSB), then no output bit is generated in this cycle, and the number of OS bits for next cycle is set equal to num_OSbit+*bitshift*. In case (2), a no_bit_flag is

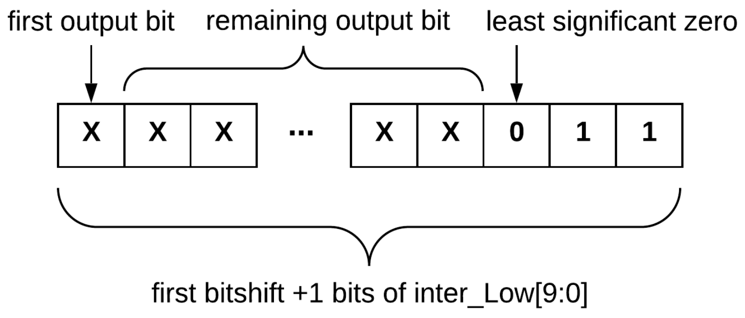


Figure 3.19: Output bit generation for regular mode

set equal to 1. Additionally, if *bitshift* equals 0, the no_bit_flag is also set equal to 1.

In terminate mode, the bitstring for terminate mode is a concatenation of the regular mode binstring and a terminate suffix, as shown in Fig. 3.17. The terminate suffix is only generated in the last cycle of a slice. The first bit of terminate suffix is codILow[9]. Following codILow[9], all the remaining OS bits are packed into the terminate suffix. At the end. 1-bit codILow[8] and 1-bit 1 are appended to the suffix.

As shown in Fig. 3.16, a shared OS bit packing module packs the first output bit, OS bits and remaining output bits together for both regular and bypass mode. A terminate bit generator packs the terminate suffix. The suffix is packed together with the regular mode bitstring before the byte packing. A 64-bit bitstring packet is generated each cycle. Compared to the initial architecture introduced in section 3.1, in which the bitstring for three coding mode is independently generated in three separate modules, the number of bit shifters used for bit packing is reduced in the proposed architecture. As long bit-length shifter is costly in hardware implementation, the area cost of the bit generation and packing module in BAE is significantly reduced.

The proposed design supports maximum 64 bits generated in one cycle. Maximum of 54 OS bits can be tolerated in the worst case when the *bitshift* equals 7, and additional 3 terminate bits (codILow[9], codILow[8] and the last bit one) are generated apart from OS bits. 54 OS bits is sufficient for very extreme case. The maximum number of OS bits observed in simulation is 34 bits in coding of 4K video. To increase the number of OS bits that can be tolerated, the size of bit-shifter and output register need to be increased. Therefore, a trade-off between OS bit length and circuit area occurs. The Ebuffer register shown in Fig. 3.16 buffers the remaining bits after byte packing process. The size of the buffer is considered in the design. In the worst case, there are 7 remaining bits from previous cycle and 64 bits generated in this cycle. 71 bits in total need to be packed into bytes. In this case, only 39 bits are needed for the Ebuffer, as the first 32 bits can be packed into 4 output bytes in one cycle. In the following cycle, 32 of the 39 remaining bits will be packed into bytes. The number of remaining bits is not going to exceed 39 in this process. Because long-length OS bits will not occurred in consecutive cycles. In fact, if 64-bit binstring is generated in this cycle, no more than 14 new bits will be generated in next two cycles.

Chapter 4

Functional Verification

The proposed CABAC encoder architecture is first implemented in register-transfer level (RTL) by Verilog hardware description language (HDL). The CABAC input vectors and reference output vectors extracted from the H.264 JM reference software [27] are used for the functional verification of the RTL implementation. The simulation tool used for functional verification is Cadence NCSim. After being verified in RTL, the proposed design is synthesized in 28 nm fully depleted silicon on insulator (FD-SOI) complementary metal–oxide–semiconductor (CMOS) technology by using Synopsys Design Compiler. Then the post-synthesis netlist is placed and routed in Cadence Innovus. Both the post-synthesis and post-layout gate-level netlist is verified by logic equivalence check in Synopsys Formality and gate-level simulation with the same test bench and test vectors as RTL. In this chapter, the methodology of verifying the function of CABAC encoder with the JM reference software is elaborated.

4.1 Overview of JM Reference Software

JM software is the H.264/AVC reference software developed by joint team of ISO/IEC MPEG and ITU-T VCEG who formulate the standard for H.264 video coding. Compared to other software implementation of H.264 encoder like X264, The JM software has more clear and complete coding procedure implemented as described in the standard. Therefore, the intermediate data, used as input and test vector of CABAC encoder, is more easily extracted from JM software. In the functional verification of the proposed CABAC encoder implementation, the JM version 8.6 is used for verifying the functional features required in main profile, level 5.1 of H.264 standard.

Table 4.1: JM encoder configurations for functional verification of CABAC encoder

Configuration Param.	Value	Description
ProfileIDC	77	66=baseline, 77=main, 88=extended
LevelIDC	51	51 = level 5.1
QPFirstFrame	0-51	QP for first IDR frame
QPRemainingFrame	0-51	QP for remaining I and P frames
QPBPicture	0-51	QP for B frames
IntraPeriod	2	Period of I-Frames
NumberBFrames	1	Number of B frames inserted
NumberReferenceFrames	1-16	Number of previous frames used for inter motion search
SymbolMode	1	Entropy coding method: 0=UVLC, 1=CABAC
ContextInitMethod	0	0: fixed, 1: adaptive
FixedModelNumber	0-2	specifies the cabc_init_idc
PicInterlace	0-2	Picture AFF: 0: frame coding 1: field coding 2: adaptive frame/field coding
MbInterlace	0-3	Macroblock AFF: 0: frame coding 1: field coding 2: adaptive frame/field coding 3: combined with PicInterlace=0 to do frame MBAFF
RDOptimization	0	Rate distortion optimized mode decision: 0: off, 1: on, 2: with losses

4.1.1 JM Encoder Configuration

Proper configurations of the JM H.264 encoder is set for the verification of CABAC encoder. The CABAC related configuration is shown in the Table 4.1. ProfileIDC is set equal to 77 for main profile and the LevelIDC is set equal to 51. In level 5.1, the highest bit-rate and the full range of MVD and coefficient levels are supported in JM software. The IntraPeriod=2 together with NumberBFrames=1 specifies the typical I-B-P-B group of picture (GOP). The context initialization index in CABAC is specified by ContextInitMethod and FixedModelNumber. The PicInterlace together with the MbInterlace specifies the MBAFF mode in the encoder. The rate distortion optimized (RDO) mode is set to off as it is not supported in the proposed design.

For comprehensive verification of the CABAC encoder implementation, various of configurations in JM encoder are applied, including frame/filed coding mode, different QP, different index for context initialization, different search range/number of reference in motion estimation and different types of GOP. Besides the typical I-B-P-B GOP, other GOPs such as I-B-B-P-B-B, are also tested. For encoding 4K (3840×2160) video, the number of previous frames for motion search should be set no larger than 5. Otherwise, the number of reference frame will exceeds the limit of the coded picture buffer in JM encoder, and the encoding process will be terminated.

4.2 Verification with JM Reference Software

The functional verification for the CABAC encoder implementation includes several verifying steps for each functional stage in the CABAC encoder. For each stage, corresponding intermediate data are extracted from the JM software and compared with the result of the implemented CABAC encoder.

4.2.1 Methodology

In step 1, the input syntax elements value and type together with corresponding neighboring macroblock information are extracted from JM software. Then, the binarizer of CABAC encoder is verified by comparing its output binstring and the binstring extracted from JM. In Step 2. The context modeler together with the first and second stage of BAE is verified. The ctxIdx result from the context modeler cannot be verified directly. Because the context indices adopted in the JM software is not the same as specified in the standard. Therefore, the next stage data MPS and pstateIdx are extracted from the JM for verification. The input bin value and coding mode of BAE is also extracted from JM for verification in step 2. In step 3, the interval value Range and Low are extracted for verifying the function of the updating and renormalization blocks in stage 3 of BAE. In the last step, the output byte stream of CABAC is extracted from the JM for verifying the bit generation and packing blocks in the final stage of BAE. By this step-by-step verification methodology, design flaws in each pipeline stage can be quickly figured out.

4.2.2 Design of Test Bench

Based on the methodology described above. The design of test bench for the CABAC encoder is illustrated in Fig. 4.1. In the coding process of JM, necessary parameters, input vectors,

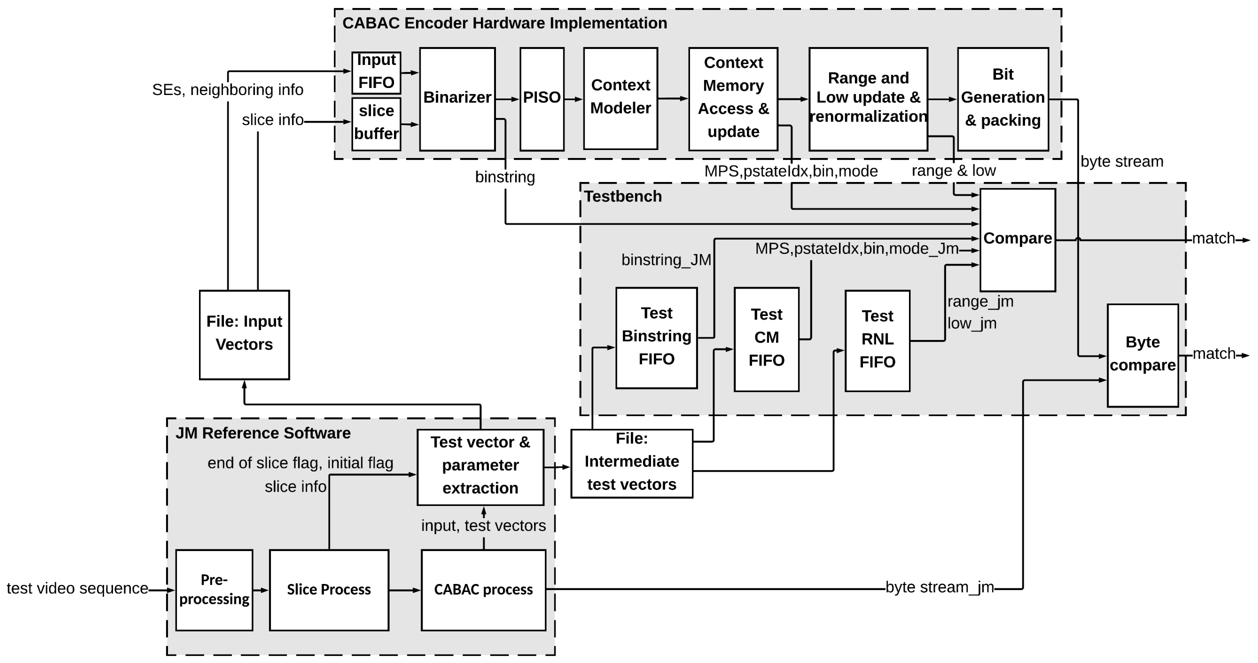


Figure 4.1: The block diagram of test bench

and test vectors are extracted to several text files. The test bench is implemented by several FIFO buffers and comparing modules. Test vectors of different stages are buffered in different FIFO. The dequeuing operation of the test binstring FIFO is synchronized with the input FIFO. The test CM FIFO is synchronized with the second stage of the BAE. The data compared in this stage are the context data MPS and pstateIdx, as well as the bin value and the mode. The test RNL FIFO is synchronized to the third stage of the BAE, where the renormalized interval Range and Low are verified. The byte comparison module is implemented at the end.

4.2.3 Extraction of Test Vectors

The CABAC related functions in the encoder of JM software is listed in the Table 4.2. The test vector extracted from each function is also specified. The test binstring can not be extracted directly as it is generated bit by bit. Special for-loops are implemented for concatenating these bits crossing different binarization functions during the Unary coding TU coding, UEGk coding process. Special functions are also implemented for concatenating the prefix and suffix bits for the binstring of SE_type, MVD and CBP. Furthermore, the extraction function is implemented independently for each syntax element type. It is easy to figure out design flaws for a specific syntax element in the proposed CABAC implementation, by applying the single syntax element test vector extraction.

Table 4.2: Extraction of test vectors from JM reference software

File	Function Name	Test Vectors or Prama. Extracted
slice.c	terminate_slice	end_of_slice =1
context_init.c	init_contexts	Slice information: 1. Slice type 2. MbaffFrameFlag 3. Slice QP 4. cabac_init_idc
macroblock.c	write_one_macroblock	end_of_slice =0
		Sub-macroblock Index
biariencode.c	biari_encode_symbol	MPS, pstateIdx, range, low for regular coding
	biari_encode_symbol_eq_prob	MPS, pstateIdx, range, low for bypass coding
	biari_encode_symbol_final	MPS, pstateIdx, range, low for terminate coding
	put_byte	Output bytes of CABAC
	put_one_bit	
	put_one_bit_plus_outstanding	
cabac.c	writeFieldModeInfo_CABAC	1. Input syntax element value 2. Input syntax element type 3. Corresponding syntax element and info of neighbor macroblocks 4. Partition index for MVD, ref_idx_lX, CBF 5. Prediction direction for MVD, ref_idx_lX 6. iCbCr for CBF
	writeMB_skip_flagInfo_CABAC	
	writeMB_typeInfo_CABAC	
	writeB8_typeInfo_CABAC	
	writeIntraPredMode_CABAC	
	writeRefFrame_CABAC	
	writeDquant_CABAC	
	writeMVD_CABAC	
	writeCIPredMode_CABAC	
	writeCBP_BIT_CABAC	
	writeCBP_CABAC	
	write_and_store_CBP_block_bit	
	write_significance_map	
	write_significant_coefficients	
	writeRunLevel_CABAC	ctxBlockCat

4.2.4 Video Sequences for Verification

Raw YUV-sampled video sequence is used as input of the JM H.264 encoder. The video sequences used in the functional verification are listed in Table 4.3. All of these sequences are sampled in 4:2:0 YUV format. Different video resolution are used to verified different syntax element value ranges in CABAC encoder. For example, the full range of MVD specified in the level 5.1 of the H.264 standard can only be verified with quad full high definition (QFHD, or 4K UHD)

Table 4.3: Test video sequences for CABAC encoder functional verification. Sequences Akiyo, Bus, Stockholm and Blue sky are from Xiph.org [28], sequences ShakeNDry, YachtRide, ReadySetGo and Bosphorus are from Ultra Video Group [29].

Name of Sequence	Resolution	Format
Akiyo	176×144	QCIF 4:2:0
Bus	352×288	CIF 4:2:0
Stockholm	1080×720	HD 4:2:0
Blue sky	1920×1088	FHD 4:2:0
ShakeNDry	3840×2160	QFHD 4:2:0
YachtRide	3840×2160	QFHD 4:2:0
ReadySetGo	3840×2160	QFHD 4:2:0
Bosphorus	3840×2160	QFHD 4:2:0

sequence. Furthermore, for frame coding, the width and height of picture must be multiple of 16 (width of macroblock). Hence, the full high definition (FHD) 1920×1080 video sequence is scaled to 1920×1088 in JM encoder. For field coding, the height of picture must be multiple of 32 (double height of macroblock). Therefore, only the common intermediate format (CIF) and FHD sequence is suitable for testing in these cases.

Chapter 5

Experimental Result

The proposed design of CABAC encoder is implemented in 28 nm FD-SOI CMOS technology. The circuit performance and cost of the presented implementation is evaluated at both post-synthesis and post-layout stage. The experimental results of synthesis and physical design, including circuit area, throughput and power consumption, are presented and analyzed in this chapter, and a comparison with previous CABAC encoder designs is given.

5.1 Synthesis Result

5.1.1 Throughput

The throughput of CABAC encoder is measured in bins per second, which depends on symbol encoding rate in bin/cycle and maximum clock frequency of the circuit. In coding process, the proposed design is capable of a sustained consistent coding rate of 1 bin/cycle. However, due to the stalls for initialization between encoding two slices, the actual encoding rate is slightly lower than 1. In simulation of encoding sequences at 30 fps, typical QP, with resolution from 352×288 to 3840×2160, the symbol encoding rate ranges from 0.99869 to 0.99996 bin/cycle. The encoding rate increases when the video resolution increases. As the targeting application of the proposed implementation is 4K UHD video coding, the impact of initialization stall can be ignored and approximate 1 bin/cycle encoding rate is achieved in the design.

The proposed implementation is synthesized using typical-typical corner devices in a 28 nm standard cell library. The synthesis result of maximum clock frequency at different operating voltage

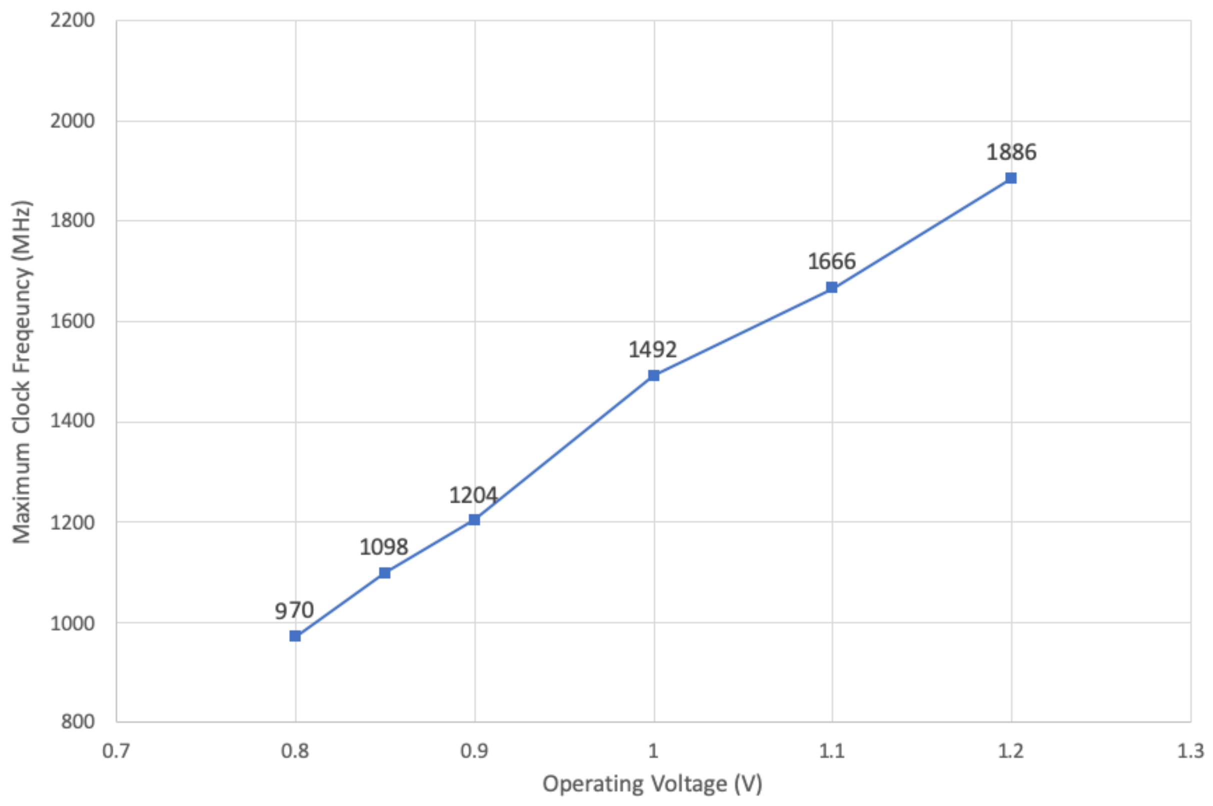


Figure 5.1: Maximum clock frequency of the CABAC encoder

is shown in Fig. 5.1. As the proposed implementation achieves symbol encoding rate of 1 bin/cycle and maximum clock frequency of 1,886 MHz after synthesis, the maximum overall throughput is 1,886 million bins per second (Mbin/s) at 1.2 V.

In practice, the throughput requirement for CABAC encoder depends on the H.264 coding bit-rate which varies in coding different video sequences. In the process of output bit generation in BAE, the processed bins will be converted to output bits. According to Marpe [30], a typical ratio of bin to output bit in CABAC is 1.3. In order to encode 4K UHD video sequence in real time, the CABAC encoder needs to support the highest bit-rate specified in the H.264 standard. As shown in Table 5.1, four 4K UHD video sequences are encoded in the CABAC encoder at 30 fps to test the real-time throughput requirement in bin/second. As specified in level 5.1 of H.264 standard, the maximum video bit-rate supported is 240 million bits per second (Mbps), which is sufficient for encoding 4K UHD video at 30 fps. Therefore, the QP value for the four sequences is set to the lowest possible value that makes the bit-rate close to 240 Mbps in the simulation. The number of bins being processed during the CABAC coding process is counted. Based on the simulation result, the average

1.3 bins to output bit ratio is verified, and the throughput requirement for CABAC encoder is not larger than 320 Mbin/s. According to the results at 30 fps, the throughput requirement for 60 fps 4K UHD video coding is inferred to be no larger than 640 Mbin/s. Therefore, the post-synthesis throughput of 970 Mbin/s at 0.8 V is already sufficient for encoding the 4K UHD sequence in real time at 60 fps. In order to save both dynamic and leakage power, the post-synthesis netlist at 0.8 V is chosen instead of typical 1.0 V, for following physical design of the CABAC encoder.

Table 5.1: Throughput requirement for encoding different 4K UHD video sequences

Sequence Name	Resolution	QP	Bit Rate @ 30 fps (Mbps)	Throughput Required (Mbin/s)	Bin to Output Bit Ratio
Bosphorus	3840×2160	14	238.5	312.9	1.3120
ReadySetGo	3840×2160	17	204.5	261.9	1.2808
ShakeNDry	3840×2160	19	235.2	302.9	1.2880
YachtRide	3840×2160	15	229.0	298.5	1.3035

5.1.2 Circuit Area

The synthesis result of area is shown in Table 5.2. The post-synthesis area is measured in gate count of equivalent minimum NAND gate in the library. In the proposed design, the RAMs used in FIFO buffers and the context memory are implemented in flip-flops. Therefore, the gate count for the RAMs is also given. The gate counts of RAM includes only the memory cost. The area of combinational interface for the context memory and FIFOs is not included in the RAM area shown in the table.

Excluding the area for memory cost, the area utilization of each module in the CABAC encoder is shown in Fig. 5.2. 56.89% of the total area is occupied by the BAE. In BAE, the context ROM and the bit generating & packing module occupies major portion of area. The module for bit generating & packing consumes 41.9% area of BAE, because of the several long-bit-length barrel shifters utilized in this module. The 16×1484 bits ROM consumes another 33.29% area of BAE. Besides BAE, the context modeler occupies another 29.18% of the whole area, due to the big registers used for storing coded syntax elements of current macroblock.

Table 5.2: Circuit area of each functional module in the CABAC encoder

Module Name		Area in Gate Count
Binarizer		3,256
Context Modeler		6,822
BAE	Total	13,300
	Bit Generating & Packing	5,574
	Context ROM	4,429
Total (exclude RAM)		23,378
RAM	Context Memory	24,583
	FIFO	5,259
Total		53,219

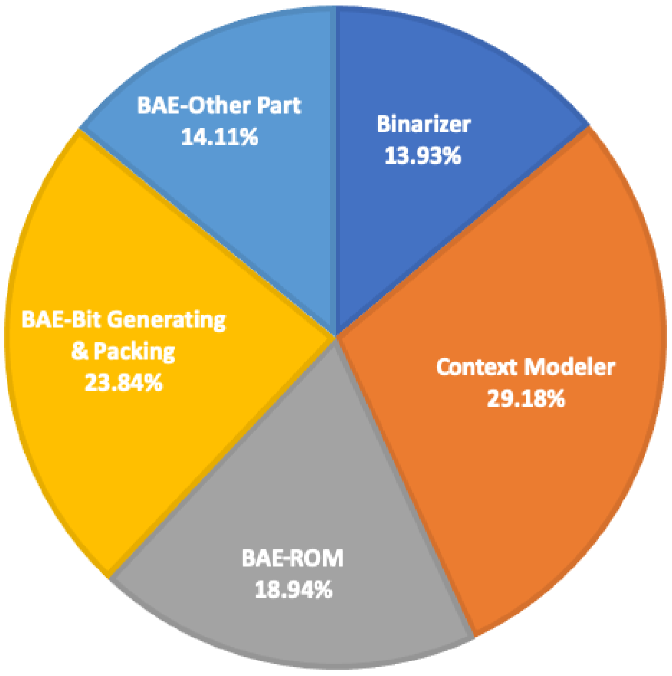


Figure 5.2: Area utilization of the CABAC encoder

5.1.3 Improvement Compared to the Initial Architecture

An initial version of CABAC encoder is implemented based on the preliminary architecture introduced in section 3.1. In this initial version, the context modeling procedure is not divided into two pipeline stages, and the bit generating and packing stage in BAE includes three independent modules for regular, bypass and terminate coding mode.

In the final implementation, the two-stage context modeling architecture is adopted. By splitting the context modeler, the FIFO size between first and second pipeline stage is reduced from 65 bits to 28 bits. The length of critical path in second pipeline stage is also reduced. As shown in Table 5.3, the throughput of the final design at 0.8 V increases by 10.60% compared to the initial version. The area for total FIFO RAM decreases by 20.68%.

In the initial architecture, three independent sub-modules are adopted for bit generation of regular, bypass and terminate coding mode. In the final version, the generation of bitstring is implemented by concatenating the first bit, OS bits and remaining bits for each coding mode, and several bit packing sub-modules are shared by the three coding modes. Therefore, compared with the initial version, the number of barrel shifter used is reduced. As a result, the area of bit generating & packing module is reduced by 17.51% in the final design.

Table 5.3: Comparison between the initial and final version

Version	Technology	Maximum Clock Frequency (MHz)	Area in Gate Count	
			FIFO RAM	Bit Generating & Packing
Initial	28 nm FD-SOI @ 0.8 V	877	6,630	6,757
Final		970	5,259	5,574

5.2 Physical Design Result

The proposed CABAC encoder is placed and routed using a 28 nm standard cell library. The results of the physical design are summarized in Table 5.4. The placement of standard cells is shown in Fig. 5.3, while the routing of circuit is shown in Fig. 5.4. The clock tree distribution network is illustrated in Fig. 5.5. The maximum clock frequency and power dissipation is evaluated at a supply voltage of 0.8 V and a typical-case corner (typical device, typical temperature, nominal RC parasitic). The post-layout maximum clock frequency of the CABAC encoder is 769 MHz. Hence the CABAC encoder implementation achieves 769 Mbin/s throughput which is sufficient for encoding 4K UHD (3840×2160) video in real time at 60 fps. Due to confidentiality concerns, only the central region of the layout is presented.

To evaluate the power dissipation of the proposed circuit layout, a VCD file is generated from the post-layout gate-level simulation with test video sequence. The VCD file records switching

Table 5.4: Layout of the CABAC encoder at a glance

Supply Voltage	0.8 V
Clock Frequency	769 MHz
Power	13.75 mW
Core Area	33,411 μm^2
Core Dimension	219.23 $\mu m \times 152.40$ μm
Core Utilization	86.28%

activity of each node in the post-layout netlist during the simulation. With the switching activity back-annotated by the VCD file, the time-based power consumption can be evaluated in Synopsys PrimePower. The post-layout power dissipation result for maximum clock frequency and real-time applications of encoding 1080p and 4K UHD sequence is shown in Table 5.5. For the targeting applications, the power is scaled to 640 MHz, 320 MHz and 200 MHz, respectively. The power dissipation for 4K UHD real-time coding at 60 fps is 11.48 mW. Furthermore, in average 81.47% of the total power is consumed by the clock tree and registers. Therefore, the future power optimization should focus on reducing the context memory access frequency and applying clock gating technique on registers.

Table 5.5: Post-Layout power dissipation for different application

Application	Clock Frequency (MHz)	Power (mW) @ 0.8 V		
		Total	Dynamic	Leakage
Maximum Clock Frequency	769	13.75	13.51	0.24
4K UHD @ 60 fps	640	11.48	11.24	0.24
4K UHD @ 30 fps	320	5.86	5.62	0.24
1080p @ 75 fps	200	3.75	3.51	0.24

Figure 5.3: Standard cell placement in the central region of the CABAC encoder layout

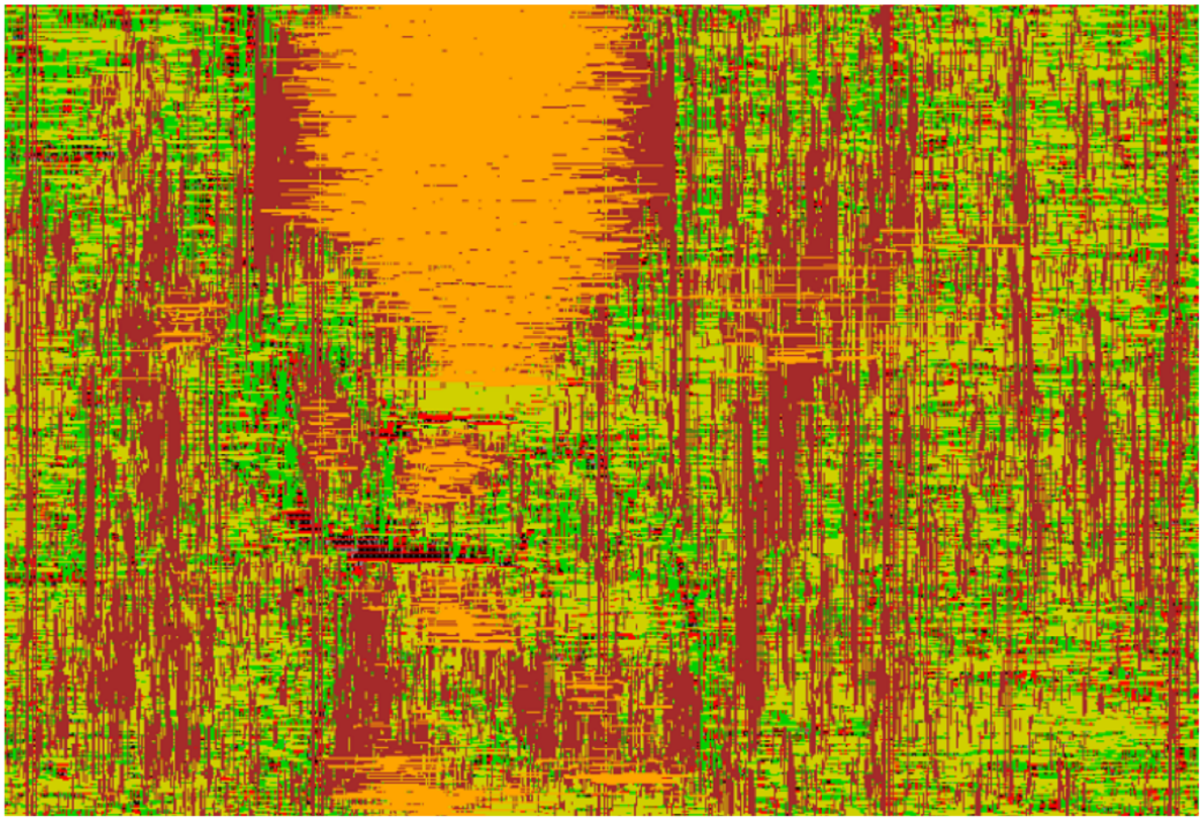


Figure 5.4: Routing in the central region of the CABAC encoder layout

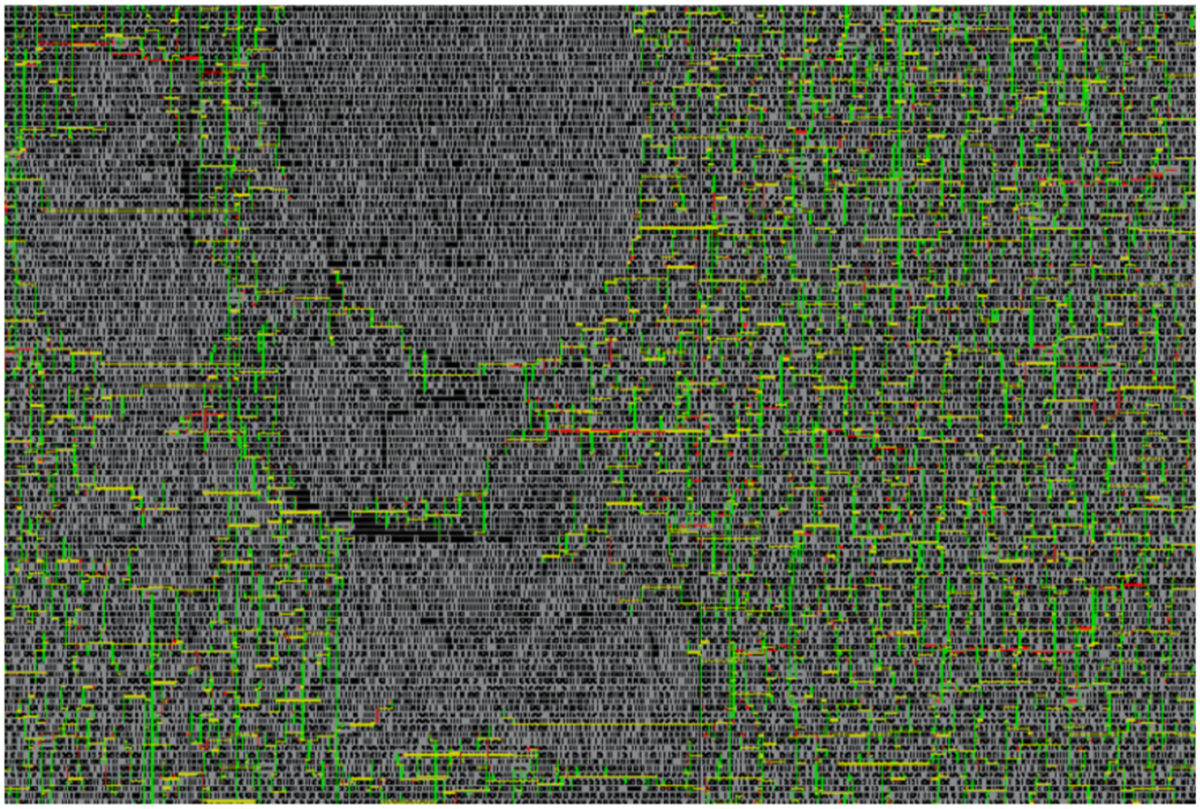


Figure 5.5: Clock tree distribution network in the central region of the CABAC encoder layout

5.3 Comparison with Previous Work

5.3.1 Comparison of Throughput

The comparison of throughput with previous work is shown in Table 5.6. The symbol encoding rate in bin per cycle is determined by the architecture design. Compared to the previous design, the proposed implementation achieves higher bin/cycle rate than design [15], [16], [31] and [18]. Design [9], [10], [12], [14], [11] and [13] have higher bin/cycle rate than the proposed design. However the hardware cost is also higher in these designs. Furthermore, the maximum clock frequency is determined by both architecture and technology in which the CABAC encoder is implemented, and the throughput in bins per second depends on both bin/cycle rate and maximum clock frequency. Compared to previous work. the proposed CABAC encoder has the highest absolute post-synthesis throughput in MBin/s when synthesized at 1.2 V. However, if the previous work were implemented in 28 nm process at 1.2 V, the advantage of throughput over them does not retain.

Table 5.6: Throughput comparison with previous work of CABAC encoder

Design	Technology	Throughput (MBin/s)	Maximum Clock Frequency (MHz)	Bin/Cycle
Shojania [15]	0.18 µm	87	263	0.33
Li [16]	0.35 µm	80	150	0.53
Kuo [17]	0.18 µm	200	200	1
Osorio [9]	0.35 µm	350-428	186	1.9-2.3
Chen [31]	0.15 µm	216	333	0.56
Liu [18]	0.13 µm	134	200	0.67
Tian [25]	0.13 µm	620	620	1
Tian [19]	0.13 µm	578 (328*)	578	1
Chen [10]	0.13 µm	315	222	1.42
Liu [12]	90 nm	476	238	2
Fei [14]	90 nm	1116	279	4
Tsai [11]	0.13 µm	1191	254	**4.69**
Zhou [13]	65 nm	1409	330	4.27
This work	28 nm @ 1.2 V	**1886**	**1886**	1
	28 nm @ 1.1 V	1666	1666	
	28 nm @ 1.0 V	1492	1492	
	28 nm @ 0.9 V	1204	1204	
	28 nm @ 0.8 V	970	970	
This work post layout	28 nm @ 0.8 V	**769**	**769**	

* Post-layout maximum throughput.

5.3.2 Comparison of Area in Gate Count

The area comparison in gate count with previous work is shown in Table 5.7. In most of the previous design, the memory of context and FIFO is implemented by on-chip SRAM, whose area is not counted in the total gate count. Therefore, the area of memory cost in the proposed design is also excluded for following comparison.

The area of each design is difficult to be compared directly as each design has different functional parts implemented as shown in the table. The result shows that the proposed design of this thesis has relatively complete function for main profile H.264 CABAC encoder in hardware. Design [15] and [16] leave the binarization and ctxIdxInc calculation finished in host processor. Therefore, the gate count of them are reasonably smaller than the proposed design. Design [10], [12], [14], [11] and [13] has reasonably higher gate counts due to their multi-bin architecture. Design [9] is also capable to process multiple bins in one cycle. However the ctxIdxInc calculation is not implemented in this design and the binarizaiton in this design supports only syntax elements related to residual data. As a result, it has smaller area than other multi-bin design. In addition, the byte packing function is not included in the BAE of [9]. Hence the area occupied by its BAE is similar to area of the single-bin BAE in the proposed design.

Design [18], [25] and [19] supports similar function of binarization, ctxIdxInc calculation and BAE as the proposed design of this thesis. However, these design has much larger area in gate count, because an additional neighbor memory is implemented. As introduced in section 2.5, the neighbor memory reduces the process in host processor and makes the CABAC encoder more plug-able in H.264 hardware system. But the additional memory access logic circuit and buffers consume large circuit area. Furthermore, design [25] and [19] supports additional RDO function. The RDO function also adds additional area to the circuit.

Comparing area of the BAE is more clear than the whole CABAC encoder. Compared to the BAE in design [25], the area of BAE is smaller in the proposed design. Furthermore, the bit generating and packing module reported in [25] has gate count of 9472 which is 1.7× larger than the gate count of same part in the proposed design. BAE in [18], [16] and [15] has smaller area than the proposed design. Because no byte packing function is implemented in these design.

Table 5.7: Comparison of area in gate count and functional part implemented with previous work of CABAC encoder

Design	Bin/Cycle	Area in Gate Count			Functional Parts Implemented					
		RAM	Total (Exclude Memory)	BAE	Binarization	Modeler (ctxIdxInc Calculation)	BAE	Byte Packing In BAE	Neighbor Memory	RDO Support
Shojania [15]	0.33	10K Bits	933	933	No	No	Yes	No	No	No
li [16]	0.53	4.1K Bits	4.57K	4.57K	No	No	Yes	No	No	No
Osorio [9]	1.9-2.3	5.6K Bits	19.4K	12.9K	Partial	No	Yes	No	No	No
Chen [31]	0.56	-	13.3K	-	Yes	No	Yes	Yes	No	No
liu [18]	0.67	-	34.3K	5.1K	Yes	Yes	Yes	No	Yes	No
Tian [25]	1	8.6K Bits	27.5K	14.5K	Yes	No	Yes	Yes	No	Yes
Tian [19]	1	-	44.6K	-	Yes	Yes	Yes	Yes	Yes	Yes
Chen [10]	1.42	-	40.8K	14.7K	Yes	Yes	Yes	Yes	Yes	No
Liu [12]	2	-	33.9K	24.4K	Yes	Yes	Yes	Yes	No	No
Fei [14]	4	-	36.2K	-	Yes	No	Yes	-	No	No
Tsai [11]	4.69	-	42.5K	26.9K	Yes	Yes	Yes	-	No	No
Zhou [13]	4.27	-	57.3K	-	Yes	No	Yes	-	No	No
This Work	1	3.4K Bits	23.4K	13.3K	Yes	Yes	Yes	Yes	No	No

- Data not available.

Chapter 6

Thesis Summary and Future Work

6.1 Thesis Summary

In this thesis, an low-area-cost high-throughput design of CABAC encoder for H.264 main profile is implemented. The 6-stage pipeline architecture of the CABAC encoder achieves 1 bin/second symbol processing rate, and supports the full function of context initialization, binarization, context modeling and binary arithmetic coding in hardware. The post-layout implementation is shown to be capable of encoding 4K UHD video at 60 fps in real time.

Several design issues of CABAC encoder in hardware are discussed in this thesis, including (1) handling the different symbol processing rate between binarization and BAE; (2) unrolling loops in the original algorithm of UEGk coding, interval renormalization and bit generation described in H.264 standard; (3) data value range and binstring format in binarization. The solutions for first two issues are presented in the thesis while the third issue is analyzed based on H.264 standard. In addition, several optimization, including splitting the CM and re-organizing the bit generating and packing in BAE, is applied to achieve higher throughput and lower area cost in the proposed design.

6.2 Future Work

6.2.1 Architecture with Neighbor Macroblock Memory

In order to integrate the proposed CABAC encoder into some specific H.264 platforms, additional pre-processing of neighbor macroblock information is needed. For example, design [32] is

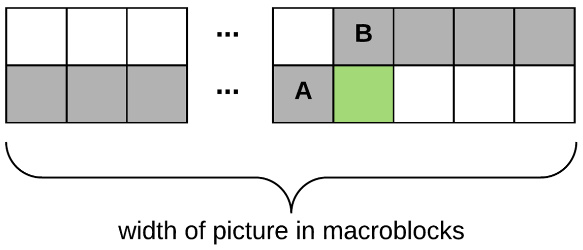


Figure 6.1: Coded macroblocks stored for context modeling

a baseline H.264 encoder implemented on a many-core computational platform called Asynchronous Array of Simple Processors (ASAP). In this H.264 encoder, only necessary macroblock data for baseline function is stored in memory, including motion vectors, non-zero coefficients and parameters for managing current/reference frame. In order to add CABAC function to this base-line H.264 encoder, the CABAC process needs to buffer and manage the neighboring macroblock reference data by itself. The pluggability of the proposed CABAC encoder can be enhanced by adopting a neighbor macroblock memory as introduced in design [20], [19], [18] and [10]. Based on the analysis and design of neighbor memory in [19] and [20], a preliminary architecture for the neighbor macroblock memory interface is discussed in this subsection.

As shown in Fig. 6.1, the minimum number of coded macroblocks to be stored in the memory is equal to the width of picture in macroblocks. If the targeting application is 4K UHD video coding, at least 240 macroblocks need to be stored. Only the data used as upper neighbor reference need to be stored in this memory. The reference data of left neighbor macroblock (previous macroblock) can be stored in a local register which is updated every time coding a macroblock is accomplished. As a result, the size of the memory can be significantly reduced.

For each macroblock, syntax elements including mb_field_coding_flag, mb_type, mb_skip, MVD, ref_idx, ICPM, CBP and CBF, need to be stored. Fig. 6.2 (a) shows the syntax element data to be stored at macroblock level, total 12 bits are stored at this level. Fig. 6.2 (b) presents the syntax element data to be stored at 8×8 block level, total 8×2 bits are stored as upper reference. Although direct_mode_flag is not an syntax element, it is also stored for context modeling of ref_idx. Fig. 6.2 (c) illustrate the syntax element data to be stored at 4×4 block level, 7×4+1 bits are stored for each upper reference 4×4 block. Therefore, 144 bits in total need to be stored for each upper neighbor reference in the memory. The local register for previous macroblock is also 144 bits. An additional 144-bit register is need for storing coded syntax elements of current macroblock,

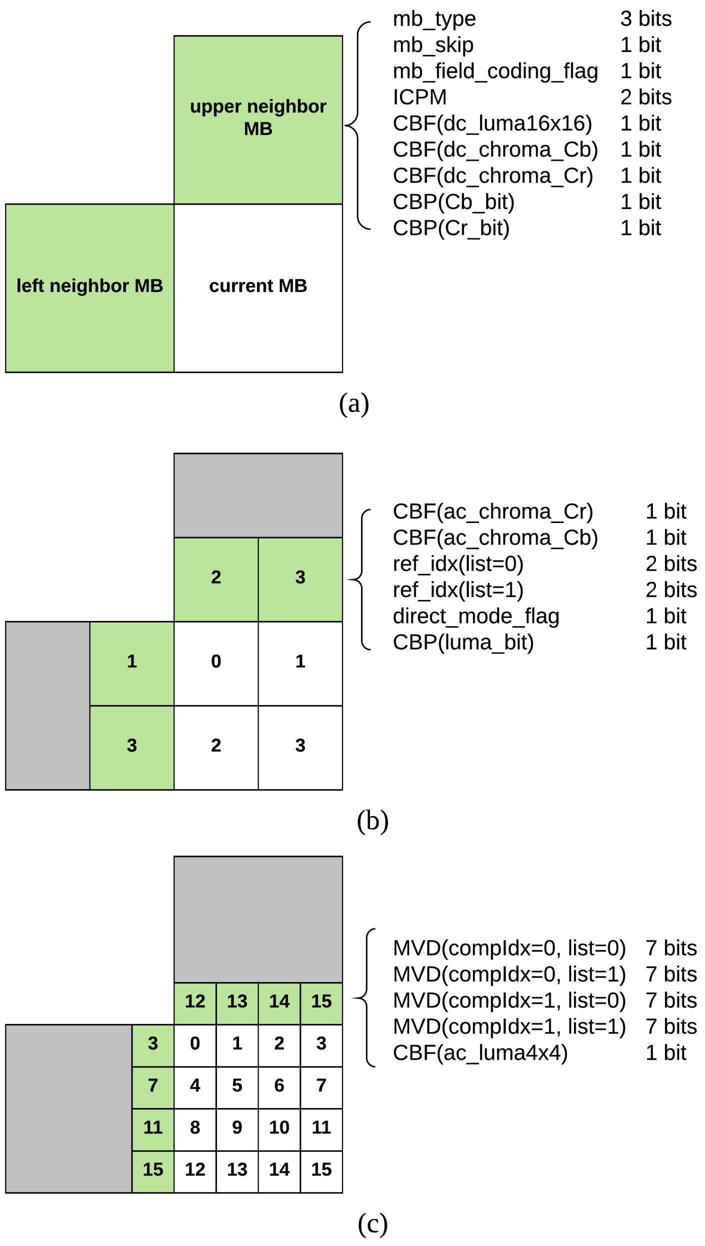


Figure 6.2: Data to be stored in each macroblock

when coding of current macroblock is finished, the data of this register is written into the neighbor macroblock memory and also the register for previous macroblock. In conclusion, 240×144 bits memory and two 144-bit registers are needed in this new architecture.

Furthermore, one additional unit is needed for mapping the index of current coding partition to the index of neighbor partition outside current macroblock, as shown in Fig. 6.2 (b) and (c). Another unit is needed for counting up the index of macroblock in one slice. According to the index of macroblock and the width of picture in macroblocks, the position of macroblock can be derived. Based on the position of macroblock, the availability of each neighbor macroblock is derived.

6.2.2 Strategy of Power Reduction

Clock Gating

In synchronous circuits, the clock signal switches at every cycle and drives a large capacitance in the circuit, and large portions of power is dissipated by the circuit driven by clock signal. By adopting clock gating technique, part of the clock tree can be turned off without affecting the circuit function. Therefore, clock gating is an effective and widely-used technique for dynamic power saving [33]. In the proposed CABAC implementation, as presented in section 5.2, a large portion of power is dissipated by the context memory and clock tree network. Hence, clock gating should be a promising method of power optimization in CABAC.

Clock gating can be implemented at both RTL and synthesis level. Overall higher effectiveness of clock gating can be achieved in synthesis level, as the enable signal of clock gating can be searched through different levels of hierarchy during synthesizing the circuit. RTL clock gating is used for optimizing a particular sequential module such as memory, the design of clock gating in RTL is more customized.

The effectiveness of clock gating is also largely depends on the active ratio of registers and access frequency of memory. To get better optimization by clock gating, the context memory access frequency need to be reduced by some additional optimization.

Context Memory Re-allocating

The access of context memory is already disabled when the coding mode is not regular mode in the proposed design. In regular mode, the memory access frequency is also reduced when two consecutive address is the same. According to design [17] and [19], the memory access frequency can be further reduced by applying a memory-cache hierarchy in BAE. In each cycle, multiple context models are read from the context memory and buffered in a cache. If following context model to be read is in the cache buffer, the access of memory is not needed. Combining clock gating and context model prefetching, the switching power is saved. In this architecture, multiple context models are packed into one word of the memory, and assigned with the same address. A LUT is needed for mapping the input ctxIdx to the memory address. The effectiveness of this memory-cache hierarchy depends on the hit rate of the cache. Therefore, the context models in the memory need to be reallocated based on their coding order and coding frequency.

For example, the previous syntax element of LSCF is always SCF. Hence, the context models correspond to LSCF and SCF can be packed into one word. Four sub_mb_type are coded consecutively. Therefore, the four context models for sub_mb_type in P slice and B slice can be packed into two words, respectively. Furthermore, the interval between encoding two CBFs is very likely smaller than 8 cycles. Therefore, if the hierarchy support more than 7 context model to be prefetched in one cycle, the models of CBF can be packed into one word. The trade-off between hit ratio of cache and cost of cache needs to be considered in this architecture.

SRAM-based and Latch-based Context Memory

Replacing the register-based context memory with SRAM or latch-based memory is another promising way of optimization. According to [34], in single-bin architecture, the BAE with SRAM-based context memory is 33% faster, and 71.2% smaller in area, compared to the BAE with register-based memory. According to [35], SRAM has lower power dissipation than register-based memory, and latch-based memory dissipates even less power than SRAM. However the area cost of latch-based memory is higher than SRAM but still smaller than register-based memory.

Glossary

ASIC Application-Specific Integrated Circuit. ASICs are integrated circuits that are designed for a specific application and are not capable of being reprogrammed.

AVC Advanced Video Coding

BAE Binary Arithmetic Encoder

Bin Binarized symbol encoded in binary arithmetic coding in CABAC.

CABAC Context-Adaptive Binary Arithmetic Coding

CAVLC Context-Adaptive Variable-Length Coding

CBF Coded_Block_Flag. CBF is a syntax element.

CBP Coded_Block_Pattern. CBP is a syntax element.

CIF Common Intermediate Format. CIF is the resolution 352×288.

CM Context Modeling

CMOS Complementary Metal–Oxide–Semiconductor

DCT Discrete Cosine Transforming

FD-SOI Fully Depleted Silicon on Insulator

FHD Full High Definition. FHD is the resolution 1920×1080.

FIFO First-in-First-out buffer

FL Fixed Length

FPGA Field Programmable Gate Arrays. FPGAs are semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects.

GOP Group of Picture

HD High Definition. HD is the resolution 1280×720.

HDL Hardware Description Language

HEVC High Efficiency Video Coding

ICPM Intra_Chroma_Prediction_Mode. ICPM is a syntax element.

ITU-T International Telecommunication Union Telecommunication

LPS Least Probable Symbol

LSCF Last_Significant_Coefficient_Flag. LSCF is a syntax element.

LUT Looking Up Table

MBAFF Macroblock-Adaptive Frame/Field

MPEG Moving Picture Experts Group

MPS Most Probable Symbol

MVD Motion Vector Difference. MVD is the syntax element mvd_lX.

OS Outstanding bits in CABAC are the carry-over bits generated in interval renormalization.

PAFF Picture-Adaptive Frame/Field

PISO Parallel-in-Serial-out buffer

QCIF Quarter Common Intermediate Format. QCIF is the resolution 176×144.

QFHD Quad Full High Definition. QFHD is the resolution 3840×2160. QFHD is also known as 4K UHD which is the dominant 4K standard in television and consumer media.

QP Quantization Parameter

QPD Quantization Parameter Delta. QPD is the syntax element mb_qp_delta.

RAM Random-Access Memory

RDO Rate-Distortion Optimization

RGB Red, Green and Blue. RGB refers to a system for representing the colors samples in computer display.

rLPS Range of sub-interval corresponding to least probable symbol in binary arithmetic coding.

rMPS Range of sub-interval corresponding to most probable symbol in binary arithmetic coding.

ROM Read-Only Memory

RTL Register-Transfer Level

SCF Significant_Coefficient_Flag. SCF is a syntax element.

SE Syntax Element. Syntax Element is the input for CABAC Encoder. It carries coding semantics from previous stages of H.264 Encoder.

SoC System on Chip

SRAM Static Random-Access Memory

TU Truncated Unary

UEGk Unary Exponential Golomb kth-order

UHD Ultra High Definition. UHD resolution standard includes 4K UHD and 8K UHD.

VCEG Video Coding Experts Group

Bibliography

[1] Joint Video Team of ISO/IDC MPEG and ITU-T VCEG. *ITU-T Recommendation H.264: Advanced Video Coding for Generic Audiovisual Services.* May 2003.

[2] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression Video Coding for Next Generation Multimedia.* John Wiley Sons Ltd, 2003.

[3] B. Juurlink, M. Alvarez-Mesa, C. Chi, A. Azevedo, C. Meenderinck, and A. Ramirez. *Scalable Parallel Programming Applied to H.264/AVC Decoding.* Springer, 2012.

[4] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[5] Stephen Le. A fine grained many-core h.264 video encoder. Master's thesis, University of California, Davis, CA, USA, March 2007. http://vcl.ece.ucdavis.edu/pubs/theses/2010-03/.

[6] D. Ammous, F. Kammoun, and N. Masmoudi. A comparative evaluation between cabac and cavlc. *Journal of Testing and Evaluation*, 46(3):1111–1121, May 2018.

[7] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, July 2003.

[8] Detlev Marpe. Context-based adaptive binary arithmetic coding (cabac). http://iphome.hhi.de/marpe/cabac.html.

[9] R. R. Osorio and J. D. Bruguera. High-throughput architecture for h.264/avc cabac compression system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(11):1376–1384, Nov 2006.

[10] J. Chen, L. Wu, P. Liu, and Y. Lin. A high-throughput fully hardwired cabac encoder for qfhd h.264/avc main profile video. *IEEE Transactions on Consumer Electronics*, 56(4):2529–2536, November 2010.

[11] C. Tsai, C. Tang, and L. Chen. A flexible fully hardwired cabac encoder for uhdtv h.264/avc high profile video. *IEEE Transactions on Consumer Electronics*, 58(4):1329–1337, November 2012.

[12] Z. Liu and D. Wang. One-round renormalization based 2-bin/cycle h.264/avc cabac encoder. In *2011 18th IEEE International Conference on Image Processing*, pages 369–372, Sep. 2011.

[13] J. Zhou, D. Zhou, W. Fei, and S. Goto. A high-performance cabac encoder architecture for hevc and h.264/avc. In *2013 IEEE International Conference on Image Processing*, pages 1568–1572, Sep. 2013.

[14] Wei Fei, D. Zhou, and S. Goto. A 1 gbin/s cabac encoder for h.264/avc. In *2011 19th European Signal Processing Conference*, pages 1524–1528, Aug 2011.

[15] H. Shojania and S. Sudharsanan. A high performance cabac encoder. In *The 3rd International IEEE-NEWCAS Conference, 2005.*, pages 315–318, June 2005.

[16] L. Li, Y. Song, T. Ikenaga, and S. Goto. A cabac encoding core with dynamic pipeline for h.264/avc main profile. In *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, pages 760–763, Dec 2006.

[17] C. Kuo and S. Lei. Design of a low power architecture for cabac encoder in h.264. In *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, pages 243–246, Dec 2006.

[18] P. Liu, J. Chen, and Y. Lin. A hardwired context-based adaptive binary arithmetic encoder for h. 264 advanced video coding. In *2007 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, April 2007.

[19] X. Tian, T. M. Le, X. Jiang, and Y. Lian. Full rdo-support power-aware cabac encoder with efficient context access. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(9):1262–1273, Sep. 2009.

[20] Xiaohua Tian, Thinh M. Le, and Yong Lian. *Entropy Coders of the H.264/AVC Standard: Algorithms and VLSI Architectures.* Springer, Berlin, Heidelberg, 2011.

[21] Subramania Sudharsanan and Adam Cohen. A hardware architecture for a context- adaptive binary arithmetic coder. In *Embedded Processors for Multimedia and Communications II*, page 104–112. SPIE, 2005.

[22] A. B. Hmida, S. Dhahri, and A. Zitouni. A hardware architecture binarizer design for the h.264/ avc cabac entropy coding. In *2014 International Conference on Electrical Sciences and Technologies in Maghreb (CISTEM)*, pages 1–4, Nov 2014.

[23] N. Neji, M. Jridi, A. Alfalou, and N. Masmoudi. Fpga implementation of improved binarizer design for context-based adaptive binary arithmetic coder. In *2016 International Image Processing, Applications and Systems (IPAS)*, pages 1–4, Nov 2016.

[24] N. Neji, M. Jridi, A. Alfalou, and N. Masmoudi. Evaluation and implementation of simultaneous binary arithmetic coding and encryption for hd h264/avc codec. In *10th International Multi-Conferences on Systems, Signals Devices 2013 (SSD13)*, pages 1–4, March 2013.

[25] X. H. Tian, T. M. Le, B. L. Ho, and Y. Lian. A cabac encoder design of h.264/avc with rdo support. In *18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*, pages 167–173, May 2007.

[26] L. Wu and Y. Lin. A high throughput cabac encoder for ultra high resolution video. In *2009 IEEE International Symposium on Circuits and Systems*, pages 1048–1051, May 2009.

[27] ITU. H.264/avc reference software, ver. jm 8.6, May 2015. http://iphome.hhi.de/suehring/tml/download/.

[28] Xiph.org. Xiph.org video test media: derf's collection. https://media.xiph.org/video/derf/.

[29] Ultra Video Group Tampere University. 4k video test sequences. http://ultravideo.cs.tut.fi/#testsequences.

[30] Detlev Marpe, Gabi Blättermann, and Thomas Wiegand. Proposed editorial changes and cleanup of cabac. *Joint Video Team of ISO/IEC MPEG & ITU-T VCEG*, pages 22–26, 2002.

[31] J. Chen, Y. Lin, and T. Chang. A low cost context adaptive arithmetic coder for h.264/mpeg-4 avc video coding. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, volume 2, pages II–105–II–108, April 2007.

[32] Z. Xiao, S. Le, and B. Baas. A fine-grained parallel implementation of a h.264/avc encoder on a 167-processor computational platform. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 2067–2071, Nov 2011.

[33] M. Riahi Alam, M. E. Salehi Nasab, and S. M. Fakhraie. Power efficient high-level synthesis by centralized and fine-grained clock gating. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(12):1954–1963, Dec 2015.

[34] Yu-Jen Chen, Chen-Han Tsai, and Liang-Gee Chen. Architecture design of area-efficient sram-based multi-symbol arithmetic encoder in h.264/avc. In *2006 IEEE International Symposium on Circuits and Systems*, pages 4 pp.–2624, May 2006.

[35] Y. Chen and V. Sze. A 2014 mbin/s deeply pipelined cabac decoder for hevc. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 2110–2114, Oct 2014.