VLSI DESIGN AND IMPLEMENTATION OF A HIGH PERFORMANCE H.264 CABAC ENCODER

BY

HASSAN SHOJANIA

A thesis
submitted to the Department of Electrical and Computer Engineering in
conformity with the requirements for the degree of Master of Science (Engineering)

Queen's University
Kingston, Ontario, Canada
February 2006

Abstract

One key technique for improving the coding efficiency of H.264, the state-of-the art video compression standard, is the entropy coding technique known as context-adaptive binary arithmetic coder (CABAC). However, the complexity of the encoding process of CABAC is significantly higher than the traditional table driven entropy encoding schemes such as Huffman coding. CABAC is also bit serial and its multi-bit parallelization is extremely difficult. For a high definition video encoder with a 20 Mbps output stream, multi-giga hertz RISC (reduced instruction set computer) processors will be needed to implement the CABAC encoder.

In this work, we investigate and develop an efficient, pipelined VLSI architecture for CABAC encoding. The resulting architecture efficiently decouples and pipelines the critical stages to address the bottlenecks of renormalization, outstanding bits, and regular/bypass coding modes. The final solution is a single cycle throughput for encoding a binary symbol. An FPGA (field-programmable gate array) implementation of the proposed scheme is capable of 97 Mbps encoding rate. An ASIC (application specific integrated circuit) synthesis and simulation for a 0.18 μm process technology indicates that the design is capable of encoding 190 million binary symbols per second using an area of 0.209 mm^2. The proposed design is thoroughly tested for several standard test contents through both software and hardware simulations with test vectors up to a 300 frames *foreman* content. Also, several designs for CABAC's binarization block and its interface are explored each with different levels of hardware support.

To my parents
Hamideh and Jahangir,
and my brother Amir

Acknowledgements

First and foremost, I would like to thank my thesis advisor, Professor Subramania Sudharsanan, for his invaluable directions and support throughout my research efforts towards this thesis. His insights and suggestions enlightened me in various detailed aspects throughout the work. Besides the technical subjects, I learned many things from him about industry, society, music, life and even Khayyam!

I am grateful for the support of many people who made my master's studies a joy. First, I would like to thank Sepideh Farahvashi for her support and friendship over the last six years. When I started my master's studies, I did not expect to enjoy Kingston much. Surprisingly, my sixteen months stay at Queen's turned out to a memorable portion of my life thanks to the friendship and companionship of many wonderful people, especially Amaan W.H. Mehrabian, Amir B.I. Dehghani, Arezou Mohammadi, Azadeh D.T. Moghtaderi, Babak S.G. Taati, Christine Hsu, Constantin Siriteanu, Farid B.M. Mobasser, Firouz Behnamfar, Guhaprakash Amudhan, Hamidreza S.K. Saligheh-rad, Mehdi H.M. Hedjazi, Mehdi H.S. Moradi, Mehdi H. Shabany, Mohanarajah Sinnathamby, Nooshin D.M. Kiarashi, Paria D.F. Abolmaesumi, Pezhman F.P. Foroughi, Reza S.A. Molaee, Roya D.K. Beheshti, Saeed Y.H. Varziri, Saiedeh D.J. Navabzadeh, Scott Amiss and Stephen Warrington.

Last but not least, my family deserves particular recognition for shaping who I am during all these years. I am grateful to my mother Hamideh, my father Jahangir and my brother Amir for their love, support and encouragements, without which all that I have achieved was not possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The architectural design and implementation of H.264's CABAC (context-adaptive binary arithmetic coding) is motivated by the inherent serial nature of arithmetic coding and the difficulty of designing a high performance architecture specially at the ambitious bitrates targeted by H.264 video standard. This chapter introduces the background of the research area, discusses motivations of this work, presents major features and contributions of the architecture, and outlines the rest of the thesis.

1.1 Background

With the advent of digital audio and video in the last two decades, higher quality multimedia contents have become ubiquitous in everyday life of all of us seen through CD and DVD players, home entertainment units, satellite and cable broadcasts, video telephony, and mobile phones in more recent days. The recording, storage and broadcast of multimedia have evolved significantly with continuous demand for higher quality.

Multimedia has rapidly become the biggest contributor in storage and transmission of data across the world. As a result, compression of multimedia content for

storage and transmission has been advancing significantly with the increased demand for higher quality. Several audio and video compression schemes have been proposed in the literature. The need for interoperability of media and devices produced by different manufacturers has forced them to support development of standard formats for media storage and transmission.

Different standards have emerged in this process, specialized for audio, video or both. The two main standard bodies in this process are Motion Picture Expert Group (MPEG) and International Telecommunication Union (ITU). The standards developed by these bodies have become ubiquitous in every day use. MPEG-1, H.261/3, MPEG-2 and the most well-known of all MP3 (MPEG Layer-3) are examples of these [1, 2]. MPEG-2 standard has rapidly been established as the norm for delivery of entertainment via terrestrial, satellite, and cable transmission standards, and mass storage device standards such as DVD. As a successor to MPEG-2, MPEG-4 was a new ambitious standard introduced in 2000. It was not able to repeat the huge success of MPEG-2 for several reasons. Besides the initial licensing and other issues, the fact that MPEG-4 did not deliver significant video compression performance advantage over MPEG-2 was an important factor.

H.264 [3] is a new video standard jointly developed by ITU and MPEG to address the MPEG-4 failure in advancing compression rate. This standard was finalized in 2003 and added to MPEG-4 as part 10 of the standard. H.264 promises significant improvement in video compression rates by employing many state-of-the-art features. An important feature of H.264 is a new entropy coding technique based on the well-known arithmetic coding concept. This entropy coding mechanism, known as CABAC, is the subject of this thesis.

1.2 Motivation

The H.264 video standard includes several algorithmic improvements for hybrid motion compensated, DCT-based[1] video coding [3]. One key technique for improving the coding efficiency is the entropy coder, context-adaptive binary arithmetic coder (CABAC) [4]. The CABAC utilizes a context-sensitive, backward-adaptation mechanism for calculating the probabilities of the input symbols. The context modeling is applied to a binary sequence of the syntactical elements of the video data such as block types, motion vectors, and quantized coefficients binarized using predefined mechanisms. Each bit is then coded with either adaptive or fixed probability models. Context values are used for appropriate adaptations of the probability models corresponding to a total of 399 contexts representing various different elements of the source data. Each processing step of binarization, context assignment, probability estimation, and binary arithmetic coding is designed with some computational complexity constraint. For instance, the binary arithmetic coder uses a version that has no divisions or multiplications. However, the complexity of the encoding process in its totality is far higher than the table driven entropy encoding schemes such as Huffman coding.

The CABAC encoding process is also bit serial and multi-bit parallelization as in Huffman type encoding is difficult to achieve. Use of a modern microprocessor for encoding a bit consumes hundreds of cycles per bit [5, 6]. For a high definition (HD) video encoder working at an average rate of 20 million symbols per second (e.g., at H.264's Level 4) this can translate into a multi-giga hertz RISC processor requirement. Such large frequencies may not suit low power devices such as cameras where H.264 is to become a dominant standard. Furthermore, instantaneous symbol rates for such encoders can be significantly higher for multiple reasons: picture type (intra or

[1]DCT stands for Discrete Cosine Transform.

inter) variations and pipelined or stream-processing architectures with macro-block level granularity [7, 8]. Such pipelined architectures are preferred in processors that aim to reduce memory and inter-computational block bandwidth requirements [8]. Additionally, if a motion estimator uses rate constrained motion estimation technique, the CABAC encoding symbol rate requirement can go up significantly higher. Under these possibilities, a highly tuned hardware architecture for CABAC encoding is a better alternative than programmable processor-based solutions.

Successful implementation work on CABAC decoding has started earlier than the encoding side [9]. More recent works have attempted to provide efficient schemes for the encoding problem [10, 5]. Nunez, et al. proposed an efficient binary arithmetic coder with a corresponding VLSI architecture as an alternative to the highly complex CABAC process [6]. The solution however is not compatible with the H.264 standard. The scheme proposed in [10] uses a hybrid hardware - software approach with some estimation on the number of cycles per bit and the required silicon area but it addresses the issue of outstanding bits and bit generation incorrectly. The work of Sudharsanan, et al. introduced a novel architecture for a CABAC coprocessor that can be easily integrated on system-on-chip designs [5]. It was shown, with FPGA (field programmable gate array) implementation results, that under certain circumstances, the circuit could achieve the speed of single bit encoding for every two clock cycles but the solution is not complete and its throughput is not enough for high quality contents. One critical step in arithmetic coding is the renormalization of the state registers [11]. The design in [5] addressed renormalization using a simple and bit serial circuit that affected overall performance. The renormalization solution presented by Osorio, et al. [10] is based on a QM-coder implementation [11]. The solution does not elaborate how this is applicable for H.264, particularly with respect to handling "outstanding bits" which is a complex problem.

This work seeks an efficient high throughput CABAC solution targeting real-time

encoding for high quality levels defined in [3], e.g., 20 Mbps for level 4 and 50 Mbps for Levels 4.1 and 4.2.

1.3 Features and Contributions

This work examines hardware implementation of all different parts of CABAC building blocks. The design is thoroughly tested to make it easily portable to real-life SoC (system on chip) designs. The design can be deployed as part of a full or partial (accelerated) hardware-based H.264 encoder design. Efficient solutions for the arithmetic coder and the renormalization process are provided that guarantee a single cycle throughput performance per binary symbol. A number of other issues are addressed too that help reduce the silicon area while maintaining the coprocessor architecture as presented in section 2.4.

The key contributions of the CABAC encoder architecture presented in this thesis are summarized as follows.

1. The *renormalization process* of CABAC's arithmetic coding is analyzed thoroughly and several possibilities for its hardware-friendly implementation are discussed. The major difficulty of its implementation stems from its iterative nature and the fact that the number of iterations can vary at each execution. The most efficient approach decouples the two major parts of the renormalization process (rescale of the coding states and bit generation) from each other. This is achieved by formation of a *parsing area* to be processed based on some *parsing rules*. This approach opens up the possibility of a fully pipelined architecture.

2. The *bypass coding* algorithm is rearranged in such way which allows update of *codILow* coding state and its associated bit generation go through the same pipe

as the one used for *context-based coding*. This streamlines the renormalization stage of arithmetic coding.

3. The issue of generated *outstanding bits* and proper techniques for handling and resolving them are thoroughly addressed. The longest sequence of outstanding bits pattern is investigated and its implications on the right choice of the *intermediate buffer* size, the output FIFO (first-in first-out buffer) width, and the probability of pipeline stall are studied.

4. A complete architecture for arithmetic coding engine and bit generation blocks of CABAC goes through several design stages after exploring multiple venues for increasing the performance. By right arrangement of update and rescale phases of *codIRange* and *codILow* coding states, a fully pipelined architecture emerges which is capable of reaching input *bin* encoding rates of 97 Mbps on FPGA and 190 Mbps on ASIC designs. This rate can easily handle profile Levels 4, 4.1 and 4.2 of H.264.

5. A software/hardwrae co-design approach speeded up both the design exploration and simulation phases. The proposed design is thoroughly tested for several standard test contents through both software and hardware simulations with test vectors up to a full 300 frames *foreman* test content. Other design issues (including system-level ones) are identified and possible solutions explored to make the design easily portable into real SoC designs.

6. For binarization, several design choices are possible based on the upstream block (i.e., the block interfacing with front-end of CABAC) and the system architecture. Different scenarios for binarization with different level of hardware support are explored and related interfaces are designed and discussed.

1.4 Outline of the thesis

In the remainder of this thesis, a thorough examination of our work, VLSI design and implementation of a CABAC encoder, is given. Chapter 2 presents an overview of H.264, entropy coding concepts in the context of video compression, operation of CABAC's building blocks, and their related algorithms. Chapter 3 examines the arithmetic coding algorithm of CABAC more closely and analyzes different issues related to implementation of regular/bypass coding, renormalization process and termination flag. Several architecture designs for arithmetic coding engine and bit generation blocks are presented in chapter 4 which leads to a fully-pipelined architecture. Then, the issue of outstanding bits along with some other design issues is discussed. Chapter 5 shifts the investigation towards the front end of CABAC and its system interface. Several binarizer design choices with different level of hardware support from a thin binarizer layer, intermediate and full hardware binarizers are discussed in detail. Chapter 6 reviews our mixed hardware/software design approach. Taking advantage of functional software models of hardware blocks, the reference software [12] is presented as a fundamental tool in design and verification of this work. The FPGA and ASIC implementation results of the architecture along with the simulation strategy are reviewed in chapter 7. At the end, chapter 8 concludes the thesis and discusses the future work.

Appendix A presents a quick overview of entropy coding concept and advantages of arithmetic coding over traditional Huffman coding. The proofs for some facts about the coding states and arithmetic coding behavior of CABAC are given in appendix B. These proofs will help with assigning proper hardware resources for implementation of arithmetic coding algorithm. Appendix C describes the format of binarization commands defined in section 5.2.

1.5 Notation

To present data values (e.g. register content, instruction format) and binary numbers, a notation similar to the notation of Verilog HDL (hardware description language) is used. For example, $\{1001,\ \{r\{1'b1\}\}, 0110\}$ represents a binary number of width $8 + r$ bits resulting from concatenation of 1001, a string of 1 bits of size r and 0110. As another example, *7'b0111001* shows a 7-bit binary number. In a few occasions, a C language notation is used for representation of hexadecimal values (like *0x1FE*).

Please note that for brevity, often references to "CABAC encoder" are shortened to "CABAC". The CABAC encoding is the focus of the following discussions unless the "decoder" term is explicitly mentioned or the context implies reference to both of the encode and decode processes.

Chapter 2

CABAC: An overview

Video standards in general and H.264 in particular are very complex technical documents. H.264 standard document [3] spans over 280 pages and the reference software [12] developed by the standard team is over 65,000 lines of C code without almost any documentation. This complexity leads to different interpretation of the standard and occasional mistakes in published reviews and explanations of the standard. This chapter provides enough background about the basics, concepts and building blocks of CABAC and prepares the discussion of analysis and hardware implementation of CABAC in the following chapters.

2.1 H.264 introduction

Since introduction of MPEG-4 in 2000, several issues have prevented it from achieving the overwhelming success that MPEG-2 has achieved over the last decade. MPEG-4 is a huge and very ambitious standard with many parts (Systems, Visual, Audio, Content delivery over IP networks, ...) and features like object based coding, support for animated human faces and bodies, hybrid video applications (real-world video, still image and computer generated graphics), a binary language for scene description

called BIFS (BInary Format for Scenes), etc. [2]

MPEG-4 Visual is Part 2 of the standard and is related to coding and representation of visual information. It includes several *profiles* and *levels* like previous standards. Unlike MPEG-2 and its common *MP@ML* (Main Profile at Main Level) combination, MPEG-4 suffered from lack of a focus point and convergence of applications [1]. Further, performance improvement of MPEG-4's video compression over MPEG-2's was not very significant. Its tough licensing terms and disagreements between its patent holders added to the above problems.

In 2001, the Motion Pictures Expert Group (MPEG) started a new initiative to achieve "substantial improvement" in video coding efficiency over MPEG-2. This need came from increasing number of services, growing popularity of high definition TV and lower data rate of existing transmission media, such as Cable Modem and DSL, compared to broadcast channels [13]. Along this path, the MPEG joint forces with the Video Coding Experts Group (VCEG) of the International Telecommunication Union (ITU) and formed the Joint Video Team (JVT). The new standard named Advanced Video Coding (AVC) was published jointly as Part 10 of MPEG-4 and ITU-T Recommendation H.264 in 2003 [2, 1]. As a result, the new standard is known as both H.264 and AVC.

While the main concern of MPEG-4 Visual was flexibility, H.264 emphasizes on efficiency and reliability [2]. H.264's improved prediction and coding efficiency were achieved through several enhancements including [13]: variable block-size motion compensation with small block sizes, quarter-sample-accurate motion compensation, motion vectors over picture boundaries, multiple reference picture motion compensation, decoupling of referencing order from display order, more flexible prediction by allowing B-pictures to be used as reference pictures, weighted prediction, improved "skipped" and "direct" motion inference, directional spatial prediction for intra coding, in-the-loop deblocking filtering, small block-size transform, hierarchical block

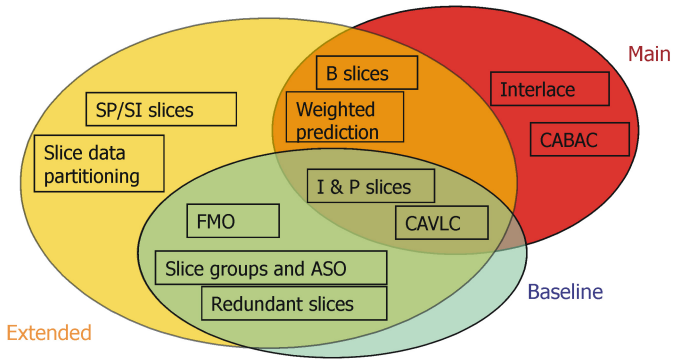


Figure 2.1: Main features of H.264 profiles (adapted from [2, 13]).

transform, short word-length transform by requiring only 16-bit arithmetic, exact-match inverse transform, context-adaptive entropy coding, and arithmetic entropy coding.

Not all of the H.264 features are required for every application. H.264, like previous standards, defines *profiles* and *levels*. Profiles and levels are conformance points for facilitating interoperability between various applications of the standard with similar requirements. A set of coding tools and algorithms that can be used in generating a conforming bitstream are grouped as a profile, whereas a level places constraints on certain key parameters of the bitstream [13] like maximum frame size, maximum macroblock processing rate and maximum video bit rate ranging from 64 kbps[1] to 240 Mbps[1] [3].

Figure 2.1 shows the main features of the three profiles of H.264. Each profile is designed for a particular class of applications: the Baseline Profile for videotelephony, videoconferencing and wireless communications, the Main Profile for television broadcasting and video storage, and the Extended Profile for multimedia streaming

[1]kbps and Mbps stand for "kilo bits per second" and "Mega bits per second" respectively. Each kilo and mega bits are 1024 and 1,048,576 bits respectively.

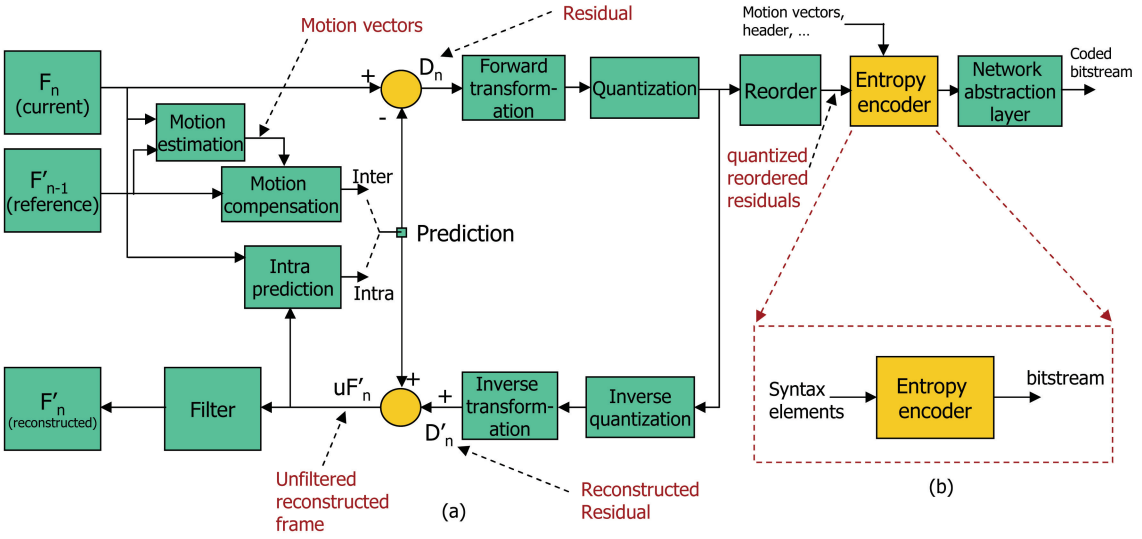


Figure 2.2: Encode path of H.264 (adapted from [2]).

applications [2]. To increase the compression ratio, H.264 designers explored improvement of every section of the encode path. The *Entropy Encoder* is one of the sections significantly improved in H.264.

Figure 2.2 depicts the block diagram of the encoder and highlights the Entropy Encoder in the hierarchy of encode process. Similar to MPEG1 and MPEG2 standards, H.264 is based on hybrid motion compensated, DCT-based video coding. Temporal redundancy of a video sequence can lead to significant compression through *motion compensated predication* techniques. The picture is broken down to tiles of non-overlapping squares each called a *macroblock*. A close match of each macroblock is sought within the neighboring pictures (in time domain) and the motion vector difference between the two macroblocks and the block residuals (with much smaller dynamic range) are encoded. Spatial redundancy within a picture is reduced through *intra-picture* coding technique by using discrete cosine transform (DCT) applied to square blocks of a picture and then quantizing each transform coefficient. Both inter

and intra coding schemes result in a sequence of high-level discrete values, *syntax elements*, which describe the compressed video sequence. This sequence is entropy coded resulting in the encoded bitstream which is wrapped with proper header information in the final stage.[2]

H.264 brought in context-based adaptivity to entropy coding. First, by switching between codebook tables based on the current statistics, it added adaptivity to the common variable length coding (VLC) method employed in the earlier standards. This method is called *context-adaptive variable length coding* (CAVLC) in H.264 literature. Second, a new entropy coding method termed *context-adaptive binary arithmetic coding* (CABAC) was designed to improve the compression further for higher bitrate applications. Because of higher complexity of CABAC, only the Main Profile has included it.[3]

The rest of this chapter introduces entropy coding, compares CAVLC vs. CABAC, and explains different parts of the CABAC encoding algorithm in details.

2.2 Entropy coding

Entropy coder in the context of video encoding converts a series of symbols representing elements of the video sequence into a compressed bitstream suitable for transmission or storage. Input symbols include quantized transform coefficients, motion vectors, markers (indicating synchronization points), headers and other side information [2].

Two widely used entropy coding techniques are Huffman coding and arithmetic coding which are the basis of CAVLC and CABAC employed in H.264 standard. The

[2]This was a very rough description of the basics of hybrid motion compensated, DCT-based video coding. Refer to [1] and [2] for more information.

[3]Anyway, the Main Profile is expected to be more computationally demanding than other profiles because of set of features it supports and high quality of its target applications.

basis of both techniques are reviewed briefly in appendix A. Arithmetic coding has several advantages to Huffman coding. First, it does not need to know the probability distribution of the input symbols and can dynamically adapt to the encoded stream characteristics without requiring continuous update of codebook between encoder and decoder. Further, it does not assign fixed integer-sized codes to each symbol. This leads to more efficient handling of symbols with skewed probability distribution. Overall, arithmetic coding leads to more efficient coding than Huffman coding. On the other hand, its major disadvantage is the complexity of its implementation. Appendix A gives a more detailed description of Huffman and arithmetic coding.

2.3 CABAC vs. CAVLC

The Baseline and Extended profiles of H.264 use CAVLC for their entropy coding stages. This method is based on VLC using several codebook tables and *Exponential Golomb* codes to encode video stream *syntax elements*. Video stream syntax elements are a series of high level symbols comprising the compressed video sequence like transform coefficients residuals, motion vectors, markers, macroblock/slice headers, etc. which are actually the input elements to the entropy coder as shown in Fig. 2.2. A new feature employed in CAVLC is context-adaptivity which is a look-up table based method used for encode of syntax elements related to the total number of nonzero transform coefficients (*TotalCoeffs*) and the number of trailing ones (*TrailingOnes*) in the residual data. Because of "correlation" of *TotalCoeffs* and *TrailingOnes* of the neighboring blocks, the choice between one of the three VLC look-up tables and fixed-length coding is adapted based on the values of same syntax elements at the upper and left blocks. Furthermore, context-adaptivity is used for encode of the level (magnitude) of nonzero transform coefficients where the choice of the VLC table is based on the magnitude of the recently encoded coefficient [14]. Because of correlation

between nonzero transform coefficients within a block, adaptation of the VLC table will lead to higher compression. Though CAVLC performs better than VLC, VLC and subsequently CAVLC are known for the following deficiencies [4] similar to Huffman Coding:

- Coding events with a probability greater than 0.5 can not be efficiently represented. To improve this, an alphabet extension of run symbols is used instead to make up for it.
- Though CAVLC adaptively picks a codebook from the available VLC tables, these general tables are fixed and does not adapt to the actual symbol statistics which varies over space, time, different source material and coding conditions.
- Because of the fixed assignment of a VLC to a syntax element, existing "inter-symbol redundancies" can not be exploited.

The Main Profile of H.264 standard is targeted for high bitrate applications (e.g. TV broadcast) where bitrates of 20-50 Mbps are typical rates for HD content. At such high rates, maximum achievable compression ratio is desirable. Besides other techniques, the Main Profile takes advantage of more advanced entropy coding method of arithmetic coding. While arithmetic coding was an optional feature of H.263 [4], a more effective use of this technique in H.264 creates a very powerful entropy coding method known as CABAC [13].

CABAC achieves between 9% to 14% bitrate savings over CAVLC for a set of test sequences representing broadcast applications at video quality level of 30 to 38 dB as depicted in Figure 2.3. CABAC's main drawback is its higher complexity which requires very efficient execution for high bitrate contents. This is why CABAC is employed only in the Main Profile of H.264 where higher computational power is

[4]H.263 is a videoconferencing standard developed by VCEG of ITU.

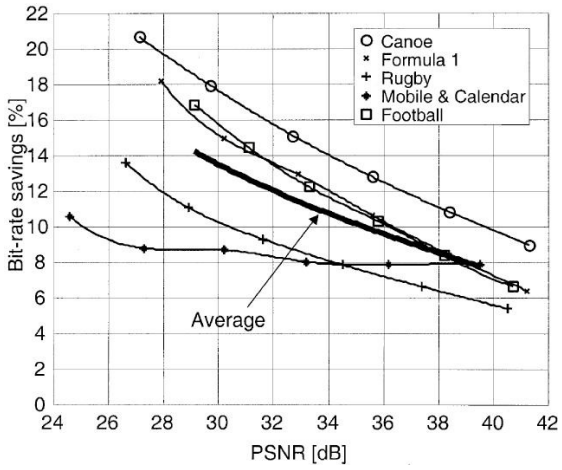


Figure 2.3: CABAC's bit saving relative to CAVLC (from [4]).

expected to be available compared to the Baseline and Extended Profiles. The rest of this chapter introduces CABAC and its components in details.

2.4 CABAC building blocks

CABAC achieves a high degree of adaptation and redundancy reduction by combining an adaptive binary arithmetic coding method with context modeling. To make its efficient software and hardware implementation possible, low-complexity methods for binary arithmetic coding and probability estimation are developed. Here, an overview of the CABAC framework and high-level description of its building blocks are given. Then, each building block is discussed in further details. The main reference of this discussion is [4].

The encoding process of a syntax element goes through the three main steps of *binarization*, *context modeling* and *arithmetic coding* as shown in Figure 2.4. First, a non-binary valued syntax element is uniquely mapped to a sequence of binary symbols, also known as *bin*s or *bin string*. This step is not needed when the syntax element

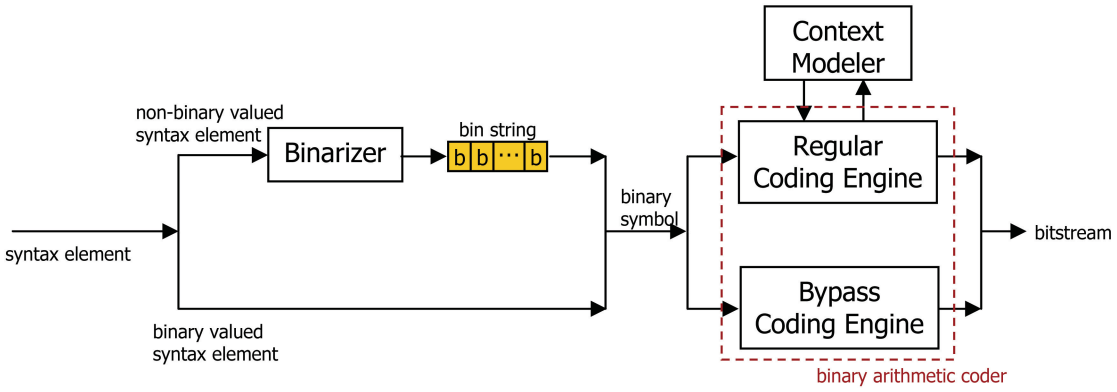


Figure 2.4: High-level diagram of CABAC encoder (from [4] with some modifications).

is already a binary valued symbol, i.e. zero or one. Then, each bin is arithmetically encoded using the *Regular Coding Engine* or *Bypass Coding Engine*. The coding engine used for encode of each bin is decided during the binarization process. The arithmetic encode of each bin can generate zero, one or several coded (output) bits so the relation is not one to one.

The following sections discuss each one of these three main steps in more details.

2.4.1 Binarizer

Binarization is a pre-processing step intended to allow more efficient operation of the subsequent modeling stage through *alphabet reduction*. The maximally reduced binary alphabet is generated by a binarization scheme that assigns a unique intermediate binary codeword, *bin string*, to each non-binary syntax element. This approach helps with both context modeling and easier implementation of CABAC [4].

Instead of the conventional approach of using context models in the original domain of the source with large alphabet size (i.e. in syntax elements space), binarization allows context modeling at sub-symbol level [4]. This is achieved by using independent probability models for each bin of the bin string mapped from the syntax

element. It is reasonable to restrict probability estimation of individual symbol statistics to the area of the largest statistical variations. Such bins are modeled individually while bins with less statistical variation will use a joint model.

Also, binarization allows use of "binary" arithmetic coding instead of q-ary coder operating on the original q-ary source alphabet. As discussed in section A.3, such higher level arithmetic coders demand much higher computational power while achieving limited coding gains. The computational overhead of encoding several bins instead of one pass in an q-ary coder can be compensated by a fast binary coding engine and the fact that the probability of symbols resulting in long bin strings is typically very low [4].

The CABAC's binarizer uses different binarization schemes. The main schemes rely on a few basic code trees which can be calculated on the fly without table lookup. They include *unary code*, *truncated unary code*, the *k-th order Exp-Golomb code*, the *fixed-length code*, and three concatenations of fixed-length and Exp-Golomb codes. The other category uses five fixed unstructured binary trees that are derived manually for macroblock and submacroblock types.

The main issue with the design of binarizer is the design of right system interface (binarizer serves as the front-end block of CABAC) with proper balance of hardware support. More information about the mechanics of binarizer will be given in chapter 5 where its implementation is discussed.

2.4.2 Context modeler

An important property of arithmetic coding is the possibility of separation of modeling and coding stages through a clean interface [4]. At encode of each binary symbol, the coding engine utilizes the model probability assigned to that bin. The model tracks the statistical dependencies of encoded symbols. The model cost increases by estimation of higher order conditional probabilities and at the same time, the model

performance directly affects the coding efficiency[4]. As a result, the right balance between implementation cost and performance of the model is very important.

The model cost is proportional to the alphabet size, N, and number of contexts, C, as shown in [4] which is briefly discussed here. Let's assume the modeling function $F : \mathbf{T} \longrightarrow \mathbf{C}$ operates on set $\mathbf{T}$ of past symbols, the *context template*, and models them by the context set $\mathbf{C} = \{0, ..., C-1\}$. To encode the incoming symbol x according to the probability model of already coded neighboring symbols $z \in \mathbf{T}$, the conditional probability $p(x|F(z))$ is estimated on the fly by tracking the actual source statistics. After encode of symbol x, the probability model is updated with the value of encoded symbol x. For an alphabet size of N, there will be $\tau = C(N-1)$ probability estimates to track. τ is an indication of the learning cost of the model. An unreasonable large number of contexts C could result in overfitting the model and inaccurate estimates of $p(x|F(z))$. CABAC imposes two restrictions on the choice of the context models to avoid overfitting. First, only a limited number of context templates T (e.g. a few neighbors of the current symbol) are employed so that a small number of context models C is effectively used. Second, context modeling is restricted to selected bins of the binarized symbols. This ad-hoc design of context models may not result in the optimal coding efficiency but reduces the model cost significantly [4].

CABAC utilizes four types of conditional probabilities for retrieval of context models. The first type uses two neighboring syntax elements in the past of the current syntax element as context template. Usually, the neighboring elements are the ones to the left, A, and on the top, B, of the current syntax element, `x`, as shown in Fig. 2.5-(a). *mb_skip_flag* is an example of this type. The second type is defined for *mb_type* and *sub_mb_type* syntax elements where the values of prior coded bins $(b_0, b_1, ..., b_{i-1})$ within the bin string are used as context template (like nodes of an equivalent binary tree) for the current bin b_i. A sample tree for binarization of *sub_mb_type* is shown in Fig. 2.5-(b) where conditions *C0*, *C1* and *C2* at nodes of the binary tree map

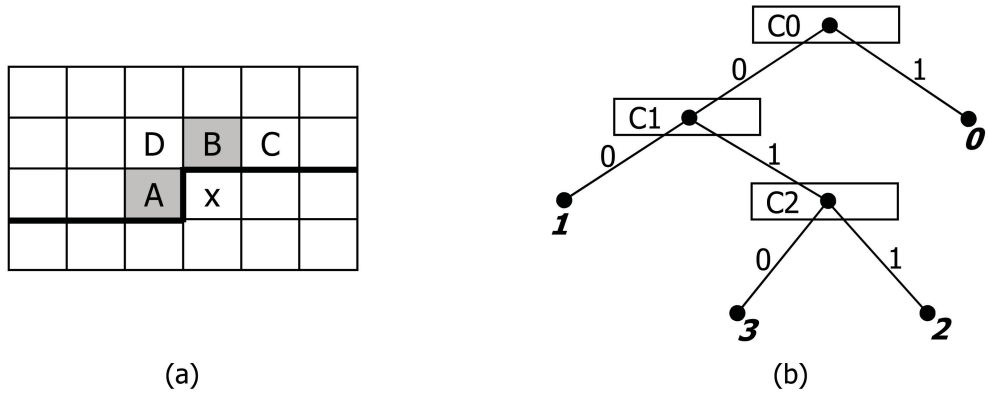


Figure 2.5: Context templates for two groups of syntax elements: (a) neighboring syntax elements (b) sub_mb_type for P/SP slices (adapted from [4] and [3]).

sub_mb_type values of 0, 1, 2 and 3 to bin strings 1, 00, 011 and 010 respectively.

The third type is used to encode the *significance map* (includes *significant_coeff_flag* and *last_significant_coeff_flag* syntax elements) of the residual data. Here the position in the scanning path determines the context. The forth type is to encode the residual level information using the accumulated number of previously encoded levels as the condition to decide the context. All other regular-coded bins use "fixed" assignment of context models to bin indices without evaluating any template condition.

The entire context model can be arranged in a linear fashion such that each model can be identified by a unique *context index* γ as shown in Table 2.1. Each context includes a pair of two values[3], a 6-bit *probability state index* σ_γ and the binary value ϖ_γ of the *most probable symbol* (MPS). Thus, the whole context model is a table consisting of 399 pairs of $(\sigma_\gamma, \varpi_\gamma)$ for $0 \leq \gamma \leq 398$.

The contexts in range from 0 to 72 are related to syntax elements of macroblock type, submacroblock type, spatial/temporal prediction modes, and slice/macroblock control information. The ones from 73 to 398 are related to the coding of residual data. Since in *macroblock adaptive field/frame* (MBAFF) mode, mixed field and frame coding of macroblocks within a *slice*[5] is possible, separate sets of models are

Table 2.1: Syntax elements and their associated range of context indices (adapted from [4]).

Syntax elements	Slice type		
	SI/I	P/SP	B
mb_type	0/3-10	14-20	27-35
mb_skip_flag	n/a	11-13	24-26
sub_mb_type	n/a	21-23	36-39
mvd (horizontal)	n/a	40-46	
mvd (vertical)	n/a	47-53	
ref_idx	n/a	54-59	
mb_qp_delta	60-63		
intra_chroma_pred_mode	64-67		
prev_intra4x4_pred_mode_flag	68		
rem_intra4x4_pred_mode	69		
mb_field_decoding_flag	70-72		
coded_block_pattern	73-84		
coded_block_flag	85-104		
significant_coeff_flag	105-165, 277-337		
last_significant_coeff_flag	166-226, 338-398		
coeff_abs_level_minus_1	227-275		
end_of_slice_flag	276		

defined for significance map of residual data for field and frame coded macroblocks. In non-MBAFF (pure frame or field) coded pictures, only 277 of the total 399 probability models are actually used.

It is important to remember that the lifetime of probability models is limited to one slice only. They are initialized at the beginning of each slice, updated after encode of each bin of the slice and reset again at the next slice. In other words, the statistical dependencies "within each slice" is used for update of the context models so the models of different slices are decoupled. Also, only past syntax elements belonging to the same slice are evaluated as condition of context template for the current slice. Depending on the slice type (e.g., B or P slice), the initialization values of some

[5]Each picture is subdivided into one or more *slice*s that each is the basic spatial segment encoded independently from its neighbors. The slice concept is very useful in providing error resilience as errors or missing data from one slice cannot propagate to any other slice within the picture.

contexts are different.[6]

Because the context index retrieval and the binarization are closely related, more details on calculation of context index for different syntax elements will be given in chapter 5. For now, it is enough to know that for each bin encoded in the regular coding mode, a context index is "somehow" calculated to fetch the pair of $(\sigma_\gamma, \varpi_\gamma)$ from the context table.

2.4.3 Binary arithmetic coder

As discussed in section A.3, arithmetic coding is based on recursive interval subdivision and generates codewords for sequence of symbols rather than generating a separate codeword for each symbol in a sequence. The interval is represented by its lower bound *low* and its width *range*. If $p_{LPS} \in (0, 0.5]$ is the probability estimate of the *least probable symbol* (LPS) (i.e. probability of the next encoded symbol having the opposite polarity of the most probable symbol, MPS), then division of the interval will be according to the following equations since $p_{MPS} = 1 - p_{LPS}$ for a "binary" symbol:

$$range_{LPS} = range \cdot p_{LPS} \tag{2.1}$$

$$range_{MPS} = range - range_{LPS} \tag{2.2}$$

Depending on the polarity of next encoded symbol matching either MPS or LPS, the corresponding subinterval is chosen as the new interval and this process continues for the rest of the sequence. For unambiguous identification of the interval, the Shannon lower bound on the entropy of the sequence is asymptotically approximated by using the minimum precision of bits specifying the lower bound of the final interval

[6]That is why the first three rows of Table 2.1 use different context indices for different slice type though only one range is effectively used for a slice lifetime.

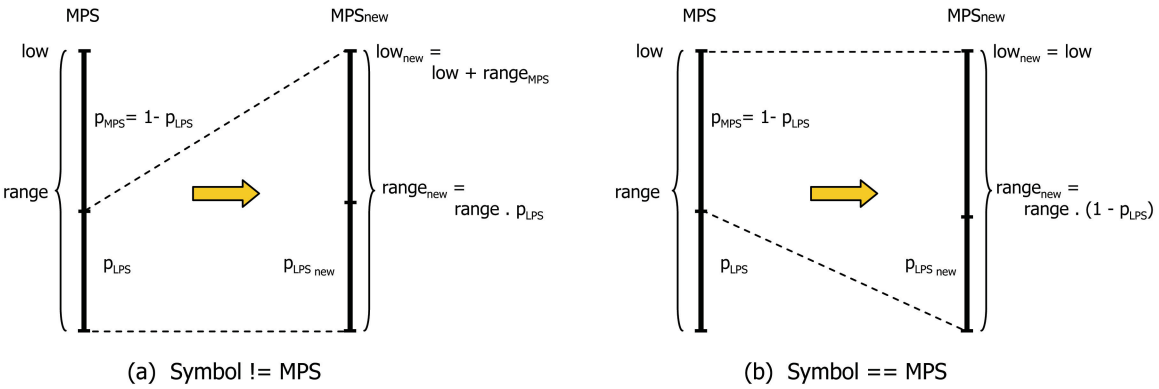


Figure 2.6: Binary arithmetic coding interval update in CABAC.

[4].

This process is shown for encode of a symbol in both cases of the symbol being equal to LPS and MPS in Fig. 2.6. The coder state consists of *range* and *low* values for identifying the interval, and p_{LPS} and MPS for identifying the symbol statistics. Note that the symbol probability estimate is updated after encode of each symbol. p_{LPS} is increased after encode of a symbol equal to LPS as Fig. 2.6-(a) and decreased after encode of a symbol matching MPS polarity as Fig. 2.6-(b). Instead of tracking the probability of a particular symbol polarity (i.e. p_0 or p_1), H.264 standard keeps p_{LPS} which is always the smaller probability value, within (0, 0.5]. Then the MPS binary value records the polarity of the most probable symbol.[7]

In an efficient implementation of binary arithmetic coding, it is desirable to avoid the multiplication operation of Equation 2.1 as it can become a bottleneck. Significant work in this area has resulted in several multiplication-free implementations through approximation of the range and probability estimate p_{LPS} like *Q coder* [15] and its derivatives *QM* and *MQ coders* [16] especially in works related to the still image

[7]This does not introduce bit savings because the extra bit dedicated to MPS could have been used for the probability estimate to store any value within (0,1] instead. But this form of definition reduces the size of *TransIdx* table required for update of the probability state since the probability range is (0, 0.5] instead of (0, 1]. This table will be introduced in section 2.5.2.

standardization groups JPEG (Joint Photographic Experts Group) and JBIG (Joint Bi-level Image experts Group). But these techniques performed poorly for H.264 video coding and made the standard designers develop an alternative multiplication-free arithmetic coding scheme called *modulo* or *M coder* [17]. This scheme uniformly quantizes the legal range of interval $[R_{min}, R_{max})$ (which is equal to $[2^{b-2}, 2^{b-1})$ for a b-bit precision used in arithmetic coding operations [18]) and the probability range associated with LPS $(0, 0.5]$ onto small set of representative values $Q = \{Q_0, ..., Q_{K-1}\}$ and $P = \{p_0, ..., p_{L-1}\}$ respectively. Then the multiplication of Equ. 2.1 can be pre-computed and stored in a table of $K \times L$ product values $Q_k \cdot p_l$ for $\{0 \leq k \leq K-1\}$ and $\{0 \leq l \leq L-1\}$ with table entries having width of $b-2$ bits [17].

For the regular coding engine of H.264, a good tradeoff between the table size and interval subdivision approximation was found by $K = 4$ quantized range intervals and $L = 64$ LPS probability values [4]. For bins with nearly uniform distribution, context modeling is skipped and the simplified bypass coding mode is used instead.

Now that the basics of the CABAC building blocks are covered, the detailed mechanics of the coding process is discussed.

2.5 The coding process

A high-level flow of processes at arithmetic encode of a slice is shown in Fig. 2.7. Since CABAC state is reset at the beginning of each slice, the encode process is shown for a slice unit from initialization of the encoder till request of slice termination.

For encode of each slice, first the CABAC state is initialized. Then the bins are read one by one from the output buffer of the binarizer (*ReadBin* process). Besides the bin value, the bin type (i.e. encode mode of the bin in bypass or regular) and the context index associated with the bin (if regular coded) accompany the bin too. Based on the bin type, one of the two main coding modes (regular or bypass) or the special

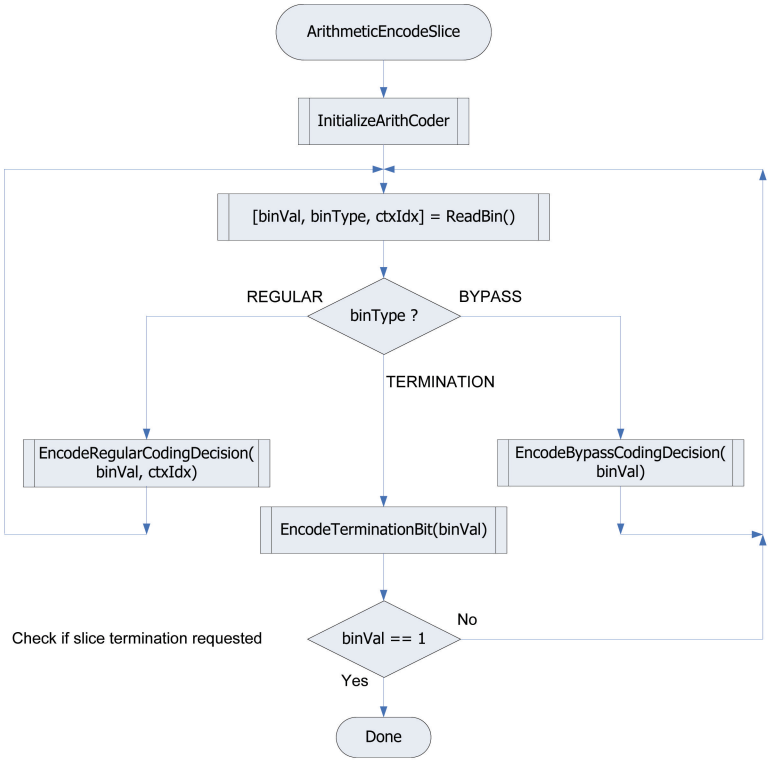


Figure 2.7: High-level view of arithmetic coding of a slice.

termination mode is selected. For the bypass mode, the bin is bypass coded using *EncodeBypassCodingDecision* process. For the regular mode, the bin is coded using its associated context index, *ctxIdx*, based on *EncodeRegularCodingDecision* process. At the end of each macroblock, a termination bit is encoded (*EncodeTerminationBit* process). If the bin is equal to one, the macroblock is the last macroblock of the current slice and the slice is terminated by flushing the CABAC arithmetic coding state into the bitstream buffer. Otherwise, the next bins are encoded with repeating the described sequence.[8]

The following sections explain different components of the coding process.

[8]Note that the "decision" term used in the name of bypass and regular coding processes above refers to the "binary decision" made for arithmetic encode of each bin, and is inherited from the standard document [3].

2.5.1 Initialization

Several variables and components of CABAC need initialization at different stages of encoding. Excluding the binarizer which will be discussed later, arithmetic coder needs to be reinitialized before encoding the first macroblock of each slice.[9] This is because the lifetime of arithmetic coder state and backward adaptation of the probability model is limited to a slice boundary. CABAC provides proper initialization values for both groups as discussed below.

Initialization of arithmetic coder states

The variables associated with arithmetic coding process, their precision and their initialization values include the followings [3]:

codILow is a 10-bit value which points to the start of the arithmetic coder interval. It is initialized with 0 to point to the very start of the available range.

codIRange is a 9-bit value specifying the interval size. It is initialized to $0x1FE$ (510 decimal) which is almost the maximum size possible for the interval.

firstBitFlag is a single-bit flag that keeps track of whether any output bit for the current slice is generated or not. It is initialized to 1 and its purpose is to drop the first generated bit as it is always zero (see *PutBit* flowchart in section 2.5.3 and section B.3).

SymbolPointer [10] is a counter that points to the next binary symbol to be encoded. Basically, incrementing this value suggests that encode of the current symbol is finished and the next bin can be encoded. It needs to be allocated

[9] To be more accurate, it also needs to be reinitialized after encoding the *pcm_alignment_zero_bit* and all *pcm_byte* data for a macroblock of type I_PCM [3].

[10] It is called *SymCnt* in the standard document [3].

$\lceil \log_2 (MaxBinCountInSlice + 1) \rceil$ bits where *MaxBinCountInSlice* is the maximum number of bins existing in one slice [3].

bitsOutstanding is a counter that keeps track of number of *outstanding bits* pending to be resolved by a future *non-outstanding bit* during the renormalization process which will be described in section 2.5.3. The standard document [3] suggests allocation of the same number of bits as *SymbolPointer* because it could potentially grow as large as the whole slice!

Context model initialization

Appropriate initialization values for probability models provide *a priori* knowledge about the source statistics [4]. These initial states are adaptive in two ways.

First, initially given slice quantization parameter *SliceQP* [11] is used to provide pre-adaptation of the initial probability states to different coding conditions that *SliceQP* can represent them. This initialization scheme is called *QP-dependent initialization*. The standard document [3] defines a pair of initialization parameters (u_γ, v_γ) for each context model with index γ where $0 \leq \gamma \leq 398, \gamma \neq 276$.[12] For each context model, the (u_γ, v_γ) pair and *SliceQP* are used to derive the initial value of probability state p_{LPS} and MPS through the below procedure:

$\sigma_{pre} = clip(1, 126, ((u_\gamma * SliceQP) >> 4) + v_\gamma)$

if $\sigma_{pre} \leq 63$ **then**

$\texttt{C}ontextEntry_\gamma.p_{LPS} = 63 - \sigma_{pre}$

$\texttt{C}ontextEntry_\gamma.MPS = 0$

else

$\texttt{C}ontextEntry_\gamma.p_{LPS} = \sigma_{pre} - 64$

[11] Possible change of quantization parameter "within" the slice will not affect the initialization.

[12] Model associated with index 276 is a special non-adaptive model that will be discussed later. It has a fixed initialization and never changes state.

$\texttt{C}ontextEntry_{\gamma}.MPS = 1$

end if

The *clip* operation above makes sure the third parameter remains within [1, 126] range. Note that based on this clipping, no p_{LPS} is assigned a value of 63 as 63 remains a special non-adaptive probability state used only for context index 276. The statements inside *if/else* condition above maps the [1, 126] range of σ_{pre} to ranges of [62, 0] with $MPS = 0$ and [0, 62] with $MPS = 1$ respectively.

The second scheme for adaptivity comes from defining three different sets of (u_{γ}, v_{γ}) parameter pairs (to be used in the above procedure) for each context model used in P and B slices. This effectively increases the number of initialization tables to three for P and B slices giving encoder more flexibility to adapt to the video content for higher coding gains. This forward adaptation mechanism is called *slice-dependent initialization* and requires signaling of the initialization table index used for encode of each slice to the decoder (ranging from 0 to 2 to specify one of the three initialization tables) in the slice header [4, 3]. By employing this forward-adaptation method for initialization of the model, bitrate savings of up to 3% have been reported [4].

2.5.2 Regular or context-based coding

As mentioned before, regular coding engine of CABAC employs context modeling for efficient encode of each binary symbol. For encode of each bin in the regular mode, the bin along with its associated context model is passed to this engine. The context model is identified by a context index pointing to an entry inside the context table. Fig. 2.8 is an updated version of Fig. 2.4 which emphasizes the retrieval of context model from the context table. Each generated bin by the binarizer is associated with a *bin index* which is the index of that bin in the bin string corresponding to the syntax element. Context index is calculated by *Context Index Calculator* block on Fig. 2.8

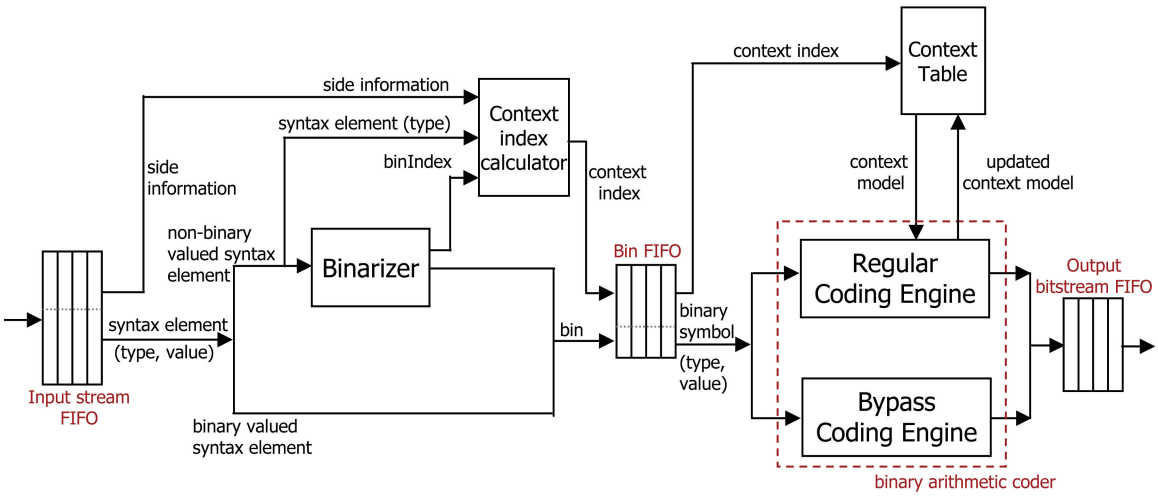


Figure 2.8: Updated version of Fig. 2.4 emphasizing context model retrieval and intermediate FIFO's at block interfaces.

using the bin index, type of the syntax element and other side information specific to the syntax element type. As this calculation is closely related to the binarization process, it will be discussed in chapter 5.

First, the context model at index γ is looked up from the context table. The context model along with the bin are used by the regular coding engine and the updated context model is written back to the context table at the original γ location. Then, the generated bits (if any) will be added to the output bitstream.

Regular coding engine accesses several tables for its encode process. Figure 2.9 shows organization and interface of these tables. Their addressing and data access are done through the left and right ports of each memory respectively.

Their description is as follows.

Context Table This table stores the whole context model and consists of 399 context entries defined by the standard documentation [3]. Each entry stores the

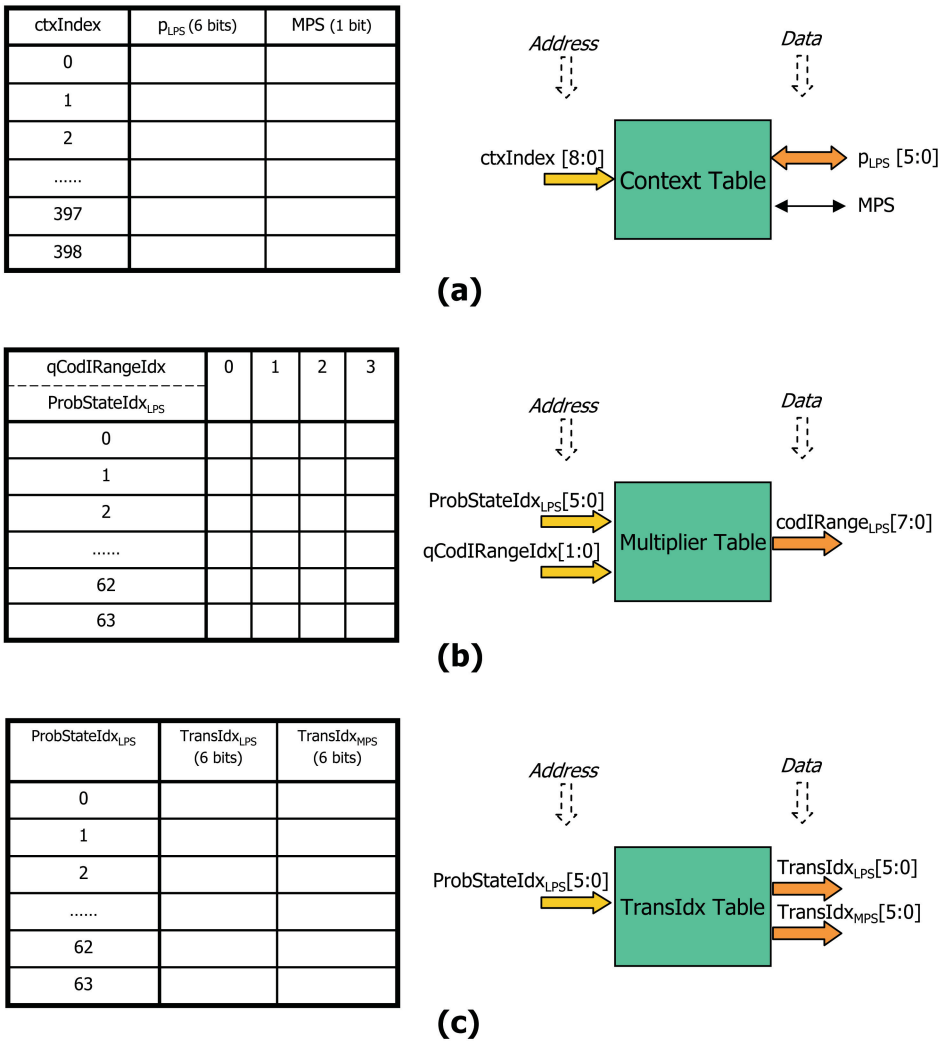


Figure 2.9: Memory tables used in CABAC's arithmetic encode process: (a) *Context Table* content and its interface (b) *Multiplier Table* content and its interface (c) *Next Probability State Table* content and its interface.

context model for a particular type of bin and has two fields: a 6-bit quantized probability state σ_γ and a binary value ϖ_γ for the MPS. Each context entry is identified by a *context index* which can be represented by a 9-bit value ($\lceil log_2 399 \rceil = 9$). At encode of each regular-coded bin, the engine reads a context entry and writes back the updated context model in the same location of

the table. The total size of the table will be $399 * (6 + 1) = 2,793$ bits. The context table is initialized based on the process described in section 2.5.1.

Multiplier Table As discussed in section 2.4.3, H.264 designers picked $K = 4$ quantized range intervals and $L = 64$ LPS probability values for their multiplication-free implementation of binary arithmetic coder using table lookup. With a picked $b = 10$ bits precision in CABAC, each table entry is $b - 2 = 8$ bits wide [18]. The table size is $K \cdot L = 4 * 64 = 256$. A 2-bit range indicator and another 6-bit indicator for probability state form an 8-bit address to fetch the multiplication result of interval and probability state as shown in Fig. 2.9-(b).

Next Probability State Table The valid range (0, 0.5] for probability of the least probable symbol p_{LPS} is quantized to 64 cells so a 6-bit indicator can represent the probability state (as stored in *Context Table*). The quantization is not uniform and is realized by a Markov chain [17] based on the following formulas.

$$\begin{aligned} & N = 64, \ p_0 = 0.5 \\ & p_{N-1} = p_{min} = 0.01875 \\ & \alpha = \left(\frac{p_{N-1}}{p_0} \right)^{1/(N-1)} \\ & p_i = \alpha \cdot p_{i-1} \quad i = 1, ..., N-1 \end{aligned} \tag{2.3}$$

Fig. 2.10 is the MATLAB simulation of Equ. 2.3. As it shows, the idea is that there is less need for an accurate estimation of p_{LPS} near 0.5 (since two closely neighboring states result in nearly the same code length) compared to p_{LPS} near 0 (i.e., p_{MPS} near 1) [17]. After encode of each bin, the model's probability state is updated. Encode of a MPS bin decreases p_{LPS} (with increase of state index) and similarly after encode of a LPS symbol p_{LPS} increases (with decrease of state index). Two tables of 64 entries each provide the next state for both

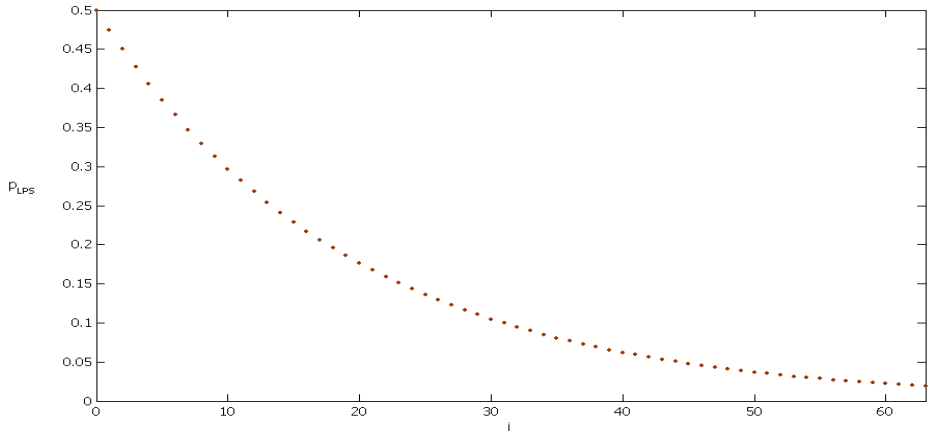


Figure 2.10: Non-uniform quantization of p_{LPS} within (0, 0.5].

cases. Figure 2.11 shows the transition graph for the next probability state for both cases.[13]

The flowchart of H.264 binary arithmetic encoding process in the regular (also known as *context-based*) mode is shown in Figure 2.12. For easier comparison against the standard document, mainly the same terminology of the H.264 standard document is followed here. *codIRange* and *codILow* specify the binary arithmetic "coding states", i.e., the interval width *range* and the tag value *low* respectively. Since a precision of $b = 10$ bits is used for the interval states, *codILow* is assigned a 10-bit integer. The legal range for *codIRange* is within $[2^{b-2}, 2^{b-1})$ [18]. Because *codIRange* could temporarily fall below 2^{b-2}, it is assigned a 9-bit integer to cover the whole range of $[0, 2^{b-1})$ though renormalization process will bring it back within the legal range (see section 2.5.3).

As discussed in section 2.4.2, each bin *binVal* encoded in the regular mode is assigned a context model. The context model is identified by its context index γ,

[13] State 63 remains at 63 at encode of either LPS or MPS symbol. It is a non-adaptive probability state used only by context entry $\gamma = 276$ for encode of termination bit (see section 3.6). The upper saturation point for other p_{LPS} states is 62 meaning that no other state can go to state 63.

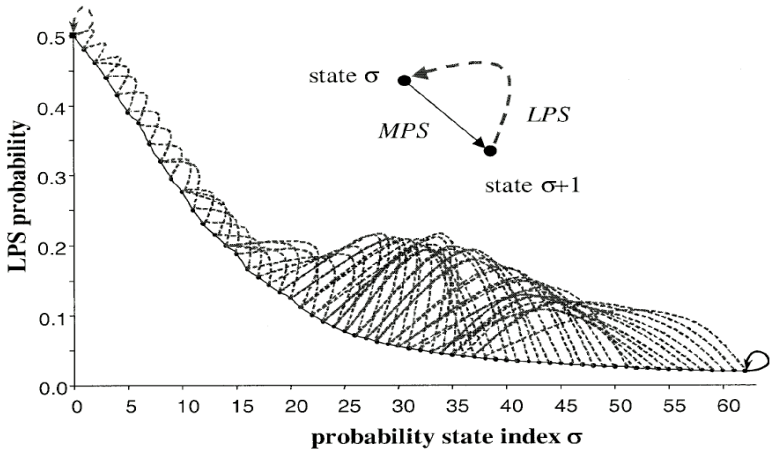


Figure 2.11: LPS probability values and transition rules for updating the probability estimation of each state after observing a LPS (dashed lines in left direction) and a MPS (solid lines in right direction) [4].

which is referred to as *ctxIdx* in this flowchart, and points to a memory location in the context table. The bin and context index are the explicit inputs. The interval states and the context model are the implied inputs. Each step of the flowchart is described below.

1. The context model associated with the bin is looked up from the context table. p_{LPS} and MPS of the bin context model are stored in $ProbStateIdx_{LPS}$ and $valMPS$ respectively.

2. *codIRange* is within its legal range of $[2^{b-2}, 2^{b-1}) = [256, 512)$ at start of encoding and its top bit (bit 8 in zero-based indexing) is known to be always one. As a result, the two bits necessary to uniformly quantize its legal range (since $K = 4$) will be its next top two bits, i.e., bits 7-th and 6-th. After retrieving these two bits, the Multiplier table is looked up to find size of the sub-interval associated with the least probability symbol, $codIRange_{LPS}$ per Equation 2.1 (see Fig. 2.6). Then, size of the dual area $codIRange_{MPS}$ is calculated per Equation 2.2.

3. The bin value *binVal* is checked against the MPS polarity of the statistical model available for that bin and one of the two paths is followed.

4. The interval state variables are updated based on match or mismatch of the encoded bin and MPS of the model.

5. The probability state p_{LPS} is updated by looking up the next state from either lookup tables of $TransIdx_{LPS}$ or $TransIdx_{MPS}$ depending on the *binVal* polarity. If the encoded symbol is a LPS and its next probability value is going above 0.5 (its current probability index is 0), then the polarity of MPS need to be reversed (instead of decreasing further the probability state into a negative number). Otherwise the polarity of MPS will not change.

6. The new probability state and MPS polarity are written back to the *Context Table* at location *ctxIdx*.

7. The renormalization process discussed in the next section potentially "rescales" the interval state variables and generates some output bits.

8. Next binary symbol is picked from the input to be encoded either in the regular or bypass mode.

The explicit output of regular-mode encode of a bin is either no or several output bits which will be appended to the output bitstream. Other side-effects include update of the context table and interval state variables. Update of interval states happens in two phases: first, at interval subdivision; second, at potential rescale by the renormalization process. Next section discusses the critical step of renormalization and its merit within binary arithmetic coding.

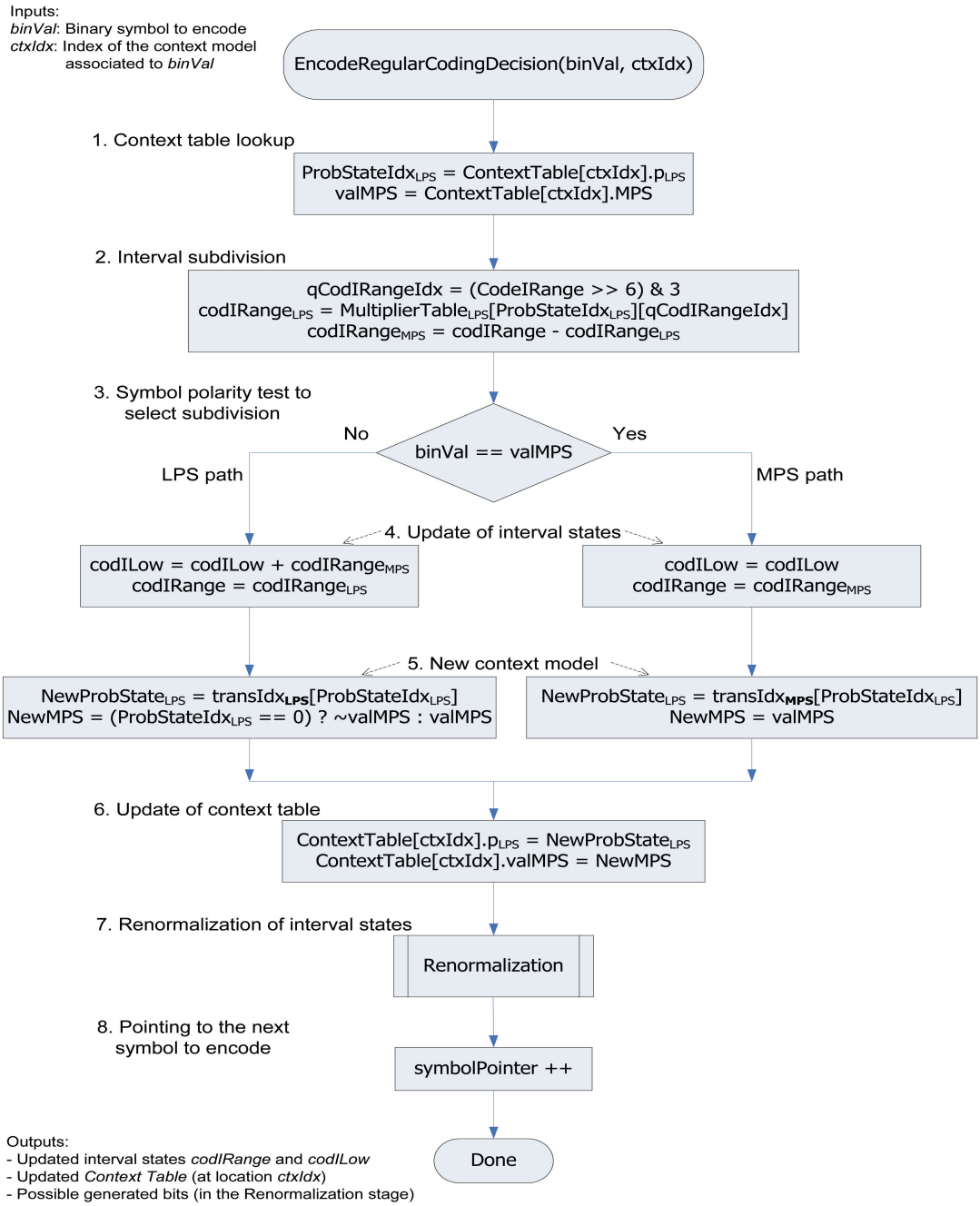


Figure 2.12: Flowchart for regular encode of a binary symbol (adapted from [3]).

2.5.3 Renormalization in regular-mode coding

Renormalization is an important part of binary arithmetic coding. After encode of each bin and interval subdivision, the interval size decreases. This will result in imprecise division of the interval after encode of few bins. This is known as the *underflow* problem [19]. To guarantee correct operation of the encoder, the interval size *codIRange* should be kept as large as possible.

Similar to the approach of [18], CABAC's implementation of renormalization maintains *codIRange* within $256 \leq codIRange < 512$ before encoding each new bin. This is done through rescale of the interval states. Because of limited precision of the state variables, rescaling can not continue without loss of data. As a result, the renormalization process generates the output bits gradually. Actually this "incremental code generation" is favorable as it allows "incremental transmission".

Flowchart of the renormalization process is shown in Fig. 2.13. It is an iterative process that updates *codIRange* and *codILow* till *codIRange* reaches a minimum of 256. If *codILow* has a value less than 256, then a 0 bit is generated. If *codILow* is greater than or equal to 512 (i.e. the interval is fully within the upper half of the possible range space of the 10 bits *codILow*), the right branch generates a 1 bit and lowers the interval by 512. Otherwise, the middle branch (when $256 \leq codILow < 512$), will "eventually" cause generation of a bit but with unknown polarity at this time. These "to be generated bits" are called *outstanding bits* and their occurrence is tracked by *bitsOutstanding* counter. In case the middle or the right branch is taken, the interval is shifted down by 256 and 512 respectively as the corresponding bit in *codILow* is already tracked by the proper generated bit. After the three branches, the interval states are both doubled to increase the interval accuracy. This process continues till the interval size reaches a minimum value of 256. Each iteration will generate a bit: of outstanding type if the middle branch is taken; otherwise, of non-outstanding type with a value of 0 or 1.

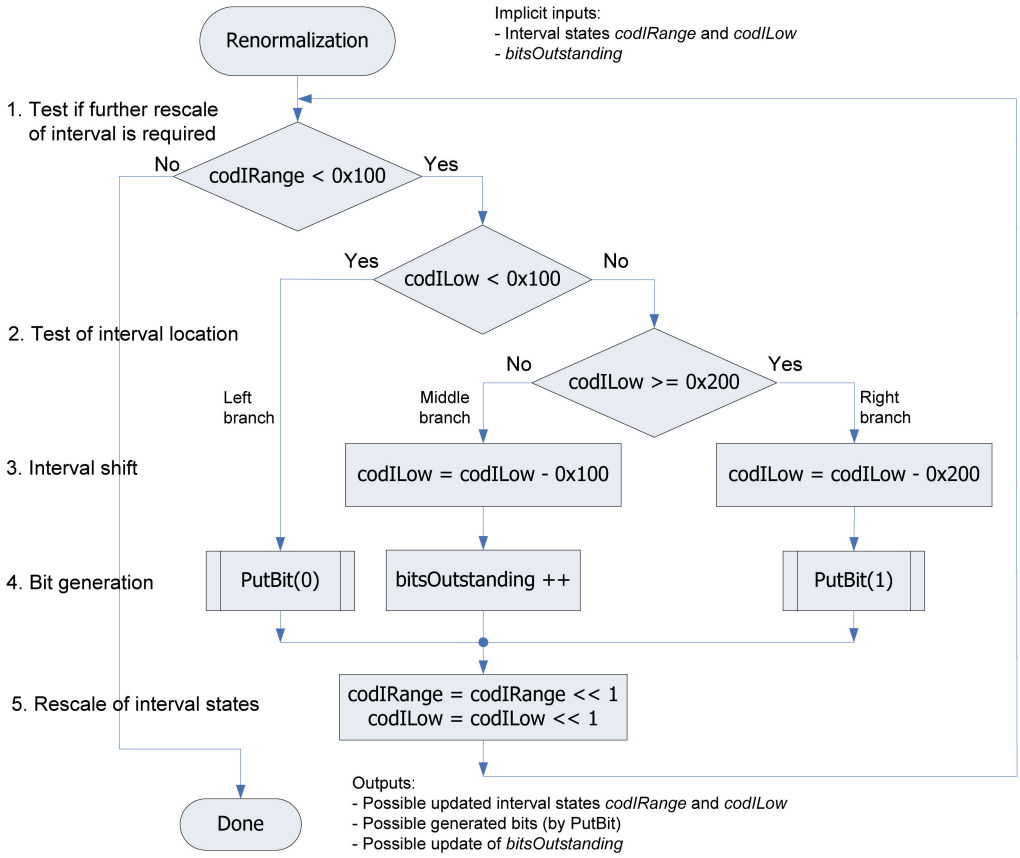


Figure 2.13: Flowchart of renormalization for regular encode of a binary symbol [3].

The polarity of outstanding bits will be known when the next *non-outstanding bit* is generated by either the left or right branch. Such bit could happen in the renormalization process associated with encode of the current bin or encode of a future bin. This is why *PutBit* process is called at left and right branches instead of a simple write to the output buffer so the polarity of the outstanding bits can be resolved.

The flowchart of *PutBit* is shown in Fig. 2.14. This procedure normally[14] outputs a bin with the requested polarity to the output stream followed by a series of bits

[14]Except the very first bit generated in encode of every slice.

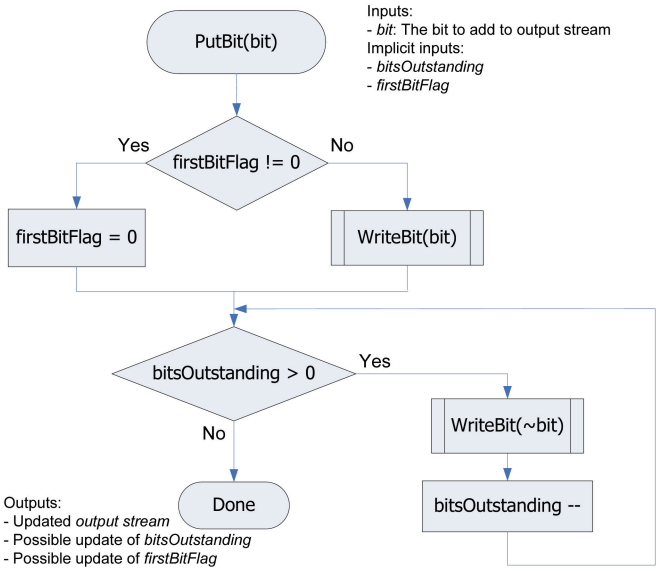


Figure 2.14: Flowchart of bit generation for encode of a binary symbol [3].

with opposite polarity with a size equal to *bitsOutstanding*. This is the point that the previously unknown polarity of the outstanding bits is resolved by the arrival of the first non-outstanding bit.[15]

Based on the standard specifications, the very first bit within each slice is ignored (to be discussed in section 4.3.2). Variable *firstBitFlag* tracks the arrival of the first bit within a slice and drops it. After that, it will not affect *PutBit* behavior anymore. At the end of this procedure, always *firstBitFlag* and *bitsOutstanding* will be 0. *WriteBit* simply sends a bit to the output stream by appending it to the stream buffer.[16]

More discussion about renormalization process will be given in chapter 4 where the difficulties of implementing this process is discussed.

[15]Note that a bit with the polarity of the non-outstanding bit (which is passed as a parameter here) is sent to the output as if the bit is "inserted" before the first outstanding bit.

[16]Note that this definition of *WriteBit* is simpler than the definition in [3] as no second parameter for repeat of the bit is used.

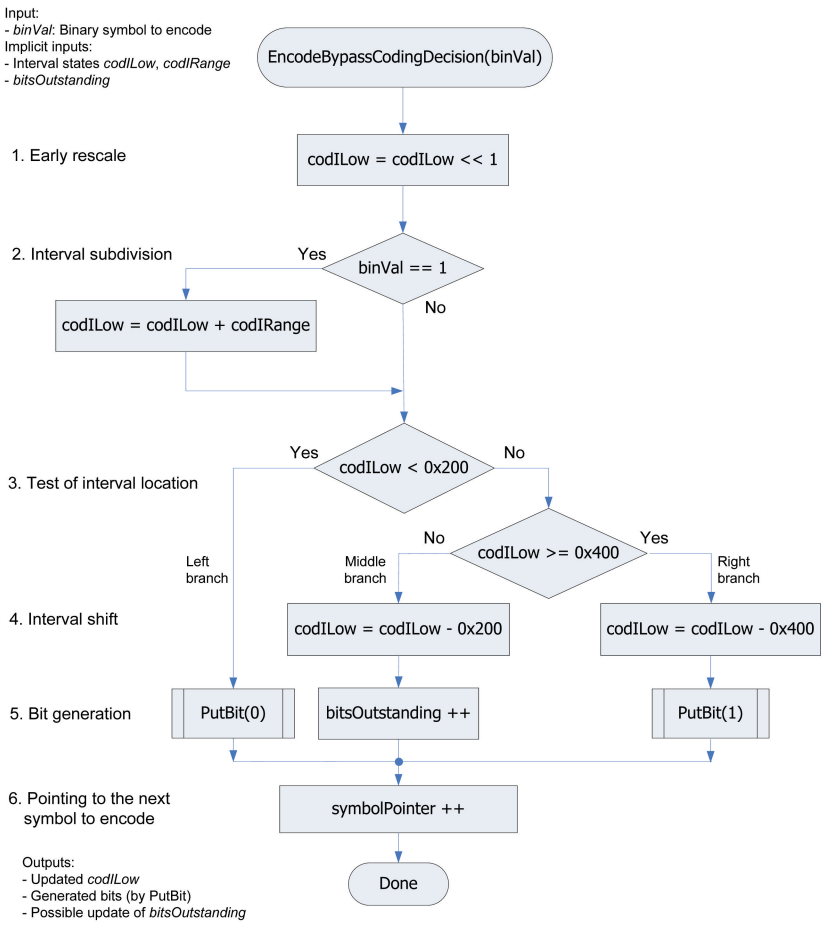


Figure 2.15: Flowchart of bypass encode of a binary symbol (adapted from [3]).

2.5.4 Bypass coding

As discussed in section 2.4.3, some bins show almost equiprobable behavior so context modeling is not necessary for them. For such bins, a much simpler encoding process compared to the regular mode is implemented where probability estimation is totally skipped and the arithmetic coding and the renormalization stages are integrated. Interval subdivision in this mode should create two equal subintervals half of the

original interval. But rescaling the interval states would double the subinterval again ending up with the original interval size. As a result, doubling of interval states for renormalization is ignored, *codILow* is doubled before interval decision (i.e. selection of upper or lower subinterval based on *binVal*), and renormalization is done with "doubled decision thresholds" [4] instead in condition check and update of *codILow*.

Figure 2.15 shows the bypass process. As mentioned above, *codIRange* will not change. Considering the fact that *codIRange* was already more than 256 before start of coding this bin, rescale of interval state variables would need to be done only once. That is why the effect of subdivision and rescale for *codIRange* are canceled and there is no iteration loop in the bypass mode unlike the renormalization in the regular coding mode. Because of the early rescale of *codIlow* at the first step, renormalization conditions and shift of interval happens with doubled threshold values. Further discussion of why this approach works and some suggested modifications for easier hardware implementation will be given in section 3.5.

As shown in the flowchart, bypass encode of each bin always generates a single output bit (though it could be an outstanding one) so no compression happens for bypass coded bins. One might suggest that the whole binary arithmetic coding could have been skipped for bypass-coded bins altogether and the "input bin" *binVal* been added directly to the output stream. But the problem is that multiplex of a raw bit within an arithmetic codeword is not an easy task and requires a form of "arithmetic codeword termination". Since bypass-coded symbols are not grouped together in general, the extra overhead (i.e. output bits) associated with codeword termination before each bypass-coded bin will create lots of waste. This rules out use of methods like "lazy coding mode" [16] (also known as "arithmetic coding bypass" [20]) established in JPEG 2000 area in the context of H.264 [4].

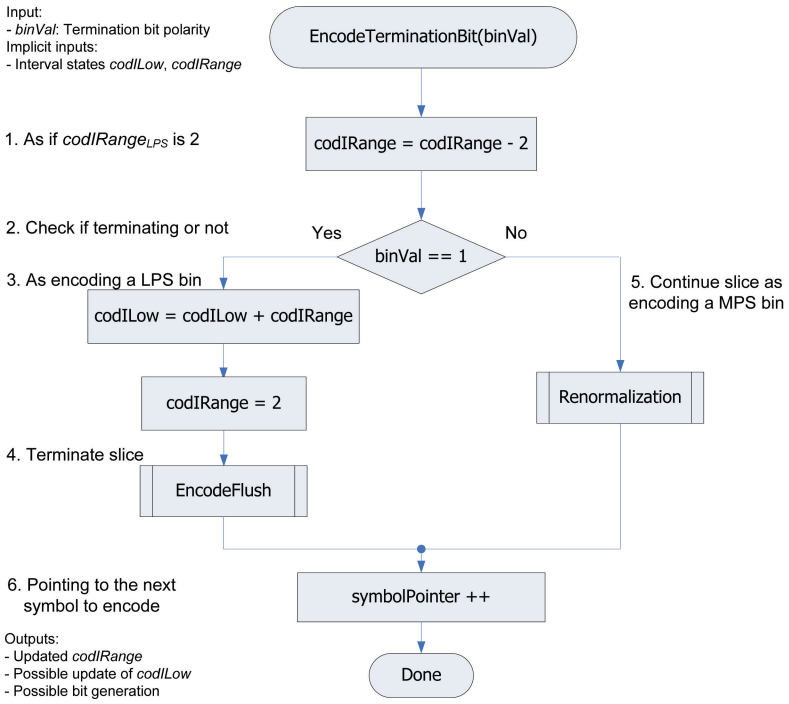


Figure 2.16: Flowchart of encode of the termination flag, *end_of_slice_flag* [3].

2.5.5 Termination flag coding

Termination of a slice by encoder requires calculation and transmission of enough number of bits so that the final interval states can be identified unambiguously by decoder [18]. The bits following the termination bit within the input stream can belong to another slice and might be even encoded using a different entropy coder (e.g. VLC for slice header information of the next slice). That means the arithmetic encoder will no longer be used for the current slice and its state needs to be flushed properly into the output steam.

The termination process is shown in Figure 2.16. After each macroblock, a bit for slice termination flag *end_of_slice_flag* is encoded to show whether the last encoded

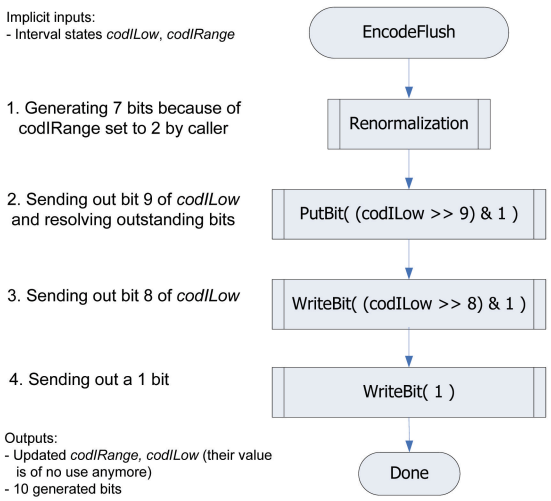


Figure 2.17: Flowchart of flushing at termination (adapted from [3]).

macroblock was the last macroblock within the current slice or not. If the flag is off (i.e. slice still continues), a fake $codIRange_{LPS}$ equal to 2 is used and interval subdivision is done as if a MPS bin is encoded (i.e. *codIRange* is set to $codIRange_{MPS}$ and *codILow* is maintained) and then regular renormalization is done through the right branch.

If the flag is on (i.e. terminating the slice), interval subdivision is done as if a LPS bin is encoded (i.e. *codIRange* is set to $codIRange_{MPS} = 2$ with update of *codILow*). The call to *EncodeFlush* will generate ten bits as shown in Figure 2.17. First, seven bits are generated during renormalization since *codIRange* is 2. In the second step, the top bit of *codILow* (which is already updated during renormalization) is sent out resolving potential outstanding bits. Then the next top bit of *codILow*, the eight bit, is sent out.[17] And finally, a 1 bit is directly sent out. What not shown here is the padding zero bits added after the terminating bits (if necessary) to make the output

[17]Note that there are no remaining outstanding bits after the last step.

stream exact multiple of bytes [12].

2.6 Arithmetic coding in JPEG2000 and H.264

JPEG2000 uses a variant of MQ arithmetic coder for its entropy coding. MQ coder itself is based on the QM coder used by JBlG (Joint Bi-level Image experts Group) standard, but uses the byte emission technique of the Q coder. MQ coder is multiplier free, all arithmetic is implemented using adders, and it is context-based like its predecessors [21].

Similar to CABAC, each symbol has an associated context determining its probability estimate through a probability state and the value of MPS. Pairs of binary decision D and context CX provided by the modeling unit are processed together to produce compressed image data. The coding operations are based on fixed precision integer arithmetic representing fractional values in which 0x8000 is equivalent to 0.75 decimal. The interval size A and interval position C are doubled in a renormalization process to maintain A within [0.75, 1.5) whenever it falls below 0x8000 [20].

Though the main concepts of context-based arithmetic coding is shared between JPEG2000 and CABAC, there are several differences in implementation details because of the different applications of the two standards. Some of the differences are in precision of interval states, definition of interval range, number of context models, number of probability states to represent probability estimates, termination, binarization, etc. But one of the most important differences is in the compressed bit generation. In CABAC, compressed data is generated bit by bit as explained in *Renormalization* and *PutBit* flowcharts. But in JPEG2000, "a byte" of compressed image data is removed from the high order bits of the C register (similar to *codILow* in CABAC) after every 8 iterations of renormalization. The removed byte can not be sent to the output stream yet until the carry-over from C register is resolved.

JPEG2000 employs a *bit-stuffing* method to insert a zero bit after a 0xFF byte to prevent carry propagation to already generated byte. Though this carry-over issue has some similarities to the concept of outstanding bits (where polarity of some bits will be known by generation of a future bit), its mechanics is completely different.

Chapter 3

Analysis for implementation

Chapter 2 provided an introduction to the CABAC encode algorithm. This chapter builds on that introduction focusing more on the issues regarding efficient hardware implementation of CABAC. The details of the difficulties arising in such implementation are discussed.

Beside the binarization, arithmetic encode of the bins and packing of the generated bits are the main other major processes involved in CABAC. Here, the path along the complete process of a bin from its regular-mode arithmetic encode, renormalization and packing of the generated bits is followed first. A simple design for implementing regular (i.e., context-based) arithmetic encode and update of the coding states is presented. Then, the renormalization loop and its variable number of iterations for rescale of the coding states are discussed. Two solutions for fixed latency implementation of the renormalization loop are presented and efficiency of the ROM-lookup approach vs. the *parsing area* approach is discussed. Packing and buffering of the generated bits and the output bitstream interface are presented in section 3.4. Next, two schemes for resolve of the buffered outstanding bits are compared.

Implementation options for the bypass coding mode are studied too. The bypass mode is integrated to the main encode path to share the same renormalization and

bit generation logic with the regular mode. And finally, encode of the termination bit is handled as a special case of regular coding.

By the end of this chapter, all major issues in design of arithmetic encode, renormalization and generation of encoded bits are discussed. The next chapter will look into putting these pieces together and designing a high performance architecture. But first, a brief review of the related works in this area is given.

3.1 Previous works

Hardware implementation of CABAC is a new interest and there has not been much work in this area. The closest such work is the work of Osorio, et al. that provides some key novel ideas that enable single cycle throughput [10], but does not conform to the H.264 standard. The renormalization step has serious problems where the outstanding bits issue is equated with the bit-stuffing process of JPEG 2000 which is far simpler. Since no verification result is provided, the correctness of the encoding process against a standard decoder cannot be established. The paper completely relegates the binarizer to a higher level RISC processor that has to be closely integrated with the CABAC encoder.

Although the work of Chang, et al. [22] is for the JPEG2000 standard, it gives some insights for possible implementation of a CABAC encoder. It suggests using a table lookup for renormalization of *range* value when LPS path is taken but this leads to different latencies depending on encoded bin taking MPS or LPS path. The work of Nunez, et al. replaces the arithmetic encode algorithm of H.264 with a different one based on *MZ-coder* and achieves simpler hardware implementation over CABAC [6]. But it suffers from incompatibility with H.264 as it can not generate the same bitstream. Again, they fail to address the binarizer implementation. Sudharsanan, et al. proposes a thin binarizer layer with context index calculation through table

lookup [5]. But its implementation of the renormalization path takes variable number of cycles without considering issues related to bit generation, e.g., outstanding bits issue.

None of the above works in the area of CABAC encoder could achieve an efficient solution capable of handling bitrates of HD-quality contents which makes the case for a fresh new initiative in this area.

3.2 Regular coding implementation

As discussed in section 2.5.2 and described in the flowchart of Fig. 2.12, a hardware implementation of regular coding is not complicated. First the context table is accessed to read the context modeling information associated with the bin. Then the multiplier table is accessed to find the interval size of LPS area, $codIRange_{LPS}$. Using $codIRange_{LPS}$, the MPS sub-interval, $codIRange_{MPS}$, is calculated through a simple subtraction. The new interval start for the LPS case also needs to be calculated through an addition. Then based on the polarity of the encoded bin *binVal*, the new interval states are selected from MPS or LPS paths. But the updated interval states have to go through the renormalization stage too before the values are written back to *codIRange* and *codILow* registers. While updating the state intervals, the context model is updated through looking up the next probability state from $TransIdx_{LPS}$ and $TransIdx_{MPS}$ tables. The MPS value of the updated context model can get reversed in case encode of an LPS bin attempts to increase the probability state to more than 0.5 (with decreasing the probability index below than the current 0 value).

Fig. 3.1 shows a possible layout of logic blocks and their interface for implementation of the regular engine. The context table can be implemented with a 399 entry RAM (Random Access Memory) as context models require update after encode of each bin. The multiplier table $Range_{LPS}$ and the next probability states

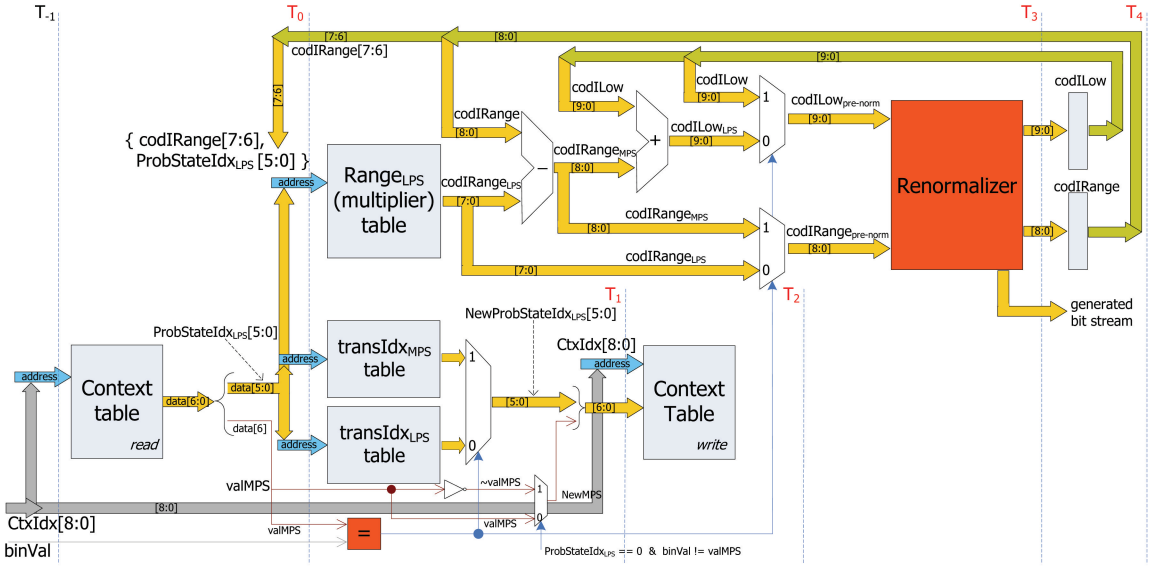


Figure 3.1: Implementation of regular encode of a binary symbol.

$TransIdx_{LPS}, TransIdx_{MPS}$ can be implemented through ROM (Read Only Memory). The accuracy of arithmetic operations does not need to be more than 10 bits as interval representation accuracy was picked as $b = 10$ by CABAC [18, 4]. Since *codIRange* is a 9-bit value and $codIRange_{LPS}$ value retrieved from the multiplier table is a $b - 2 = 8$ bits value, the subtraction operation would not need more than 9 bits of accuracy. Based on the proof in section B.1.1, it is guaranteed that the subtraction and addition operations will not generate an underflow or overflow.

The clock boundaries specified on the figure comes from the fact that the available FPGA implementation platform for this work was an *Altera Stratix 1S80* FPGA which its TriMatrix memory structure only consist of synchronous memory blocks. As a result, clock boundaries T_{-1}, T_0 and T_1 are enforced because of access to the Context Table (read at T_{-1} and write at T_1) and Multiplier Table at T_0. It was assumed here that a combinational architecture for implementation of renormalization can be found so when combined with the earlier arithmetic and multiplexing operations

after the Multiplier Table read, it would not take more than almost two cycles. More accurately, from Multiplier Table read to availability of renormalized coding states should not take more than three cycles and the new coding states can be registered at the next clock, T_4.

As the figure shows, the bottleneck for regular encode of every bit spans from the first read access to the coding states at T_0 to T_4 when the coding state registers are ready to be accessed for encode of the next bin. The read of probability state from Context Table is not part of the bottleneck as the read required for encode of the next bin can happen in parallel to T_2 or T_3 cycles of the current bin. Based on the high level architecture presented in Fig. 2.8, a series of bins and their associated context indices are ready in the FIFO between the Binarizer and Arithmetic Encoder. Therefore, the early read access to Context Table is not an issue here.

Note that the above discussion of number of cycles in the suggested architecture (especially for the parts with combinational logic) was based on rough initial design estimates. This architecture evolves significantly in the next sections and the following chapter.

3.3 Renormalization implementation

The renormalization process rescales the arithmetic coding states. A direct implementation of what presented in [3] takes a variable number of iterations to scale *codIRange* state to a minimum value of 256 with successive left shifts (Fig. 2.13). The number of iterations, *iter*, varies from zero to seven depending on the incoming

codIRange calculated in the arithmetic coding stage.[1] Section B.2 proves why number of iterations can vary between zero and seven. Each iteration updates *codILow* by potentially resetting one of its top two bits and then shifting it to the left. A "single" output bit is generated at "each" iteration and added to the output stream. The polarity of the generated bit at each iteration depends on the taken branch.

Figure 3.2 shows the flow of iterations for renormalization of a single bin as a state diagram by naming the branches of Fig. 2.13 as *1*, *1+* and *0* from right to left. While the polarity of the generated bit for *1* (one) and *0* (zero) branches are already determined (same as the branch label), the polarity for *1+* branch is unknown till a future bit (zero or one) is generated. This future bit could be generated either in the current renormalization process or in a renormalization corresponding to encode of a future symbol that could happen several symbols away. As suggested in [3], a counter, *count*, can keep track of the number of these *1+* bits (also known as "outstanding bits") until a future "non-outstanding bit" resolves them to a known value. This dependency on the future bits introduces a serious challenge to hardware implementations as the length of outstanding bits can grow. For example, the standard document [3] does not set an upper limit on *count* and suggests it could grow as large as the slice size!

When a non-outstanding bit, *resolve bit*, is generated after a sequence of outstanding bits, all the pending outstanding bits are resolved to either a one followed by *count* number of zeros or a zero followed by *count* number of ones depending on whether the resolve bit is a one or zero respectively. For example, $\{1{+}1{+}1{+}1{+}\}$ will resolve to $\{10000\}$ with arrival of a *1* bit and to $\{01111\}$ with arrival of a *0* bit.

If not addressed properly, the variable number of iterations could force frequent

[1] Some other works have derived wrong values for the maximum number of iterations possible in the renormalization stage because of either absence or not thorough simulation of their proposed implementations. Osorio et al. derives 8 as the maximum number of iterations [10] while [6] assumes 6 for the same number.

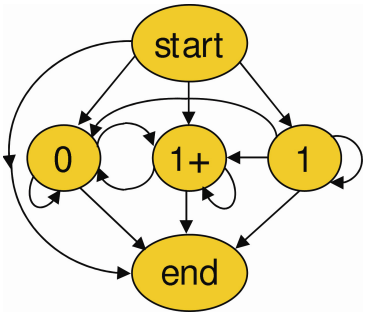


Figure 3.2: Possible flow of renormalization branches at encode of a regular bin.

stalls in the arithmetic encoder since renormalization has to be completed "before" encoding the next incoming bin. This reduces the overall throughput of the coder.

The following sections discuss further details of the renormalization process and suggest two methods for its efficient implementation. The ROM-lookup based approach is simpler but requires 172,032 bits of memory. It will be shown that by processing a *parsing area* and using a much smaller ROM with 6656 bits, renormalization can be efficiently decoupled into the two phases of coding states rescale and bit generation.

3.3.1 Iterations in renormalization loop

Better understanding of sequence of iterations in renormalization is essential before being able to tackle the difficulty of an efficient design. Note that in the following discussions, the type of taken branch (*1, 1+* or *0*) and the generated bit at each iteration are equivalent as the branches are named based on the output bit they generate. As Fig. 3.2 suggests, not all combinations of branches can happen for the renormalization stage at encode of a given bin. Here are a few observations:

- *1* bits can only be generated at the initial iterations of renormalization of a bin. A sequence of *1* branches can happen only in an uninterrupted fashion.

When a *1+* or *0* bit is generated, no more *1* bit will be generated in the current renormalization process. This is because the *1* branch is taken only when the top bit of $codILow$ is set. When a *1+* or *0* branch is taken, no more *1* bit can reach the top position, bit 9, of $codILow$. As soon as a *1* reaches the second highest position (bit 8) of $codILow$, it will be reset by the *1+* branch so it will not reach the top position with further left-shifts of $codILow$ at next iterations.

- The total number of generated bits (*0*, *1+* or *1*) is equal to the number of left-shifts required to end up with a 1 bit at the top position (bit 8) of $codIRange$. This number can be expressed by $iter = 8 - \lfloor \log_2 codIRange \rfloor$.

The following examples clarify the process of coding states update and output bits generation.[2]

Case 1- CodIRange=122; CodILow=502: The number of left-shifts to make $CodIRange \geq 256$ is two. Therefore, the number of renormalization loop iterations will be two, i.e. $iter = 2$. As a result, $CodIRange$ will end up as $122 << 2 = 488$. Both iterations will take the *1+* branch since *CodILow* is *10'b0111110110*. The generated bits are {*1+1+*} which can not be appended to the output bitstream as they need to be resolved by a future non-outstanding bit first. Instead, number of outstanding bits, *count*, is incremented by 2. The updated $CodILow$ will be *10'b0111011000=472*.

Case 2- CodIRange=3; CodILow=932: The number of required iterations will be 7. As a result, $CodIRange$ will end up as 384. The taken branches pattern, i.e., the generated bits, are {*1111+001+*} because *CodILow=10'b1110100100*. The first *1+* bit is followed by a non-outstanding *0* bit so they will be resolved as {*1+0*} $\Rightarrow$ {*01*} and the generated bit string becomes {*1110101+*}. The resolved

[2]The shown coding state values are the results of coding interval update.

portion of this string, {*111010*}, can now be added to the output stream but the trailing *1+* needs to wait for a future non-outstanding bit. This wait is indicated by setting the pending outstanding bits count to one i.e. $count = 1$ at the end of this renormalization process. The renormalized $CodILow$ will be *10'b0000000000=0*.

If there is no pending outstanding bits (from previous encodes), the resolved string can directly be appended to the output stream. Otherwise, the pending outstanding bits need to be resolved first. As indicated in {***1****111010*}, the leading one is the first non-outstanding bit generated so it behaves as the resolve bit for any potential pending outstanding bits. Let us assume here the number of pending outstanding bits was 3 ($count = 3$) before starting current renormalization so {*1+1+1+*} are the pending bits. These two strings added together will result in {*1+1+1+* ***1****111010*} which will be resolved to {*100011010*} and appended to the output bitstream.

Case 3- CodIRange=72; CodILow=610: *iter* will be 2 for renormalizing the interval range to make it $72 << 2 = 288$. The two iterations will take *1* and *0* branches respectively because *CodILow* is equal to *10'b1001100010*. As a result, the generated bits will be {*10*} which can be appended directly to the output bitstream as they do not contain outstanding bits. The renormalized $CodILow$ will be *10'b0110001000=392*.

Case 4- CodIRange=54; CodILow=362: *iter* will be 3 for renormalizing the interval size to make it $54 << 3 = 432$. The two iterations will take the *1* and *0* branches respectively as *CodILow=10'b0101101010*. As a result, the generated bits will be {*1+01+*} which its trailing *1+* bit need to wait for future resolution by setting $count = 1$. Note that first, the current content of *count* is read to determine pending outstanding bits and potentially resolve them. Then it will

be updated with a new value.

The leading {*1+0*} is resolved to {*01*} which can be appended to the output bitstream directly if there is no pending bit before this renormalization, i.e. $count = 0$. Otherwise, bit *0* of {***01***} behaves as the resolve bit for pending outstanding bits. Then, the resolved pending bits and {*1*} will be appended to the output bitstream right after each other. The renormalized $CodILow$ will be *10'b0101010000=336*.

Now several methods for implementation of renormalization are proposed and evaluated.

3.3.2 Renormalization through ROM lookup

A closer look at Figure 2.13 shows that the number of scaling iterations, *iter*, is equal to the number of leading zeros of $codIRange$ because number of leading zeros will be effectively the number of required left shifts to end up with a 1 bit at the top position of the range value, i.e., $codIRange \geq 256$. Hence, the new $codIRange$ can be simply calculated by left shifting by the lead zero count. In other words, *iter* will be equal to the lead-zero count.

Obtaining the new $codILow$ is, however, more complex. A straightforward solution could implement renormalization of $codILow$ through a ROM lookup. Basically, the inputs to the renormalization process consist of: $codILow$, $codIRange$ which can effectively be replaced by *iter*, and the number of outstanding bits, *count*. The outputs of this process include updated $codILow$, updated *count*, and the new generated output bits (also known as *output string*) to be appended to the output bitstream.

The 10-bit $codILow$ could take 1024 different values and number of iterations required to scale $codIRange$ to a minimum value of 256 could be seven in the worst

address	countIncrement 3 bits: 0-7	bResolvePending 1 bit: boolean	ResolveBitPolarity 1 bit	NewOutBits 6 bits	OutBitsCount 3 bits: 0-6	NewCodILow 10 bits
0	.....	.....	.....	.....	.....	.....
.....	.....	.....	.....	.....	.....	.....
0x5F6 (case 1)	2	false	X	{XXXXXX}	0	472
.....	.....	.....	.....	.....	.....	.....
0x662 (case 3)	0	true	0	{1}	1	392
.....	.....	.....	.....	.....	.....	.....
0x998 (case 4)	1	true	1	{0}	1	336
.....	.....	.....	.....	.....	.....	.....
0x1BA4 (case 2)	1	true	1	{11010}	5	0
.....	.....	.....	.....	.....	.....	.....
0x1FFF	.....	.....	.....	.....	.....	.....

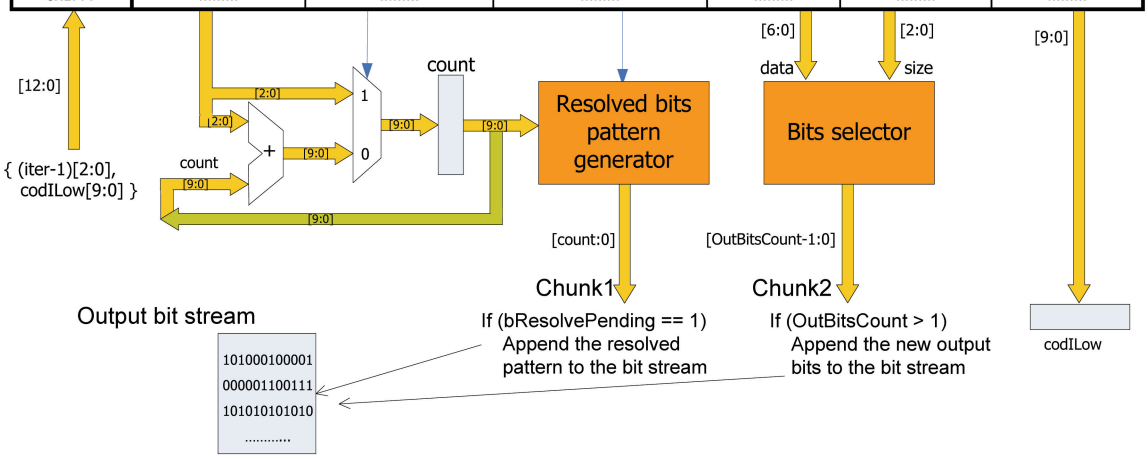


Figure 3.3: Renormalization through table lookup with some sample content.

case. Therefore, a table of $1024 * 7 = 7168$ entries (each of 24 bits) and some surrounding logic as shown in Figure 3.3 can replace the renormalization process and reduce the latency to potentially a single cycle.

Each entry of the table consists of the following fields:

countIncrement is a 3-bit number (ranging from 0 to 7) that specifies the number of "trailing" outstanding-bits generated by renormalization process associated with the current bin. These bits are unresolved so their count will update the *count* register either by adding to or replacing it.

bResolvePending is a boolean value that indicates this renormalization is resolving the pending outstanding bits. Basically, this bit is on whenever a non-outstanding bit exists in the output string generated from renormalization of the current bin.

ResolveBitPolarity is a one bit value that shows the polarity of the leading non-outstanding bit in the output string. If *bResolvePending* is off, this field does not carry useful information.

NewOutBits is a string of non-outstanding output bits generated by current renormalization. It keeps up to six output bits since the leading non-outstanding bit is indicated in *ResolveBitPolarity* field so a total of up to seven output bits (matching the maximum seven iteration) can be represented. The number of valid bits in this field is determined by *OutBitsCount*.

OutBitsCount is a 3-bit number that identifies number of valid bits in *NewOutBits* and can range from 0 to 6.

NewCodILow is a 10-bit value that reflects the updated value of *codILow* after renormalization of current bin.

The *Resolved bits pattern generator* block generates a pattern of resolved bits which the polarity of its first bit is the same as *ResolveBitPolarity* and is followed by *count* bits of the opposite polarity. The *Bits selector* block picks the top $OutBitsCount$ bits of the $NewOutBits$ field of the table entry. This is necessary as the number of valid bits in $NewOutBits$ field is variable so the right bits have to be picked from the 6-bits $NewOutBits$ field. Note that the $Chunk1$ bit string is appended to the output bitstream only when *bResolvePending* is on which means pending outstanding bits, if any, can be resolved now. The details of the method used for appending the two bit strings *Chunk1* and *Chunk2* (with variable sizes of $count + 1$ and $OutBitsCount$

respectively) to the output stream is skipped till a more mature architecture is presented.[3] Also, here a simple open-ended buffer for the output bitstream is assumed which will be much more complex in reality.

The table is addressed through appending bits of $iter - 1$ value to the 10-bit $codILow$ forming a 13-bit address. The top 3 bits of address are set to *3'b(iter-1)* because no renormalization is required when $iter == 0$. Therefore, the table entries start from *iter* equal to 1. The content of the table is precalculated through simulating the renormalization flowchart for all 7168 possible input cases (addresses) of the process. Some entries might never be accessed meaning that their corresponding addresses are never generated. Such an address is *13'h0AF0* which corresponds to $codILow$ value of 0x3F0 and *iter* value of 3. Such combination is not possible as an *iter* value of 3 means that $32 \leq codIRange < 64$ which can not be paired up with $codILow$ of 0x3F0. This is because the end point of such interval region would surpass 1023 falling outside the *legal range* (see sections B.1 and B.1.2). It is not easy to assign useful entries to such unused addresses unless much extra logic is added to remap the addresses so such holes in the address space are tolerated. To better understand how the table content is derived and accessed, below examples are provided. Each case corresponds to the same case in example of section 3.3.1. The table entry for each case is presented in Fig. 3.3.

Case 1- CodIRange=122; CodILow=502: Since both generated bits are outstanding bits, *countIncrement* will be set to two in order to increment *count* by 2. Since $bResolvePending$ is $false$, *count* is "incremented" by *countIncrement* instead of being "set" to *countIncrement*. The contents of *ResolveBitPolarity* and *NewOutBits* are "don't care". *NewCodILow* is set to 472 and *OutBitsCount*

[3]Here a 10-bit register is allocated for *count* suggesting that it is assumed that the maximum number of outstanding bits will not go over 1023. Further discussion of this assumption will be given in section 4.3.3.

set to 0 as there is no resolved bits. The address associated to the corresponding entry in the table is $\{iter - 1, codILow\} = \{3'b001, 10'b0111110110\} = 13'h05F6$.

Case 2- CodIRange=3; CodILow=932: The partially resolved bit string of $\{1110101+\}$ carries fully resolved $\{\mathbf{111010}\}$ and a trailing *1+* bit. When there is any resolved bit in the generated string, the first such bit is always stored in *ResolveBitPolarity* with *bResolvePending* set to *true*. The reason is that the table entry does not know whether there exists any pending outstanding bit from previous renormalization or not so the first resolved bit is always treated as a "resolve bit" which will be handled properly by the surrounding logic.

The remaining five resolved bits are placed in *NewOutBits* with *OutBitsCount* set to 5. *countIncrement* is set to 1 for the tailing *1+* bit. The previous content of *count* along with the polarity of the resolve bit are used for resolving the pending outstanding bits through generating the right bit pattern by *Resolved bit pattern generator* block. Then, the value 1 of *countIncrement* will overwrite *count*. *NewCodILow* is set to 0. The entry address in the table is $\{iter - 1, codILow\} = \{3'b110, 10'b1110100100\} = 13'h1BA4$.

Case 3- CodIRange=72; CodILow=610: From the resolved output bits $\{\mathbf{10}\}$, the first bit goes to *ResolveBitPolarity* by setting $ResolveBitPolarity = 1$ and $bResolvePending = true$. The remaining resolved bit is indicated by $NewOutBits = \{0\}$ and $OutBitsCount = 1$. There are no outstanding bits generated so $countIncrement$ is set to 0. The associated table entry is located at $\{3'b001, 10'b1001100010\} = 13'h0662$.

Case 4- CodIRange=54; CodILow=362: From the resolved output bits $\{\mathbf{01}\}$, the first bit goes to *ResolveBitPolarity* by setting $ResolveBitPolarity = 0$ and $bResolvePending = true$. The remaining resolved bit is indicated by

$NewOutBits = \{1\}$ and $OutBitsCount = 1$. Because of the trailing outstanding bit in the original partially resolved bits of $\{\boldsymbol{011+}\}$, $countIncrement$ is set to 1. The associated table entry is located at $\{3'b010, 10'b0110001000\} = 13'h0988$.

As seen above, the table size will be $7168 * 24 = 172,032$ bits which makes such implementation very undesirable. Otherwise, rescale of the coding states can be done in a single cycle with a memory access and some combinational logic for producing the output bits.[4] In the next section, a more efficient architecture is presented.

3.3.3 Renormalization through forming a parsing area

This section explores a new method for more efficient handling of renormalization than the ROM-lookup approach of the previous section. At the first glance, the subtraction operations on $codILow$ in *1* and *1+* branches of renormalization flowchart (Fig. 2.13) seems problematic for any unrolling attempt of the renormalization loop. A simple barrel shifter can not be used directly to mimic the cumulative effect of the iterations for $codILow$ update and output bits generation. However, since each subtraction only affects a single bit of $codILow$ (positions 9 for *1* and position 8 for *1+* branch), an *iter*-size left shift of codILow still preserves all necessary information to retrieve both the updated $codILow$ and the generated bits.

At every iteration of Fig. 2.13, always the top bit of $codILow$ is tested to decide whether the *1* branch is taken or not. If that bit is off, the next top bit (the 8-th bit) is tested to decide which one of the *1+* or *0* branches to be taken. The cumulative effect of repeating *iter* iteration will be test of the top *iter* bits of $codILow$ along with potential test of the next neighboring bit.[5] As a result, the subfield $codILow[8 : 8 - iter]$ (of $iter + 1$ size) needs to be tested. To include the effect of

[4]Note that the logic for appending generated bits to the output stream is excluded from this discussion at this point.

[5]The term "potential" used to reflect the fact that depending on the polarity of $codILow$'s bit at position *(9-iter+1)*, bit *(9-iter)* may need to be tested too.

update of $codILow$, an alternative but equivalent interpretation can be made. First, $codILow$ is barrel-shifted to the left for a size of $iter$ and called $ShiftedCodILow$. The top $iter + 1$ bits of the shifted value form a *parsing area*. This area will be the subfield $ShiftedCodILow[9 + iter : 9]$ with a size of $iter + 1$. Now, the generated bits and the updated $codILow$ can be retrieved through the following rules:

Rule 1: By parsing the $iter + 1$ bits of the *parsing area* from left to right, the $iter$-sized generated bit string can be retrieved. The proof is that the parsing area has enough information because it includes "all the bits from $codILow$" that the renormalization algorithm tests for "making decision about the taken branches" that generate the output bits subsequently.

Rule 2: Only the leading *1*'s in the parsing area are real generated *1* bits and correspond to taken *1* branches. There is an exception to this rule when all the top $iter$ bits of the parsing area are *1*. This case is called *All 1's case* for future reference. In this case, the last bit of the parsing area, $ShiftedCodILow[9]$, does not correspond to any generated bit since only the top bit of $codILow$, $codILow[9]$, is tested at every iteration. In other words, this is the only case that $codILow[8]$ bit is never tested in any iteration which means it will not carry information about the generated bits. In the last iteration, $codILow[8]$ will end up in $codILow[9]$ after the final shift. This means that the final $codILow[9]$, i.e. $ShiftedCodILow[9]$, was never used in branch decision making so even if it carries a *1* bit, it is not a generated output bit.

Rule 3: All non-leading *1*'s in the parsing area are of *1+* (outstanding) type.

Rule 4: The first *0* bit encountered in the parsing area does not correspond to any taken branch and to be ignored. To prove it, two cases are considered. In the first case, the *0* bit is following one or a set of leading *1's* in the parsing area.[6]

[6]This excludes the *All 1's case* described in **Rule 2**.

In the iteration which this first *0* bit reaches the 8-th bit of $codILow$, it will not be tested as $codILow[9]$ is a *1* bit. In the next iteration, the *0* bit is at the 9-th position of $codILow$ but the generated bit (*1+* or *0*) is decided based on $codILow[8]$ so the first *0* bit does not specify the branch outcome in this case.

In the second case, the first *0* bit happens to be the top bit of the parsing area ($ShiftedCodILow[9 + iter]$). This case happens when $codILow[9]$ is *0* from start of the renormalization. Because the top bit is *0* in the first iteration, the next bit ($codILow[8]$) will decide about the first taken branch. As a result, the first *0* bit in the parsing area does not determine any taken branch here too.

Rule 5: All other *0*'s (excluding **Rule 4**) in the parsing area correspond to the generated *0* bits.

Rule 6: The generated outstanding bits detected based on **Rule 3** can be resolved right away if they are followed by a non-outstanding bit from the parsing area. Note that such a non-outstanding bit can be of *0* type only because in renormalization of a bin, a *1* bit can not be generated after a *1+* bit per **Rules 2** and **3**. This case is discussed further in section 3.4.2.

Rule 7: The updated $codILow$ can be retrieved from $ShiftedCodILow$. It can be easily verified that the renormalized $codILow[8:0]$ subfield will be equal to $ShiftedCodILow[8:0]$. The only issue is to determine its top bit, $codILow[9]$. The below two cases must be considered to determine $codILow[9]$ after renormalization:

Case 1- The parsing area is a *All 1's case*: Here, *ShiftedCodILow[9]* was never tested for branch decision making as described in **Rule 2**. As a result, *ShiftedCodILow[9]* remains part of the renormalized $codILow$. This means the new $codILow$ can be retrieved easily through setting

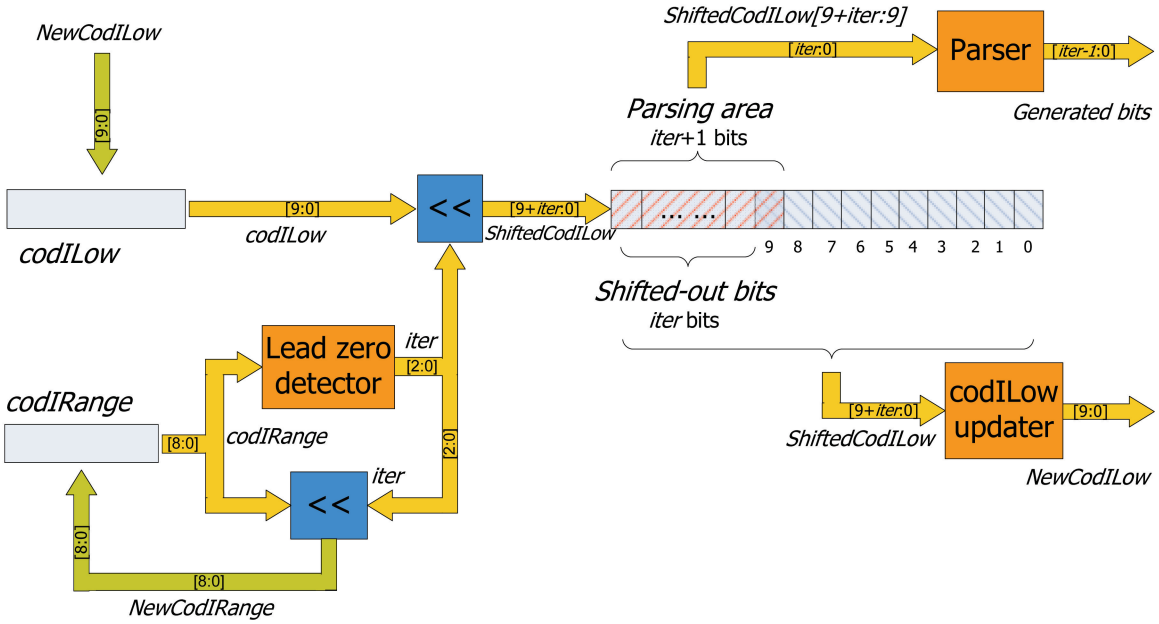


Figure 3.4: Renormalization through forming a *parsing area* and interpreting it.

$$codILow[9:0] = ShiftedCodILow[9:0].$$

Case 2- The parsing area is not a *All 1's case*: In this case, the first *0* in the parsing area is ignored according to **Rule 4**. This means that $ShiftedCodILow[9]$ has been used in determining the last taken branch. It is easy to verify that any bit used in deciding the taken branches is set to *0* in the renormalization loop and will not reach the upper positions with successive left shifts. As a result, the renormalized $codILow$ will receive a 0 bit for its 9-th bit and will be equal to $\{1'b0, ShiftedCodILow[8:0]\}$.

The new rescaled $codILow$ can be easily calculated through $ShiftedCodILow$ based on **Rule 7**. The block marked as *codILow rescaler* in Figure 3.4 implements **Rule 7** using combinational logic. **Rule 1** to **Rule 5** help construct a *Parser* block using combinational logic to obtain the generated bit string for renormalization of the current bin. The latency of the parser can be potentially reduced to a single cycle.

The details of parser implementation depend on how the generated bits are resolved and delivered out of CABAC. Two possible design choices are discussed in the next section.

3.4 Parser design and generated bits packing

The idea of forming a parsing area and processing it to retrieve the generated bits was introduced in the previous section. Here, the details of parser implementation, temporary storage of the generated bits and their transfer to the output bitstream are studied.

As discussed earlier in section 3.3, encode of each bin can generate zero to seven output bits. The polarity of any outstanding bit will be determined by generation of the next non-outstanding bit. Arrival of this non-outstanding bit could happen either in renormalization of the same bin or a future bin which could be potentially several bins away. Remember that the output interface of the CABAC block is through a FIFO interface as depicted in Fig. 2.8. Two main reasons dictate use of an *intermediate buffer* before the output FIFO:

- Encode of each bin generates variable number of output bits. It is not efficient to allocate a FIFO entry to each generated bit string. An intermediate buffer needs to sit between the parser and the output FIFO to pack enough bits for filling a FIFO entry before the write to FIFO happens.

- Besides the packing issue, outstanding bits need to be tracked and buffered till their polarity can be resolved by the upcoming resolve bit.

In the following sections, two approaches for implementation of the parser and the intermediate buffer are explored. The first approach can only resolve one outstanding bit of the intermediate buffer at every cycle which could lead to overflow of the

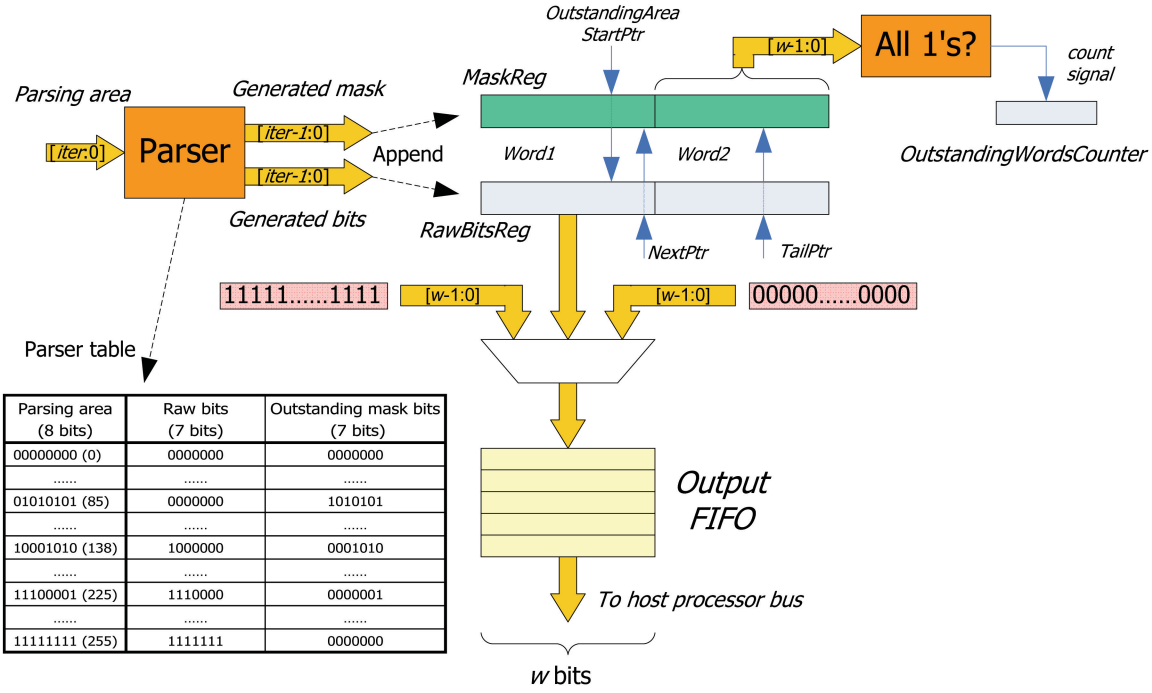
intermediate buffer. The second approach builds on the strengths of the first approach and implements a more efficient solution. It is capable of resolving the whole pending outstanding bits area at once with arrival of a resolve bit.

3.4.1 Single-bit resolver

Rule 1 to **Rule 5** described in section 3.3.3 can be easily implemented through a ROM lookup table. The number of entries in the table will be $2^8 = 256$ as the maximum size of the parsing area is 8 (the maximum size of *iter* is 7). Since each generated bit identifies one of the three states of *1*, *1+* or *0*, two bits will be necessary to keep each generated bit after the parsing process. This is done by using an *Outstanding Mask* field which is used to identify outstanding bits. Then the bits in *Raw Bits* field identify the polarity of non-outstanding bits. As a result, each one of these fields carry 7 bits of data, one for each possible generated bit.

To handle bit-packing and outstanding bits issues described in section 3.4, two *intermediate buffers* store the incoming mask and the raw bits read from the parser. The new generated bits are appended to the intermediate buffers at the position pointed by *TailPtr* and then *TailPtr* is incremented by the size of generated bits which is equal to *iter*. The width of each buffer is two words when a word is considered to be the width of the FIFO interface. For example, the buffer size will be 64 bits for a typical FIFO width of 32 bits. At every cycle, single bit from each one of the intermediate buffers, *RawBitsReg* and *MaskReg*, pointed by *NextPtr* is read and inspected for outstanding bits. Figure 3.5 shows a simplified form of the procedure.[7] The size of intermediate buffers are picked as twice wide as the FIFO width to allow

[7]Some details are dropped to avoid the figure complexity. When size of a parsing area is less than 8, it is extended from the right side with 0's. After each read from the parser, each output field is truncated to its top *iter* bits before being appended to the intermediate buffers because *iter* bits are to be generated in renormalization process. Another missing detail is a single-bit register that keeps "the last resolved bit". It is used to decide the type of potential full word ones or zeros sent to the output FIFO after resolve of the pending bits.



Parsing area (8 bits)	Raw bits (7 bits)	Outstanding mask bits (7 bits)
00000000 (0)	0000000	0000000
.....	.....	.....
01010101 (85)	0000000	1010101
.....	.....	.....
10001010 (138)	1000000	0001010
.....	.....	.....
11100001 (225)	1110000	0000001
.....	.....	.....
11111111 (255)	1111111	0000000

Figure 3.5: Bit generation with one bit per cycle resolver.

receiving more incoming generated bits from the parser into the second word while the first word is processed for packing enough resolved bits to be sent to the FIFO.

At any time, the *NextPtr* register points to the generated bit in the intermediate buffers to be processed next. After the next bit is picked from *RawBitsReg* and *MaskReg*, one of the following cases happens:

An outstanding bit is met, *MaskReg[NextPtr] == 1:*

- If this is the first outstanding bit met[8], *OutstandingAreaStartPtr* is set to current bit's position, *NextPtr*.
- If this outstanding bit is in continuation of an outstanding area, normally nothing extra needs to be done as the start position is already recorded. An outstanding area does not have much useful information and it is enough to

[8]This condition is indicated by *OutstandingAreaStartPtr* having an invalid pointer, e.g., 64 for an intermediate buffer size of 64. This means pointer values in [0, 63] are valid.

track its size and start position. To prevent long sequence of outstanding bits from filling up the intermediate buffers, whenever the second word of the intermediate buffers is completely comprised of an outstanding area, the whole second word can be removed and tracked through *OutstandingWordsCounter*.[9] By update of the relevant pointers, now the second word becomes completely available for the future generated bits.

A non-outstanding bit is met, *MaskReg[NextPtr] == 0:*

- If there exists a pending outstanding area (i.e., *OutstandingAreaStartPtr* pointing to a valid position in the intermediate buffers), then the current bit will be a *resolve bit*. A *Resolve bit pattern generator* (as the one in Fig. 3.3 but not shown in this figure) will resolve the outstanding bit string of size $NextPtr - OutstandingAreaStartPtr$ with resolve bit polarity of $RawBitsReg[NextPtr]$ and will replace the existing bits in *RawBitsReg[OutstandingAreaStartPtr:NextPtr]*. Since outstanding bits are resolved, *OutstandingAreaStartPtr* will be set to an invalid position.
- If there is no pending outstanding bit, nothing needs to be done. Next cycle will process the next bit.

After processing the current bit, *NextPtr* is advanced to the next location. If *NextPtr* has gone beyond the first word and there is no outstanding area within the first word, the content of the first word carries a fully resolved word which can be sent to the output FIFO. Also, when resolve of an outstanding area happens and *OutstandingWordsCounter* is not zero, a full word of all ones or zeros will be sent to the output FIFO after the first word is sent out. These full words of all ones or zeros make up for the long sequence of outstanding bits that already factored out.

[9]Note that in reality, the intermediate buffers need to have another extra byte to be able to append the potential generated bits beyond the two words boundary in the same cycle that the long sequence of outstanding bits is supposed to be reduced. More details are skipped here.

The multiplexer shown before the FIFO is meant for this purpose. *OutstandingWordsCounter* is decremented and the same process is repeated at the next cycles till *OutstandingWordsCounter* reaches zero. Note that all mentioned position pointers above can be used for accessing bits in both intermediate buffers *MaskReg* and *RawBitsReg* because the mix of both registers together will determine the real type of a bit out of the three possible states of *0*, *1* and *1+* at anytime and the pointers point to both registers at the same time.

At every cycle, up to seven generated bits are potentially fed to the intermediate buffers from the parser. However, only a single bit from intermediate buffers is probed and potentially resolved at every cycle. This does not necessarily lead to an overflow as the total number of generated bits will be lower than the total number of encoded bits (one bit encoded per cycle) in the "long term" as the input bins are compressed by CABAC. Anyway, there will be times that the intermediate buffers are fed with generated bits continually and get filled up quickly. Though long sequence of outstanding bits are reduced in the second word by tracking them as outstanding words instead of bits, long sequence of non-outstanding bits waiting for further processing could easily fill up the intermediate buffers and cause overflow.[10]

Using a wider word size will delay the overflow[11] but this shows a fundamental limitation of this approach where "ready for output" non-outstanding bits which do not need further processing could cause an overflow very easily. The next design will tackle this issue.

[10] A worst case scenario for a word size of 16 bits could fill the buffers as soon as 5 cycles ($\lceil (2 * 16)/7 \rceil$). The five bits processed in these 5 cycles would not be enough for a FIFO output write.

[11] As an example, simulation of intermediate buffers with a word size of 16 bits quickly overflowed after encoding a few hundred bins in a test using *foreman* clip. With a word size of 32 bits no overflow happened with a test of 1000 bins.

3.4.2 Chunk resolver

One might suggest that a simple solution to improve on the previous method is to detect the case when all the generated bits from the parser in a cycle are of the non-outstanding type. Then all bits would be processed together and *NextPtr* incremented by the chunk size, i.e., *iter*. This could have worked if at any time, all bits in the intermediate buffer were already processed. But this could not be the case when the previous generated bits included some outstanding bits too and the *prober*[12] has started to lag the parser. Recording the position of non-outstanding areas to allow *NextPtr* jump over them is not an easy task either as there could be multiple discontinuous sequences of non-outstanding areas in the intermediate buffers.

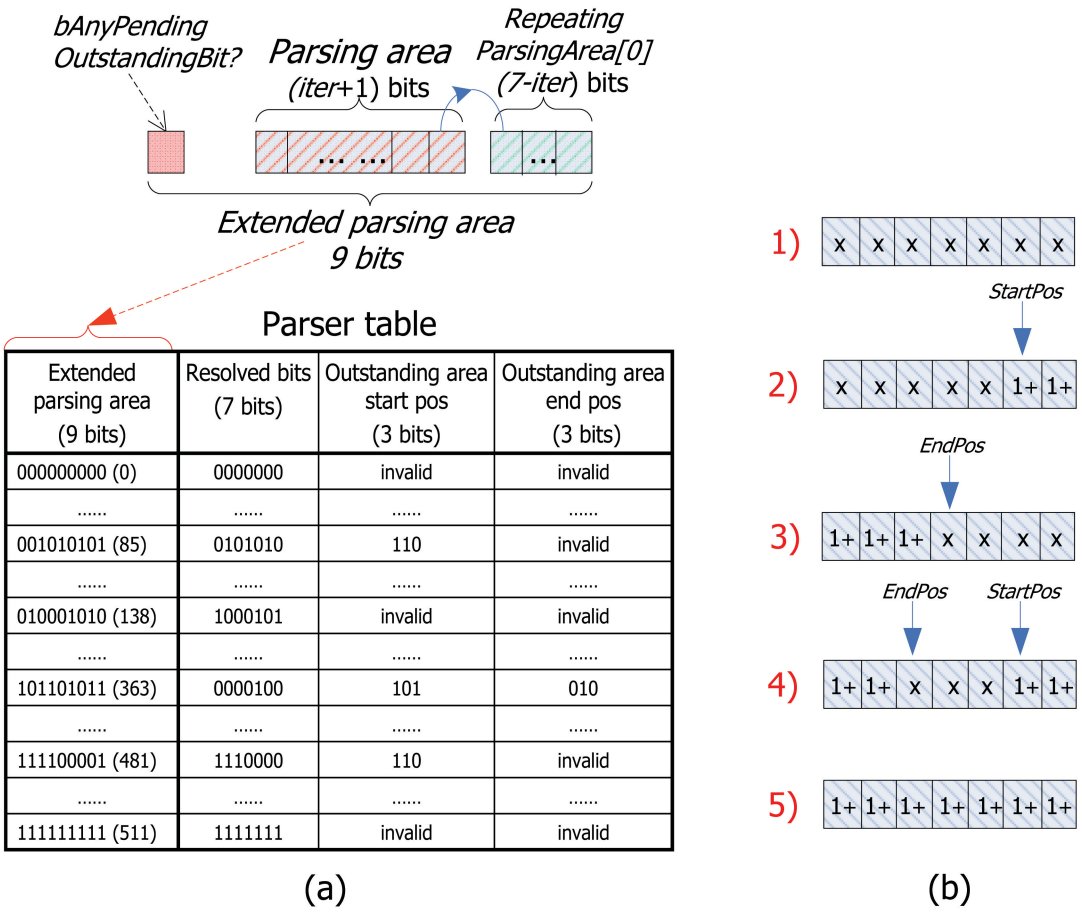
The ideal solution is to interpret the whole chunk of generated output bits arriving from the parser on the fly so the prober is not lagging the parser anymore. Actually, there is no need to have a prober picking a bit from the intermediate buffers at every cycle. Instead, the chunk of generated bits is processed at once at every cycle. Then the result is appended to the intermediate buffer and the tracking pointers are updated.[13]

Unless the generated bits chunk (of *iter* size) is ending with outstanding bits, the chunk could be fully resolved before being appended to the intermediate buffer. Otherwise, it still can be partially resolved before the append operation. Of course, the resolve process of the chunk content needs to take into account whether a pending outstanding bit exists in the intermediate buffer or not as it would affects the resolve procedure. This process of chunk resolve which is internal to the generated bit string is called *internal resolve*. The internal resolve can be integrated to the parser logic to produce *internally resolved bits* right away.

Figure 3.6-(a) shows a new ROM structure to implement such improved parser.

[12] It is the logic that tests and resolves the next outstanding bit in the intermediate buffer.

[13] As discussed later, there will be no need to use two intermediate buffers with this new approach.



Extended parsing area (9 bits)	Resolved bits (7 bits)	Outstanding area start pos (3 bits)	Outstanding area end pos (3 bits)
000000000 (0)	0000000	invalid	invalid
.....	.....	.....	.....
001010101 (85)	0101010	110	invalid
.....	.....	.....	.....
010001010 (138)	1000101	invalid	invalid
.....	.....	.....	.....
101101011 (363)	0000100	101	010
.....	.....	.....	.....
111100001 (481)	1110000	110	invalid
.....	.....	.....	.....
111111111 (511)	1111111	invalid	invalid

Figure 3.6: (a) New parser table (b) Different scenarios after internal resolve of generated bits.

Besides the old 8-bit parsing area, the existence of pending outstanding bits in the intermediate buffer need to be considered as discussed above. This adds an extra bit (to the left) of the parsing area increasing the table size to $2^9 = 512$ instead of 256 entries in the previous parser table. Since the number of generated bits, *iter* is usually less than the maximum possible of seven, the table address can not be created right away. Instead, another table entry resulting in equivalent output needs to be addressed. Such equivalent entry can be found by repeating the least significant bit of the parsing area, *ParsingArea[0]* and truncating the outputs by

iter.[14] Each table entry stores three fields. *ResolvedBits* field carries the partially resolved generated bits. *StartPos* points to the index within the *ResolvedBits* that outstanding area starts. *EndPos* points to the *resolve bit*, the first non-outstanding bit following an outstanding area. Since a non-outstanding bit at the beginning (top bit) of *ResolvedBits* would behave as the resolve bit for potential pending outstanding bits in the intermediate buffer, the invalid position in table of Fig. 3.6-(a) is picked as 0 and *EndPos* will be set to 0 in this case.[15]

Fig. 3.6-(b) shows the possible scenarios after internal resolve of the parsing area. It could be one of the following five scenarios:

1. No outstanding bit was generated or if there was any, it was resolved by a following non-outstanding bit within the generated bit string. As a result, both *StartPos* and *EndPos* fields point to the neutral positions of 0. If there are pending outstanding bits in the intermediate buffer, they will be resolved by the most left bit of *ResolvedBits*. Entries of *0*, *138* and *511* marked in the table are of this type.

2. Any potential outstanding bits area within the first part of the generated bits is already resolved (if any). A new outstanding area starts from position *StartPos* which remains unresolved. Entries of *85* and *481* are of this type.

[14]Instead of finding an equivalent entry in the parser ROM, one could have used multiple ROMs (with different sizes) each resolving a fixed-sized parsing area e.g., one ROM for parsing area of size 1, another for size 2, and so on. Instead of allocating these extra tables, same data can be retrieved as if the parsing area was extended. But extension of the parsing area must happen in a fashion not compromising validity of the generated bits. Extending the parsing area from right by repeating the lower bit will not change the top *iter* bits of the generated bits as it is similar to adding neutral bits. Then the *Resolved bits* is truncated from the top with a size of *iter* and *StartPos* field is truncated like $StartPos = (StartPos < iter)?StartPos : invalid$ since any change in the outstanding area beyond the first *iter* bits is not visible to these generated bits. Same applies to *EndPos* field too.

[15]Similarly, a value of 0 means an invalid position for *StartPos* in the parser table too. After partial resolve of the generated bits, the only case that the resolved bit could start with a outstanding bit right from position 0 which is not resolved by a following non-outstanding bit is the case where all seven bits are outstanding bits. This case is signalled with *StartPos* of 7.

3. The generated bits start with an outstanding area which ends at location *EndPos*. The bit will be used for resolve of earlier outstanding bits in *ResolvedBits* field and potential pending outstanding bits in the intermediate buffer. Any potential outstanding bits area after *EndPos* in generated bits is already resolved.

4. This is a combination of the two above cases. One outstanding area is resolved by the bit at *EndPos* but a new one starts from *StartPos* which remains unresolved. Entry *363* is of this type.

5. This scenario is associated with only two entries in the table where all generated bits are of *1+* type. These are entries *9'b001111111 = 127* and *9'b101111111 = 383* and signalled by $StartPos = 7$ which is exclusively used for this scenario.

Note that unresolved outstanding bits are marked as *0* in the corresponding bits of *ResolvedBits* since the area location is already identified by *StartPos* and *EndPos*. The examples in the parser table of Fig. 3.6-(a) are derived through this:

9'b000000000 (0): The generated bit string will be {*0000000*} (of type 1) so all are resolved and $StartPos = EndPos = invalid$.

9'b001010101 (85): The generated bit string will be {*1+01+01+01+*} which can be partially resolved to {*0101011+*}. It is of type 2 so $StartPos = 6$ and $EndPos = invalid$.

9'b010001010 (138): The generated bit string will be {*1001+01+0*} which can be fully resolved to {*1000101*}. As a result, it will be of type 1 so $StartPos = 6$ and $EndPos = invalid$.

9'b101101011 (363): The generated bit string will be {*1+1+01+01+1+*} which can be partially resolved to {*111011+1+*}. Considering that there was pending outstanding bit in the intermediate buffer waiting to be resolved using the

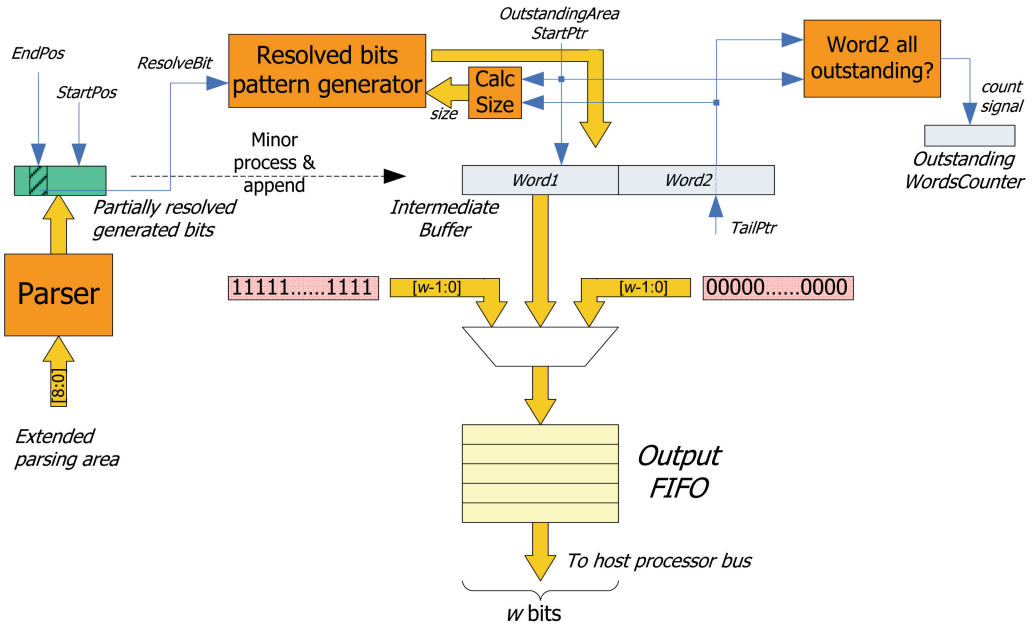


Figure 3.7: Improved bit generator.

resolve bit at $EndPos = 2$, the resolve bit and leading outstanding area are restored as $\{1{+}1{+}0011{+}1{+}\}$. This will be stored as $\{0000100\}$ in *ResolvedBits* field. It is of type 4 and $StartPos = 5$ to indicate existence of the trailing outstanding area.

9'b111100001 (481): The generated bit string will be $\{1110001{+}\}$ which is of type 2. As a result, $StartPos = 6$ and $EndPos = invalid$. Because of pending outstanding bit, the bit at $EndPos = 0$ will be used as the resolve bit.

9'b111111111 (511): The generated bit string will be $\{1111111\}$ (of type 1) so all are resolved and $StartPos = EndPos = invalid$.

High level architecture of the complete bit generation process is shown in Figure 3.7. Because the whole chunk of generated bits is now partially resolved in a single cycle, the two intermediate buffers shown in Fig. 3.5 are no longer necessary and

one buffer is sufficient. Now it is only enough to store the start position of the last unresolved outstanding area through *OutstandingAreaStartPtr* register rather than having a complete mask register. As before, *Resolved bits pattern generator* creates the right resolve pattern based on the size of pending outstanding bits and polarity of the *resolve bit*. To detect if the second word of the intermediate buffer consists of all outstanding bits, it will be enough to check the start position of outstanding area and the position of the last bit in the intermediate buffer (which is definitely of outstanding type since outstanding area is not resolved yet) through a relation like:

$$bCanReduceTheSecondWord = \quad (3.1)$$
$$OutstandingAreaStartPtr < w \;\; \&\& \;\; TailPtr \geq 2 * w$$

OutstandingWordsCounter keeps track of full words of outstanding bits factored out from the second word of the intermediate buffer. The output multiplexer on the interface to the output FIFO directs full word of all ones or zeros (depending on the resolve bit) at proper cycle when the long sequence of outstanding area is resolved.

No need to mention this approach is much more efficient than the previous method of section 3.4.1 as all the generated bits from the parser are processed at once. Though still chance of overflow exists, its probability is significantly reduced and will not happen in most practical scenarios. More details on this and the choice of word size will be provided in section 4.2.

3.5 Bypass coding implementation

In the previous sections of this chapter, the full encode path for a regular coded bin was followed from the arithmetic coding, renormalization and output bit generation. Now the case of a bypass coded bin is considered.

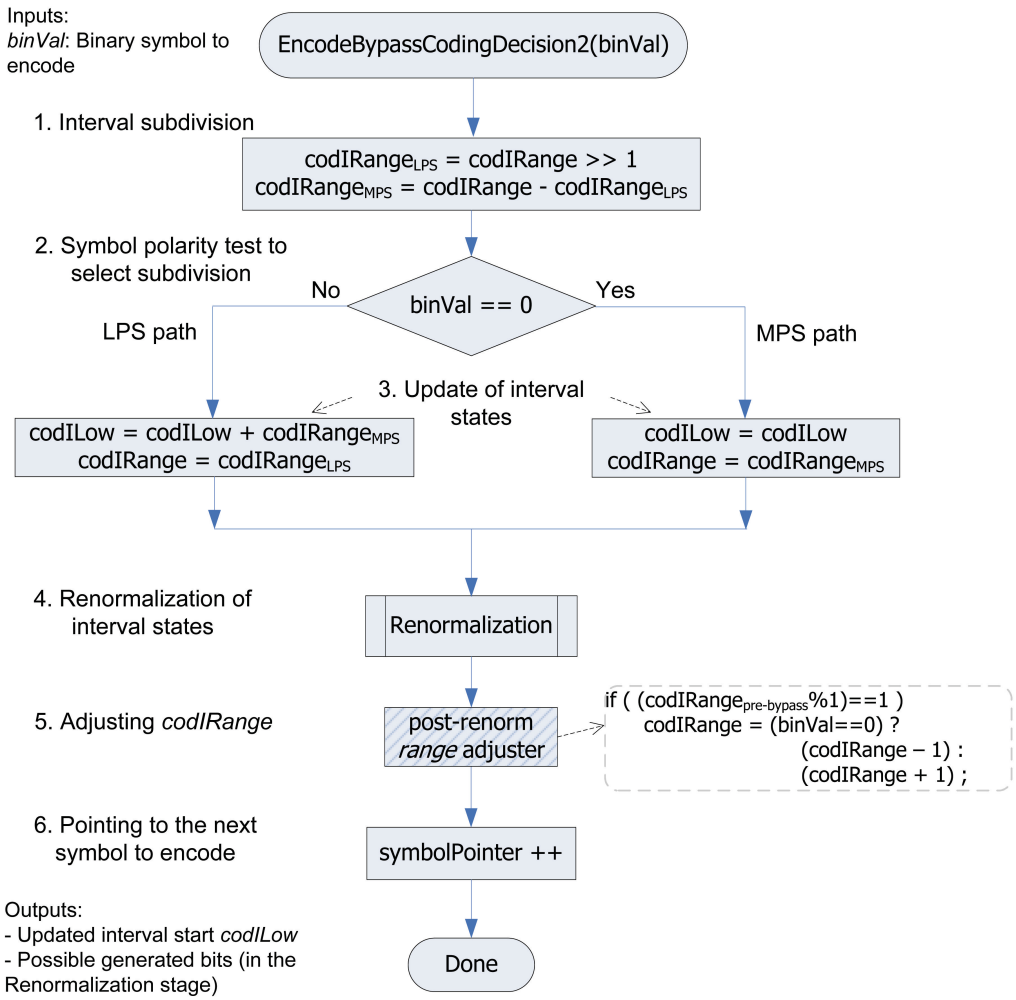


Figure 3.8: An equivalent bypass coding using the same steps as regular coding.

Bypass coding as presented in the flowchart of Fig. 2.15[16] seems to require a separate logic for update and renormalization of the coding states. Further, the "early rescale" of $codILow$ in Step 1 of Fig. 2.15 and subsequent steps require temporary operation on $codILow$ with width of 11 bits rather than the usual 10 bits. Steps 3 to 5 resemble a regular renormalization of Fig. 2.13 but with only a single iteration.

[16]The figure is is based on the standard document [3]

Also, all condition tests of Step 3 and interval shift of Step 4 (bit resets) are done with values double of the values in regular renormalization.

It can be shown that the same algorithm can be modified into an almost equivalent flowchart in Fig. 3.8. This presentation is more intuitive considering the fact that bypass coding is basically a regular coding but with equiprobable subdivision of the coding state (as if a most probable symbol of 1, i.e. $valMPS = 1$ is picked). Since the range is divided to half and $codIRange \geq 256$ before the start of bypass coding, the renormalization after subdivision will require only a single iteration to rescale $codIRange$ back to a value more than or equal to 256. Note that because $codIRange_{LPS} = codIRange >> 1$, the following is valid:

$$\begin{cases} \text{if } codIRange \text{ is even} \Rightarrow codIRange_{MPS} == codIRange_{LPS} \\ \text{if } codIRange \text{ is odd} \Rightarrow codIRange_{MPS} == (codIRange_{LPS} + 1) \end{cases} \tag{3.2}$$

This makes sense based on the equiprobable subdivision. In case $codIRange$ is odd, $codIRange_{MPS}$ will be assigned one unit more than $codIRange_{LPS}$. If MPS is taken and $codIRange$ was an odd value, this will result in a range value equal to $2 * codIRange_{MPS} = 2 * (codIRange_{LPS} + 1) = codIRange_{pre-bypass} + 1$. [17] This means that in such special case, the interval size will be one more than the size before bypass coding which contradicts the original bypass coding behavior that leaves *codIRange* untouched. Similarly, if LPS path is taken and $codIRange$ was an odd value, this will result in a range value equal to $2 * codIRange_{LPS} = 2 * ((codIRange_{pre-bypass} - 1)/2) = codIRange_{pre-bypass} - 1$. This means the interval size is one less than the original bypass coding behavior. A simple post-renormalization step like the one at step 5 of Fig. 3.8 will be required to take care of both cases.

The standard document presents an special implementation for bypass coding

[17] $codIRange_{pre-bypass}$ represents the original value of $codIRange$ before start of bypass coding.

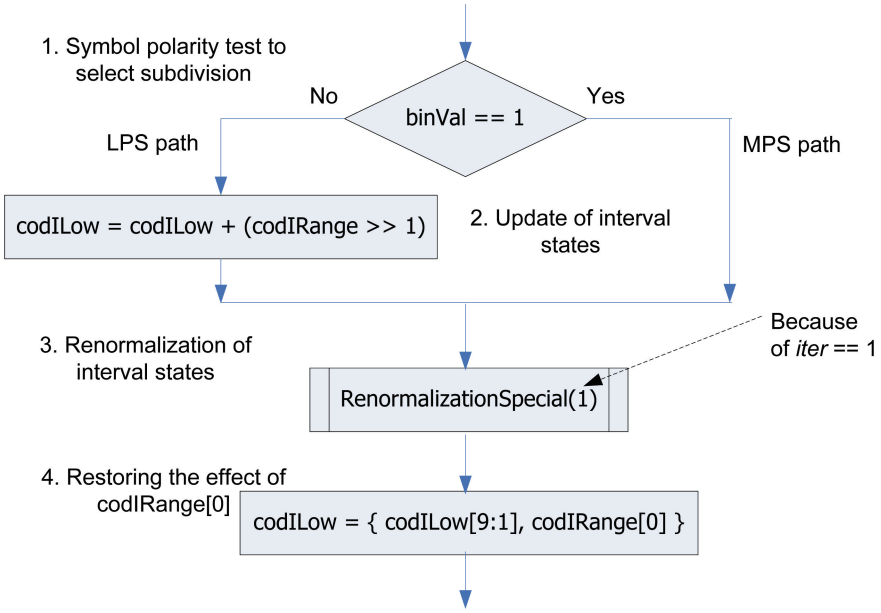


Figure 3.9: Simplified bypass coding.

(shown in Fig. 2.15) to potentially result in faster implementations (software or hardware). But it is favorable to reuse existing components which were already developed for regular coding. One possible way to streamline bypass coding into regular coding is to add a "special context" into the context table and assign it a special "non-adaptive probability state" which always stays at $p_{LPS} = \frac{1}{2}$ to achieve equiprobable subdivision. Then whenever a bypass-coded symbol is sent to CABAC, a regular coding operation is done but with the context index of the special context instead. The non-adaptive probability state guarantees that the updated probability state remains the same at $\frac{1}{2}$ regardless of the polarity of the encoded bin.

There are a few difficulties with this approach. The next probability state table, *TransIdx*, is already fully allocated for all its 64 entries. Adding a new state would require increasing the state index width from 6 to 7 which affects the size of context table subsequently and rules out this approach. Even ignoring the size issue, the calculation through multiplier table will not be exactly equiprobable because it is

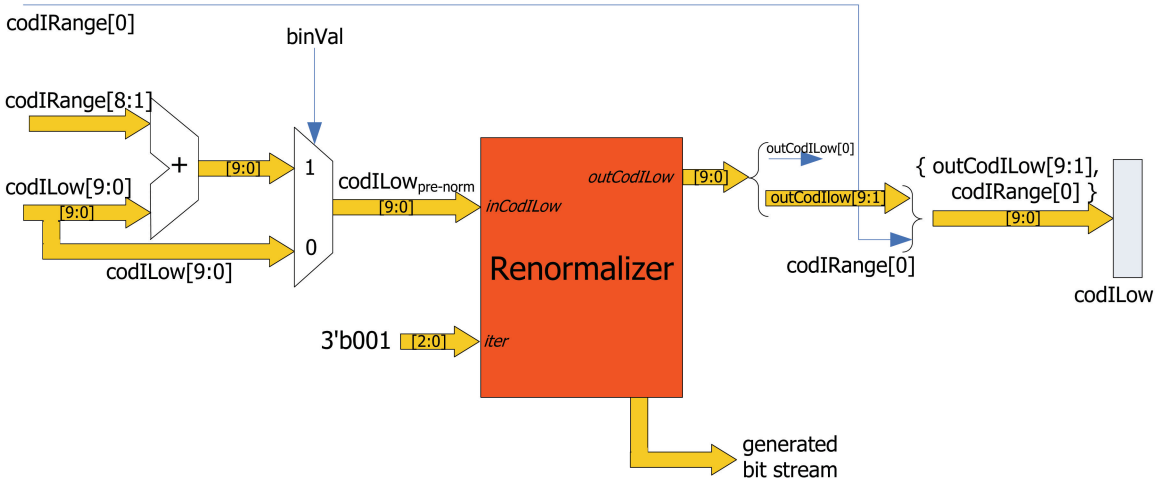


Figure 3.10: An architecture for implementation of bypass coding.

approximated through only two bits of $codIRange[7:6]$.

The best choice seems to be adding some minimal logic for updating the coding states and then using the regular renormalizer. The next chapter will show that rescale of $codIRange$ can be separated from renormalization process. Renormalizer receives number of iterations *iter* instead. It is obvious that the end result of $codIRange$ in Fig. 3.8 will be the same as its original value before start of the coding (first a divide by 2, then a left-shift in renormalization). As a result steps 1 to 4 of Fig. 3.8 can be rewritten as Fig. 3.9 (using the modified renormalizer which takes *iter* as input instead of $codIRange$).

A hardware implementation of this simplified algorithm using the special renormalizer mentioned above is shown in Figure 3.10. More details about employing this architecture in a complete solution will be given in the next chapter. Bypass coding, similar to regular coding, generates valid coding states conforming to Equations B.1, B.2 and B.3. Refer to section B.1.3 for the proof.

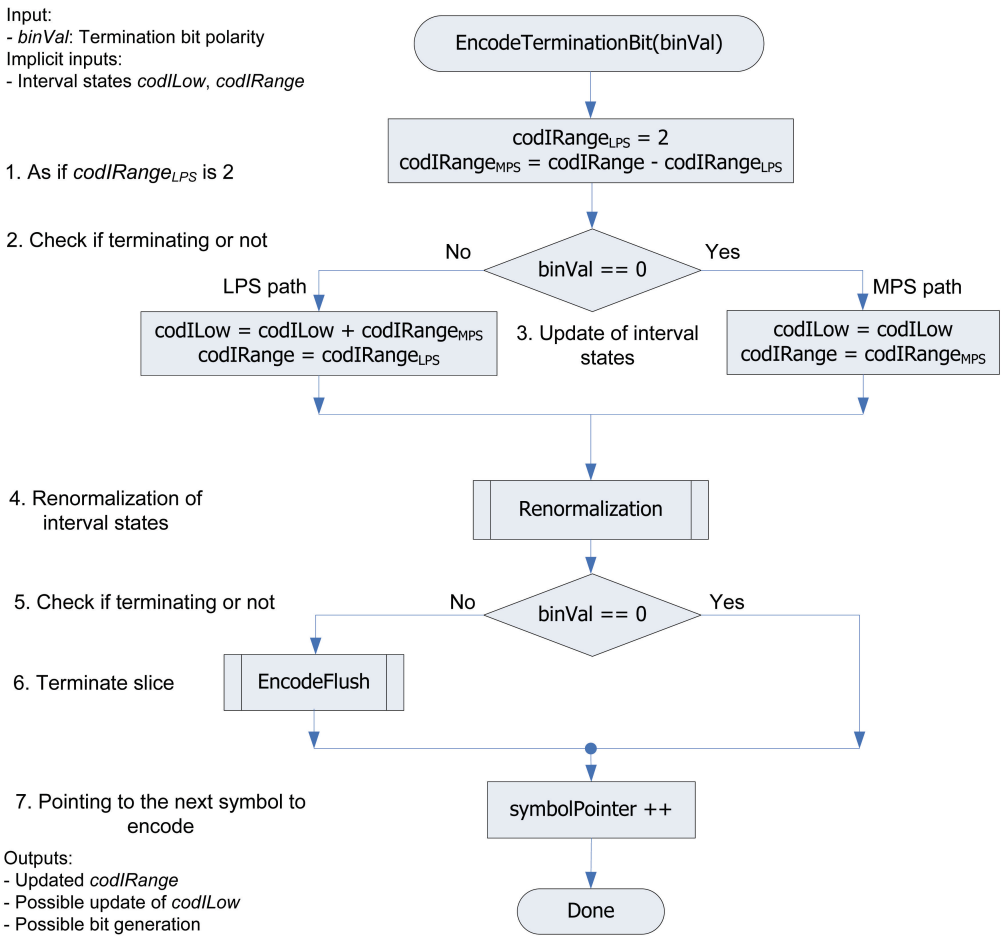


Figure 3.11: Rearranged encode of termination flag.

3.6 Termination coding implementation

The termination flag is encoded at the end of each macroblock to indicate whether the end of slice has reached or not. The flowchart of Figures 2.16 defined in the standard document [3] can be rewritten as Fig. 3.11 where its steps 1 to 4 are exactly like a regular coding with $valMPS = 0$ and fixed $codIRange_{LPS}$ set to 2.[18]

It is surprising that the standard document introduces the termination process

[18] The fixed MPS is set to 0 naturally because most of the termination flags encoded are not actually terminating the slice ($binVal == 0$).

as Fig. 2.16 and 2.17 while at the same time allocates special context index and probability state for it. The context entry with index 276 is dedicated for this purpose which is initialized with probability state index 63 and $valMPS$ of 0. Probability state 63 is a non-adaptive state which stays at 63 no matter what type of bin (MPS or LPS) is encoded.[19] The multiplier table returns 2 for a probability state of 63 no matter what $codIRange$ carries. This means that for encoding termination flag, it will be enough to do regular encode of the flag using context index 276. In case the termination flag indicates end of slice, extra flush operation will be performed too.

Termination flag generates valid coding states conforming to Equations B.1, B.2 and B.3. For proof, refer to section B.1.4. The next chapter integrates all the building blocks discussed here to build a complete arithmetic encoder.

[19]Refer to footnotes 12 and 13 in chapter 2 for more details.

Chapter 4

The complete architecture

The previous chapter, presented more detailed analysis of the arithmetic encode process and discussed possible architectural implementation of its parts. Here, all parts are put together to form the full architecture first. This design can process a new bin at every three cycles. The performance improvement of the proposed architecture and further enhancements to achieve single-cycle throughput are explored next.

The design and its performance are improved through several stages. The first attempt will use a more efficient dispatch technique for issuing input bins to the regular and bypass coding pipes. This approach can achieve an average throughput of one bin per every 2.64 cycles. The next step transform the whole architecture towards a fully pipelined design with a single-cycle throughput. This design clearly outperforms the previous ones.

At the end of the chapter, some other issues and design choices (like context table initialization) are discussed. Although the architecture descriptions given here are tried to be detailed, they still lack some technicalities. The HDL (hardware description language) code of the designs should be consulted for the actual implemented architectures.

4.1 Putting all together

Putting together *Regular Coding* part described in section 3.2, *Renormalization* and *Bit Generation* parts described in sections 3.3.3 and 3.4 respectively, and the modified *Bypass Coding* part described in section 2.5.4 will form a complete architecture for CABAC's arithmetic coding. Before describing this complete architecture, the effect of decoupling bit generation and rescale of coding states is discussed first.

4.1.1 Decoupling bit generation and coding states rescale

Section 3.3.3 discussed implementation of the renormalization algorithm depicted in the flowchart of Fig. 2.13. The modified bypass coding algorithm described in section 2.5.4 takes advantage of the generic renormalization algorithm for the regular coding mode. As a result, the renormalization of the coding states and generation of output bits resulting from the renormalization process can be implemented by single shared renormalization logic for both regular and bypass coding modes.

The original renormalization algorithm defined in the standard document [3] updates the coding states *codIRange* and *codILow* in an iterative fashion (Fig. 2.13) generating a single output bit at every iteration. The number of iterations is variable at each execution of the renormalization process and ranges from zero to maximum of 7 iterations. Because encode of a new bin, regular or bypass coded, can not be started till renormalization of the previous coded bin is finished, the latency of renormalization will be added to the latency of regular/bypass coding process to form the "bottleneck" of the encoding process.

The method described in section 3.3.3, divided the renormalization process into rescale of the coding states and bit generation by forming a *parsing area*. By decoupling these two parts, bit generation latency will be dropped from the bottleneck. It

will include only the regular/bpass coding latency plus latency of the rescale operation for the coding states. Rescale of *codIRange* can be simply carried out through a barrel shifter. Besides a barrel shifter, some simple extra logic is needed for rescale of *codILow*. These rescale operations are described in section 3.3.3. Because the bit generation operation involves more complex operations compared to rescale of the coding states, this decoupling significantly reduces the bottleneck size associated with encode of a bin. As a result, higher clock frequencies will be feasible.

4.1.2 Initial architecture

Our work presented in [23] describes a high performance architecture for CABAC encoding. The solution decoupled coding states rescale and bit generation of the renormalization process as described above. It also used a common renormalization interface for both regular and bypass coding methods (section 2.5.4) streamlining the flow of renormalization stage. The inputs for the *Low scaler* sub-block are multiplexed based on the coding mode of the encoded bin. Rest of the processing will be transparent regardless of the coding mode. Figure 4.1 shows the diagram of such architecture with memory blocks (either RAM or ROM) considered to be of synchronous type. The clock boundaries marked on the figure (from T_{-1} to T_7) are either enforced by memory access or "relative" complexity of the underlying combinational logic.

The bottleneck of arithmetic coding part now spans from T_0 to T_3 totalling 3 cycles while the previous discussion in section 3.2 referred to four cycles of T_0 to T_4 as the bottleneck (Fig. 3.1). The reason for this new argument is that $codILow$ is not needed in the earlier part of the T_0 cycle. It can be assumed that by the time $codILow$ is used for calculation of $codILow_{LPS}$ (the earliest use of $codILow$ within the pipeline), the registered $codILow$ will be available at the adder's inputs. On the other hand, $codIRange[7:6]$ is used right at the beginning of T_0. For this use, unregistered $codIRange$, called $codIRange_{scaled}$, is forwarded to $Range_{LPS}$

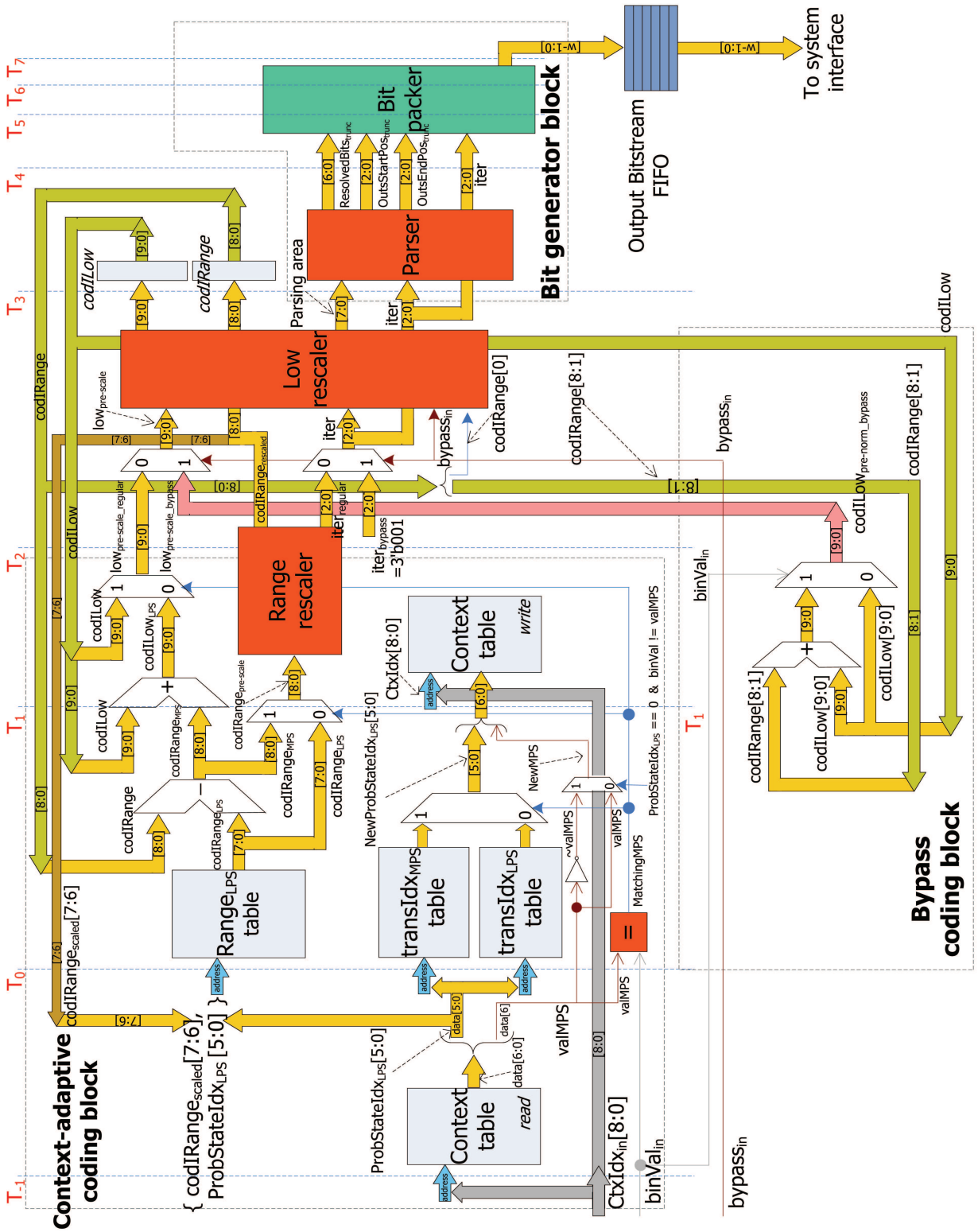


Figure 4.1: The complete architecture.

memory input. The synchronous memory will register the address lines at T_0 and the registered *codIRange* will not be needed for memory access. The other use of *codIRange* at deriving $codIRange_{MPS}$ will use the forwarded registered *codIRange* at the subtractor's inputs.

As bypass coding is simpler, its execution can be restricted to the beginning of the T_2 cycle. *Bit Generator* block is decoupled from the coding pipe as discussed before. Its first cycle is allocated for the *Parser*'s combinational logic. *Bit Packer* is more complicated and three cycles are allocated for it. As a result, it takes four cycles (i.e., four cycles latency) to finish the bit generation process corresponding to a new *parsing area*. The actual bit generation is finished at T_6 clock when the intermediate buffer is registered with new bits. The T_6 stage simply multiplexes out the low word of the intermediate buffer or one of the two resolved outstanding words (section 3.4.2). In other words, it can be assumed that the bit generation bottleneck spans three cycles (from T_3 to T_6) if implemented in all combinational logic. Since the coding block takes three cycles too, there is no need to make bit generation pipelined as the whole architecture achieves a three cycle throughput.[1] In spite of this, *Bit Generator* block is implemented in a pipelined form to facilitate further improvement to the architecture as described in the following section.

4.1.3 Accelerated dispatch of bypass coded bins

Further improvements to the architecture of the previous section can be explored (our subsequent work in [24]). Bypass coding is a much simpler operation than context-adaptive coding as it does not require access to the probability model. By tweaking bypass coding, section 3.5 simplified it by employing a generic rescaler for *codILow*

[1]Note that the multiplier table, *TransIdx* table and the *Parser ROM* register their inputs. Also, the final generated bits are registered to the intermediate buffer at T_6 clock. This forms four cascaded pipes (context table lookup at T_{-1}, coding from T_0 to T_3, bit generation from T_3 to T_6 and FIFO output selection from $T_6$to T_7).

Table 4.1: Statistics for bypass coded binary symbols

Sequence	Size and number of frames	Percentage of bins coded in bypass mode (%)	Percentage of bypass coded bins followed by a context coded bin (%)
Foreman	CIF 300	12.6	77.0
Mobile	SD 90	14.6	77.4
Coastguard	CIF 300	13.8	76.1
Football	SD 90	13.8	81.2
Susie (grey)	SD 150	8.9	77.5
Average	-	12.74	77.8

state[2] and bit generation blocks for both coding paths.

By proper rearrangement of its components, bypass coding can finish within a single cycle rather than the original three cycles similar to regular coding. As a result, the bin issue logic can be improved to incorporate completion time for different bins based on the encode mode. The next bin can be issued right after completion of the previous bin's encode rather than accommodating the larger delay of 3 cycles as the approach of section 4.1.2. For example, a completion time of three cycles for a context-adaptive coded bin and of one for bypass coded bin can be implemented in a dispatcher block issuing the bins at the right time.

By modifying H.264 reference software [12], the frequency of binary symbols arithmetically encoded in bypass mode was studied with encoding several standard test sequences. Table 4.1 shows the result. As the second column shows, almost 13% of the total coded bins are bypass coded. Since bypass coding can be issued every cycle, an improvement of two cycles per each bypass-coded bin can be made compared to issuing all bins at fixed three cycles intervals. Considering the frequency of bypass-coded bins, this results in an average throughput of $0.87 * 3 + 0.13 * 1 = 2.74$ cycles per bin which is equivalent to 9.5% improvement over the original three-cycle

[2] *codIRange* is not updated or rescaled in bypass coding process.

throughput. Figure 4.2 shows an implementation of such architecture.[3]

The figure starts with the bin FIFO sitting between the *Binarizer* block and the arithmetic coder. The bin entry read from this FIFO is registered in T_{-2} cycle. Right after completion of encode of the previous bin at T_3, the dispatcher logic sends the bin to the proper pipe (regular or bypass) based on its bin type. Note that here it is implied that encode of a bin finishes right before its bit generation phase which is not literally correct as the total latency spans till T_7 cycle for completion of a bin encode. Because update of the coding states finishes by T_3, "completion of encode" is used in this context.

There is no overlap in encode of successive bins (from T_0 to T_3) but the processing bottleneck varies depending on type of the bin: three cycles of T_0 to T_3 for regular-coded bin and one cycle of T_{2_bypass} to T_3 for bypass-coded bin. The dispatcher issues a regular-coded bin one cycle ahead of completion of encode of the previous bin. This happens concurrent to T_2 if the previous bin was regular-coded and concurrent to T_{2_bypass} if it was bypass-coded. This early issue of regular-coded bin at T_{-1} requires further registration of regular-coded bin and its parameters at T_{-1} as two successive regular-coded bins are processed in T_{-1} and T_2 concurrently though this concurrency does not happen at the bottleneck area.

A bypass-coded bin is directly sent to the T_{2_bypass} stage as soon as the previous bin enters T_3 because the new coding states are ready at that point. Note that the bit generation process still takes the original four cycles to complete but the pipelined implementation of *Bit generation* block (section 4.1.2) allows processing bypass-coded bins at every cycle.[4]

[3]To avoid complex presentation, some blocks are replaced by higher-level representation (like *NewMPS Calculator* and *Next probability state retrieval*) without drawing all feedback lines.

[4]This pipelined implementation was not required for section 4.1.2 but bit generation was implemented fully pipelined from scratch having this type of extension in mind.

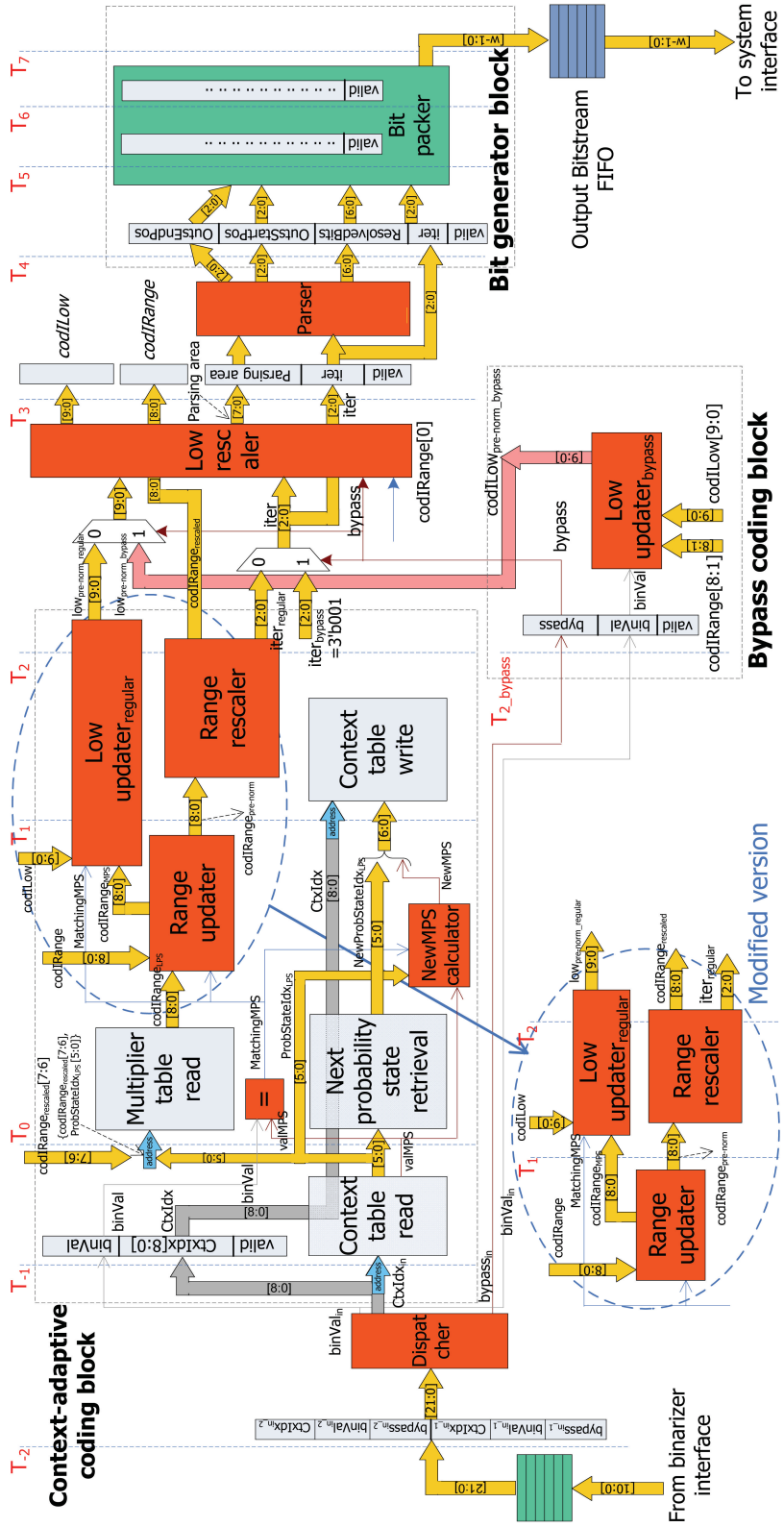


Figure 4.2: Using dispatcher for bin issue at non-fixed intervals.

Further improvement

Further improvement can be made by issuing two successive bins in a single cycle "if" the first encoded bin is bypass-coded and the second one is regular-coded. The bypass-coded bin is sent to T_{2_bypass} stage and at the same time, the following regular-coded bin should have reached T_0 stage implying the bin is issued to T_{-1} stage one cycle earlier.[5] This is possible since bypass coding does not change *codIRange* and only updates *codILow*. Because *codILow* is used towards the end of T_0 (Fig. 4.2) of the regular-coded bin, updated *codILow* of the bypass-coded bin can not be fed back to T_0 on time. By minor rearrangement of $Low\ updater_{regular}$ as shown in the modified architecture at the bottom of Fig. 4.2, now *codILow* is not needed before the T_1 stage. As a result, forwarding the updated *codILow* generated by the earlier bypass-coded bin to the T_1 stage of the regular-coded bin will be possible.

According to Table 4.1, 77.8% of the bypass coded bins in the examined test contents are followed by an arithmetic coded bin. This means that in average, 77.8% of bypass coded binary symbols will not consume any issue slot effectively as the regular-coded bin is processed at the same time. This results in an average throughput of $0.87*3+0.13*(0.778*0+0.222*1) = 2.64$ cycles per bin which is a 13.6% improvement over the original three cycles throughput or $2.74/2.64 = 3.8\%$ improvement over the first attempt.

Although the following regular-coded bin is actually issued a cycle earlier (to the T_{-1} preprocessing stage), the dispatcher still needs to be able to issue two bins concurrently. Assume the bin pattern of $\{b_n, b_{n+1}, b_{n+2}\}$ in the input FIFO where the first two are bypass bins and the last one is a regular bin. The second and third bins are supposed to reach the bypass path bottleneck (T_{2_bypass}) and the start of regular path bottleneck (T_0) at the same time implying b_n and b_{n+2} are issued at the same

[5] In fact, regular-coded bin has to be issued a cycle before the bypass-coded bin though the regular-coded bin is after the bypass-coded bin in the input stream.

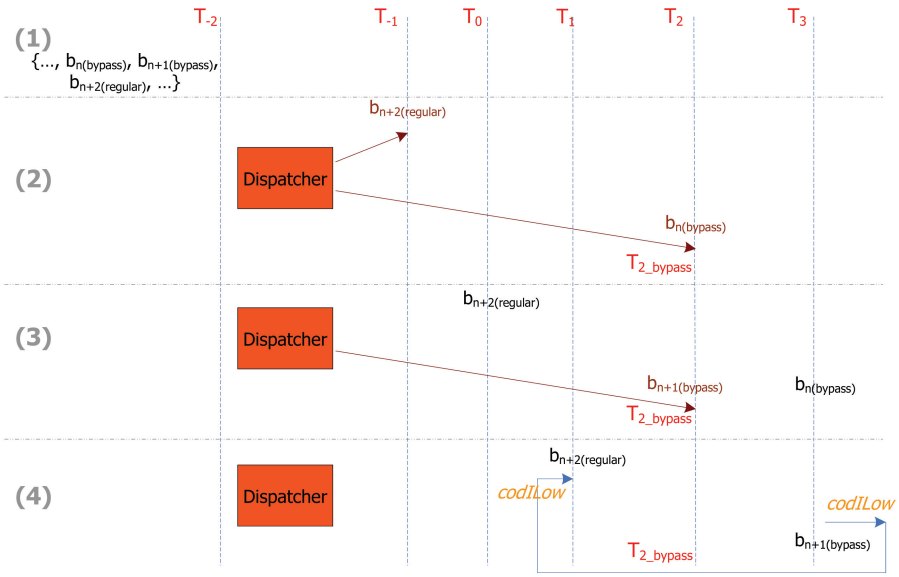


Figure 4.3: An example of issuing two bins at the same cycle.

time. Figure 4.3 shows this behavior better.

After the new modification, the bottleneck size corresponding to *codILow* is reduced to two cycles spanning from T_1 to T_3. One might suggest to increase the clock duration to fit *Range scaler* block[6] completely within the T_1 stage and register *codIRange* at T_2 instead of T_3. Then the bottleneck size associated with *codIRange* will also decrease to two cycles. As a result, the bottleneck for updating the coding states will be two cycles instead of three and could potentially lead to increased performance.[7] Instead of investigating such approach, the next section presents a fully-pipelined solution with a throughput rate of one cycle which supersedes this idea.

[6]Note that both *Low updater*$_{regular}$ and *Range scaler* blocks are combinational logics.

[7]The mentioned adjustments might require longer clock duration so there is a trade-off between decreasing the bottleneck size and the clock rate.

4.1.4 Fully pipelined arithmetic coding and bit generation

After breaking the renormalization and decoupling rescale of coding states from the bit generation, *codIRange* and *codILow* are each updated first and then rescaled as shown in Figures 4.1 and 4.2. Update and rescale of *codILow* depends on *codIRange*'s update and rescale. *codILow*'s new value is made available towards the end of the T_2 stage while new *codIRange* is almost available at start of T_2. This suggests that a fully pipelined implementation of arithmetic coding could be possible if all references to *codIRange* can be limited to one cycle while the following cycle is dedicated to use and update of *codILow*.[8] On the other hand, the read access to the multiplier ROM table at the first step of *codIRange* update would lengthen the latency of *codIRange* calculation in such approach because the ROM address depends on *codIRange*'s new value.

Using similar idea to the one of [10], the multiplier table content can be rearranged so it keeps all the four possible $Range_{LPS}$ values associated with $ProbStateIdx_{LPS}[5:0]$ in one entry of the multiplier table.[9] Then, a multiplexer selects one of the four supplied entries from the table based on $codIRange[7:6]$. By not using *codIRange* for addressing the multiplier table anymore, ROM latency will be reduced to multiplexer latency within *codIRange*'s bottleneck. This effectively dedicates T_0 for referencing the modified multiplier table.

Using the described technique, the architecture presented in Figure 4.1 is rearranged so all accesses to *codIRange* happens in the T_1 cycle.[10] Similarly, all references to *codILow* (reads and update) can be done in T_2. Now *codIRange* is registered with T_2 and *codILow* is registered with T_3. The rest of logic for context retrieval and

[8]Note that it will not be efficient to update both *codIRange* and *codILow* in the same cycle as dependency of *codILow* on *codIRange* would make the cycle too long.

[9]The ROM changes from 256 entries of 8-bits wide to 64 entries each with a width of 32-bits.

[10]Note that this change was also possible without using the modified multiplier table. But the latency of ROM access added to the latencies of update and rescale circuits for *codIRange* would make such a combined stage too long.

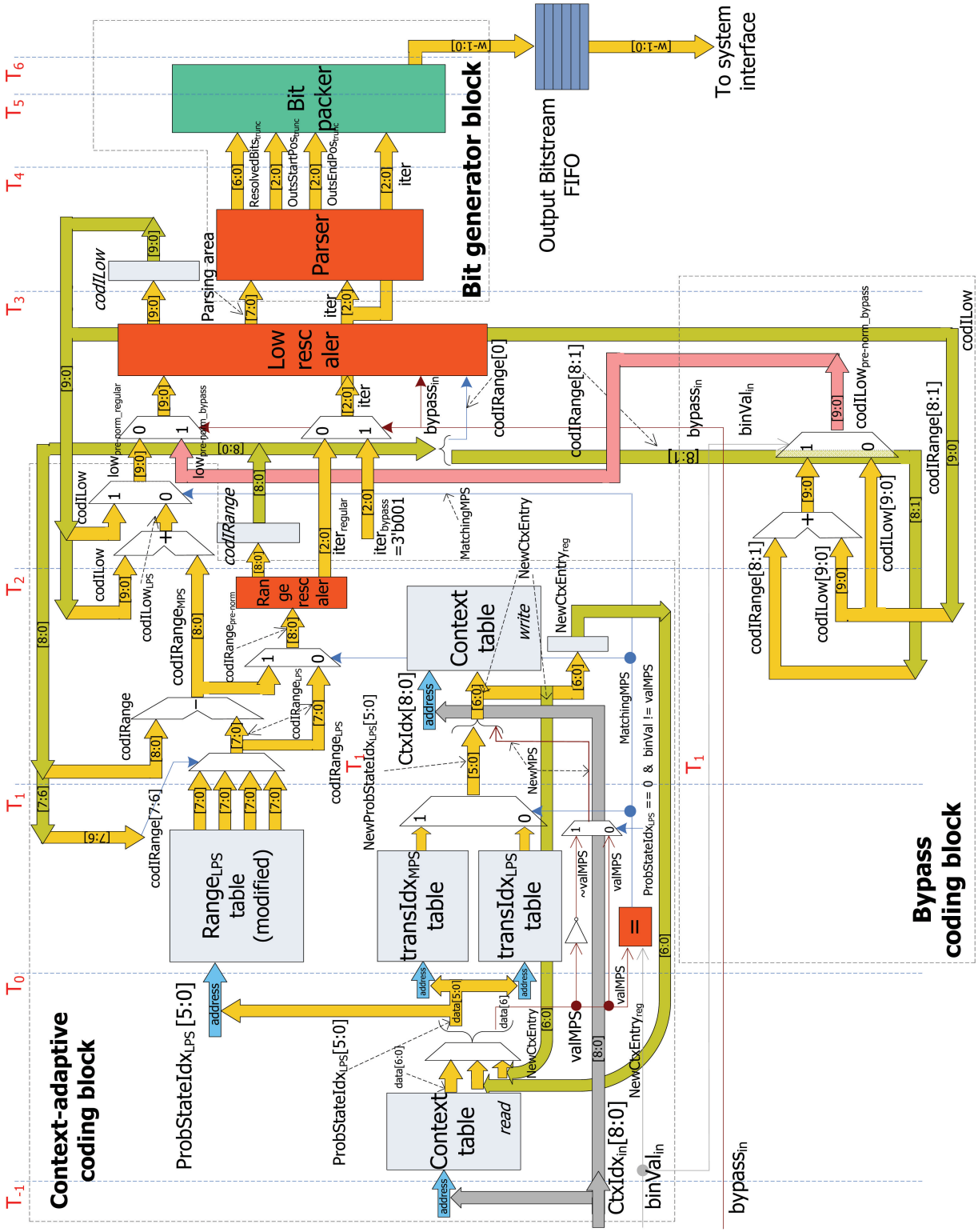


Figure 4.4: The enhanced fully pipelined architecture.

update remains the same as before. The circuit for bypass coding is already aligned properly at T_2 because it only depends on *codILow*. The bit generation logic has already been pipelined so it would not have problem handling a new *parsing area* at every cycle. Such fully pipelined architecture is shown in Figure 4.4.

Although this approach achieves a single-cycle throughput, it requires proper handling of some new issues as well. First, a single ported RAM for the context table was enough in the architectures of section 4.1.2 and 4.1.3 because there was no overlap between read and write accesses to the context RAM for two successive symbols. But in the new pipelined approach, all stages are active at every cycle so a dual-ported context RAM is required to carry out concurrent read and write accesses. The RAM is not required to be fully dual-ported as one port is dedicated for reads and the other port is for writes only. But both ports need to be independently addressable as they operate on different context indices associated to different bins.

Another issue arises when two successive[11] context-based coded bins are encoded using the same probability model, i.e., with equal context index. This is problematic because stage T_{-1} reads a context entry while stage T_1 has not received the updated entry yet or has not completed write of the context entry to the context memory.[12] The solution is to forward the updated probability model to be written or concurrently written to the context RAM within the last two cycles (as read and write are two cycles apart) to T_{-1} stage of the pipeline as shown in Figure 4.4. Now a multiplexer selects between the two forwarded data and the data coming from the context

[11] If $\{..., b_{n-2}, b_{n-1}, b_n\}$ are the last three bins issued for encoding, the discussed case is valid when b_{n-1} and b_n are regular-coded bins and both use the same context index. In this case, *NewCtxEntry* will be used. The other case is when all three bins of b_{n-2}, b_{n-1} and b_n are regular-coded and b_{n-2} and b_n have the same context index. In this case $NewCtxEntry_{reg}$ is selected from the multiplexer inputs since the updated entry is two cycles away and is still not visible when the context RAM read and write happen at the same cycle.

[12] This depends on the RAM behavior too as some dual-ported memories send out the data as it is written to the RAM when the same address is read through the other port. If this is the case for the RAM used in the final configuration, one of the forwarding paths will not be required anymore.

RAM (potentially carrying out-of-date). This is similar to bypassing or forwarding techniques used in standard processor design.

Since the architecture is fully pipelined and a new set of inputs are processed at every cycle, some signals need to be registered at the next cycle before losing their value due to new inputs. Also, extra registrations are necessary at every further cycle till such signals have reached their final destinations. Samples of such signals include $MatchingMPS$, $CtxIdx_{in}[8:0]$ and $binVal_{in}$. To avoid complexity, Figure 4.4 does not show these extra level of registration for signal delivery to the later stages and they are directly routed from their original stage.

An important note is that there is a trade-off between the degree of pipelining and the frequency the implementation can be clocked at. Obviously, more operations have to be squeezed within stage T_1 here so the cycle duration must be longer. But the implementation results in chapter 7 will show that the overall throughput will be higher despite the reduced clock frequency.

Bit generation block

The high-level operation and mechanics of bit generation was described in section 3.4.2. Figure 4.5 shows its core pipelined implementation used in the architectures presented in sections 4.1.2, 4.1.3 and 4.1.4. The design of bit generation of Figure 4.5 belongs to the fully-pipelined architecture of Figure 4.4 which its expected longer duration cycle allows removing one stage in *Bit Generator* block reducing it to three (spanning from T_3 to T_6) from four in the earlier designs of Figures 4.1 and 4.2 (spanning from T_3 to T_7). As a result, the *Bit generator* block has a latency of three cycles from T_3 to T_6 with the generated words of output stream being sent to the *Output bitstream FIFO* at T_6. It has a single-cycle throughput so it can keep up with the earlier fully pipelined arithmetic coding stage.

A parsing area is generated by the renormalization process performed at encode of

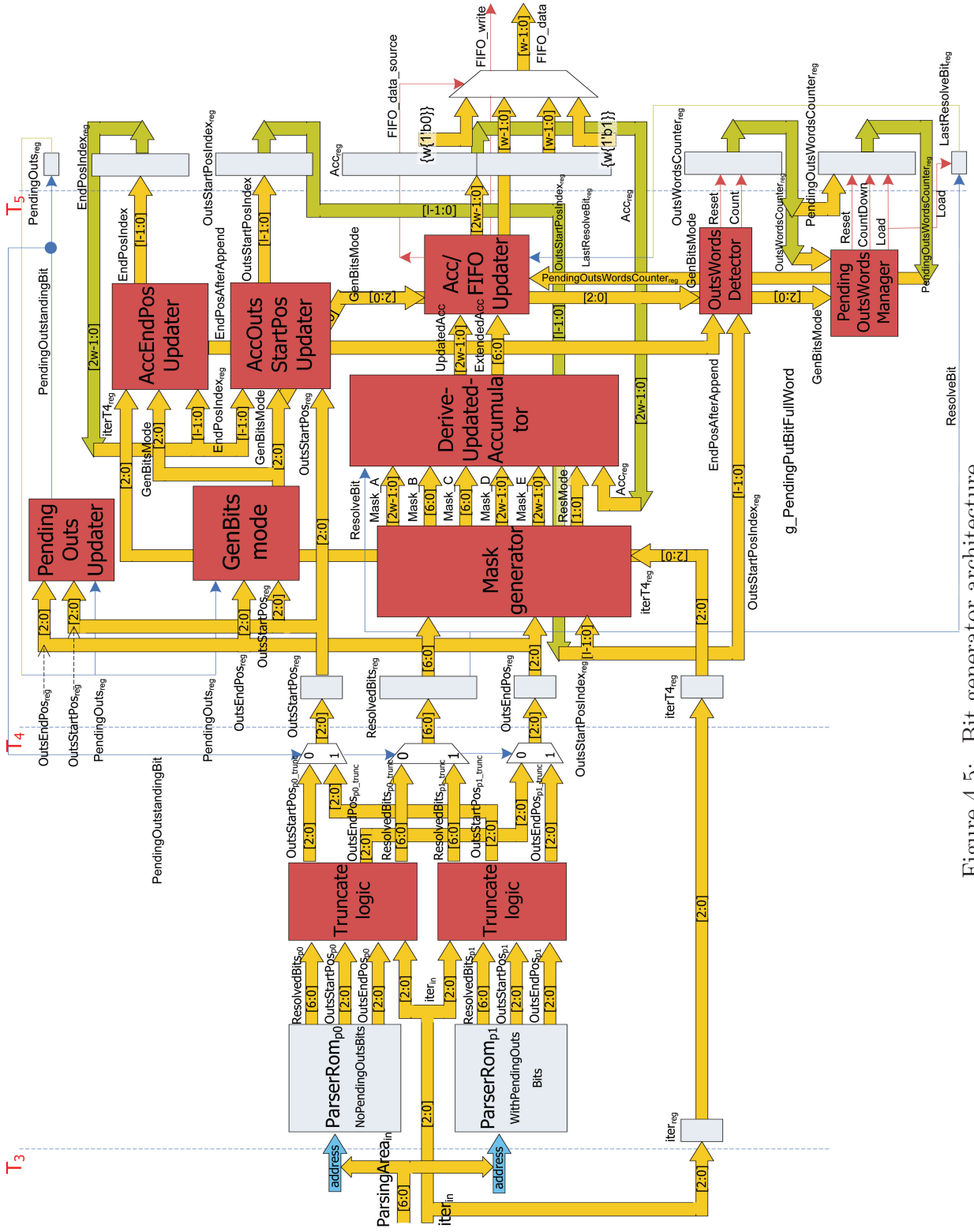


Figure 4.5: Bit generator architecture.

an input bin. The block receives the parsing area, $ParsingArea_{in}$, along with its size, $iter_{in}$. The size of the received parsing area can vary between zero to seven bits and each bit is associated with a generated bit. The unresolved *outstanding bits* within the parsing area need to wait for a future parsing area before being resolved. An accumulator register, Acc_{reg} also known as *intermediate buffer* in earlier discussions, with a size equal to double of the FIFO width, w, is used to buffer the generated bits. The outstanding bits within the accumulator will be resolved with arrival of a *resolve bit*. $OutsWordsCounter_{reg}$ tracks longer sequence of the outstanding bits in units of words to prevent the accumulator from being overflown by sequence of outstanding bits. $EndPosIndex_{reg}$ points to the last valid bit within the accumulator.

Similarly, $OutsStartPosIndex_{reg}$ points to the starting bit of the outstanding bits area within the accumulator. $PendingOutsWordsCounter_{reg}$ keeps a copy of $OutsWordsCounter_{reg}$ after arrival of a resolve bit to let $OutsWordsCounter_{reg}$ be used for a new set of possible incoming outstanding bits. $PendingOutsWordsCounter_{reg}$ will be decremented at every cycle that a FIFO write access is possible. As a result, a word of resolved outstanding bits which their polarity is derived from the polarity of the last resolved bit, $LastResolveBit_{reg}$, can be written to the output FIFO. $PendingOuts_{reg}$ tracks existence of an outstanding bit within the accumulator.

Because the block is fully pipelined, registers at T_3 and T_4 will register the intermediate results or pass the inputs to the later stages. The parser ROMs are assumed to be of synchronous type so they register $ParsingArea_{in}$ without the need for extra registers. At T_3, only $iter_{reg}$ is needed for such purpose. At the T_4 stage, $OutsStartPos_{reg}$, $OutsEndPos_{reg}$, $ResolvedBits_{reg}$ and $iterT4_{reg}$ will register the intermediate results which will be used as inputs for the next stage.

The function of each high level block shown in Figure 4.5 is described as follows.

$ParserRom_{p0}$, $ParserRom_{p1}$: As explained in section 3.4.2, the parser table consists of 512 elements. The status of a pending outstanding bit existing within the

accumulator is used as a parsing condition, i.e., one of the addressing bits. In the fully pipelined implementation of the bit generation block, this bit will be generated late in the T_4 cycle which is too late to be used as an address bit at the T_3 edge. As a result, both entries corresponding to the rest of 8 address lines are read and processed and very late in T_3, the right one is selected based on the new status of outstanding bit in the accumulator, $PendingOuts$. As a result, the parser ROM is broken into two halves one corresponding to the 256 entries where no outstanding bits exist in the accumulator and the other half associated with the case one exists. Effectively this suggests a single 256 ROM where each entry has a double width of $2 * 13 = 26$ bits.

***Truncate logic*:** The parser ROM stores the partially resolved bits (7 bits) and each of the start and end indices (each of 3 bits) for potential outstanding bits area as there are 7 bits generated for the input parsing area. This behavior is because it will be too costly to have ROMs for all cases of $iter_{in} \in [1, 7]$. The alternative is to use only parser ROM associated with the parsing area size of 7 and then truncate the resolved bits, start and end indices as $iter_{in} < 7$.

***GenBits mode*:** Using the start and end positions of the potential outstanding area within the last parsed area ($OutsStartPos_{reg}$ and $OutsEndPos_{reg}$) and whether the accumulator currently contains any outstanding bit ($PendingOuts_{reg}$), this block infers the current mode and scenario associated with the generated bits as indicated in Figure 3.6-b.

***Mask generator*:** This block creates several masks used for updating the accumulator with current resolved bits. Temporarily, the accumulator is expanded by 7 bits (maximum size of generated bits) to accommodate append of new generated bits because it might be possible to send the first word of the accumulator to the output FIFO at the same time of append operation. Then enough room will

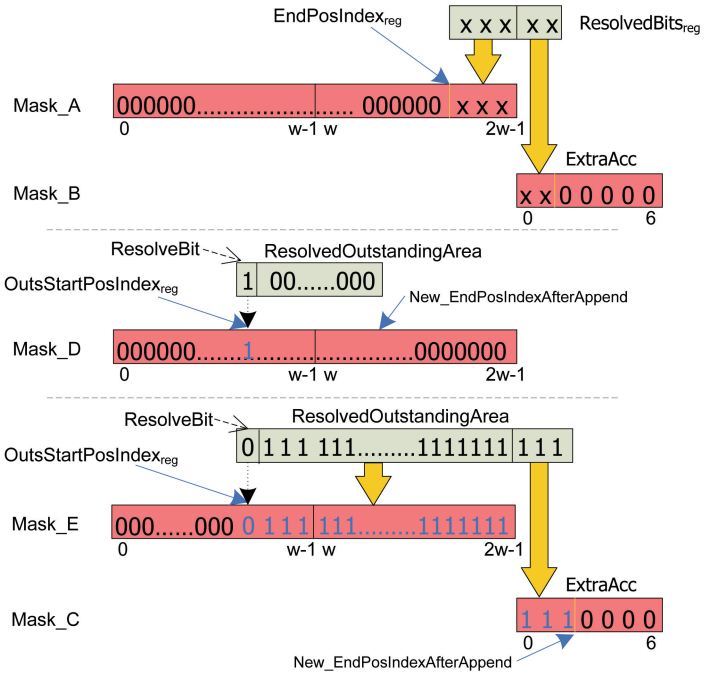


Figure 4.6: Different masks created by the *Mask generator* block for appending the generated bits and/or incorporating the resolved outstanding bits.

be still available to accommodate the new bits into the original accumulator. These different masks are depicted in Figure 4.6 and described below.

Mask A: This is a pseudo-mask with the same size as accumulator, $2w$, to facilitate update of the accumulator with the portion of $ResolvedBits_{reg}$ that is to be appended to the accumulator.

Mask B: This 7-bit area keeps the portion of $ResolvedBits_{reg}$ which could temporarily overflow from the accumulator.

Mask C: This 7-bit area keeps the tail portion of the resolved outstanding bits for a resolved bit of 0 (implying the tail portion is all 1's) which could temporarily overflow from the accumulator.

Mask D: This is a pseudo-mask with a size equal to the accumulator for incorporating the resolved outstanding bits when the resolve bit is 1. As a result, only the heading bit of 1 needs to be applied to the mask and the tailing 0's can be ignored as the mask originally carries 0. See sections 3.3.3 and 2.5.3 for more details about mechanics of resolving outstanding bits.

Mask E: This is a pseudo-mask with a size equal to the accumulator to incorporate the resolved outstanding bits when the resolve bit is 0. As a result, the tailing bits (of all 1) need to be applied to the mask and the heading bit of 0 can be ignored as the mask was originally zero.

ResMode specifies what resolve mode should be used by *DeriveUpdatedAccumulator* block. It basically indicates what combination of generated masks should be used by that block.

***DeriveUpdatedAccumulator*:** This block uses the masks generated by the *Mask generator* to append the new generated bit and/or incorporate the resolved outstanding bits based on the polarity of *ResolveBit*. This process creates the updated accumulator, *UpdatedAcc*, and possible overflow bits from it, *ExtendedAcc*.

***Acc/FIFO Updater*:** This block uses the updated accumulator *UpdatedAcc* and possible overflow bits *ExtendedAcc* generated by *DeriveUpdatedAccumulator* block. It also takes into account available FIFO access slot in case the low word of *UpdatedAcc* is fully resolved and ready to be sent out to the output stream. The block updates the accumulator register, Acc_{reg} and generates possible FIFO output. If there are any extra bits generated in *ExtendedAcc*, they are shifted into the accumulator register since the lower word of the accumulator can be dumped to the output FIFO for sure. This is guaranteed as the *Accumulator Overflow Detector* block (not shown in the figure) would detect

such condition. Then it stalls the bit generator and the earlier arithmetic coding pipelines till the contention at access to the output FIFO is resolved and *ExtendedAcc* can be fully consumed by Acc_{reg}.

***PendingOutsUpdater*:** Simply updates the status of whether any pending outstanding bit remains in the accumulator and saves it in $PendingOuts_{reg}$.

***AccEndPosUpdater*:** This block comes up with the new index pointing to the last valid bit inside the accumulator register considering all factors like append of the generated bits, resolve of outstanding bits, and dump of the low word to the output FIFO. The new index is saved in $EndPosIndex_{reg}$ register.

***AccOutsStartPosUpdater*:** This block comes up with the new index pointing to the start of the outstanding bits area pending in the accumulator considering factors like append of the new generated bits, and dump of the low word to the output FIFO. The new index is saved in $OutsStartPosIndex_{reg}$ register.

***OutsWordsDetector*:** This logic checks the pending outstanding area and tries to reduce the portion that fully overlaps the second-word of the accumulator (indices w to $2w-1$). It counts the number of such words and adjusts the end position of the outstanding area to accommodate more room within the accumulator for future generated bits. The counter value is kept in $OutsWordsCounter_{reg}$ register.

***PendingOutsWordsManager*:** This logic manages $PendingOutsWordsCounter_{reg}$ register which receives the value of $OutsWordsCounter_{reg}$ when the existing outstanding area is resolved. It lets $OutsWordsCounter_{reg}$ track the new incoming outstanding area instead of being stuck with the old outstanding area. $PendingOutsWordsCounter_{reg}$ is later decremented at cycles that access to

output FIFO is possible and a full resolved word of form $\{w\{1'b0\}\}$ or $\{w\{1'b1\}\}$ corresponding to a resolve bit of 1 and 0 respectively is sent to the FIFO.

4.2 Efficient handling of outstanding bits

A major architectural challenge for designing a high performance CABAC encoder is that of the ability to handle the outstanding bits. Proper handling of outstanding bits is an important issue both in renormalization and bit generation blocks. Outstanding bits is a pattern of bits (a single one/zero bit followed by *count* number of bits of opposite polarity) where *count* is incremented whenever the middle branch of renormalization is taken (Fig. 2.13). The pattern will be known at the *resolve time* when a non-outstanding zero or one bit, the *resolve bit*, is generated. The arrival of a *resolve bit* resets the *count* value after resolving the pending outstanding bits. The standard document [3] does not enforce a maximum bound on the size of outstanding bits and only suggests a counter size to be able to track the whole slice size. This becomes problematic in a hardware implementation as discussed below. Refer to sections 2.5.3 and 3.3.3 for review of the issue.

The problem arises since the polarity of the *resolve bit* is not known till its arrival which can happen within the current renormalization process or in later renormalizations corresponding to encode of future *bin*s. This suggests that an *intermediate buffer*, also called *accumulator* in section 4.1.4, is necessary for accumulating the generated bits before sending them to the output FIFO. Section 3.3.3 suggested forming and processing a *parsing area* to decouple rescale of the coding states from bit generation. Then proper "parsing rules" were introduced to resolve the outstanding bits that can be resolved internally. The different scenarios are possible for the *parsed area* which each needs its own handling where potentially a mix of raw bits and resolved outstanding bits are appended to the accumulator. Whenever enough resolved bits

Table 4.2: Size and probability of the longest sequence of outstanding bits.

Sequence	Size & number of frames	Longest sequence of outstanding bits	Probability of occurrence of the longest sequence
Foreman	CIF 300	35	$8.01 * 10^{-9}$
Mobile	SD 90	42	$5.79 * 10^{-9}$
Coastguard	CIF 300	41	$4.63 * 10^{-9}$
Football	SD 90	38	$4.37 * 10^{-9}$
Susie (grey)	SD 150	36	$1.57 * 10^{-9}$

are available in the accumulator, they are removed from the buffer and transferred to the output FIFO. Size of the intermediate buffer is intentionally made double of the output FIFO width to handle possible overflow scenarios because of limited bandwidth of access to the output FIFO.

In a hypothetical worst case scenario where encode of successive symbols all generate maximum possible bits of seven, an accumulator size of 64 (implying word size of 32 bits) can handle a maximum of 96 outstanding bits. This maximum reduces to 32 bits when the accumulator is set to 32 bits (word size is 16 bits).[13] To prevent overflow of the accumulator, a solution adds a stall logic in the design to stall the arithmetic encoding engine and stop it from processing new bins and generating further bits when such overflow scenario becomes imminent. The stall continues till enough bits become available in the accumulator to accommodate next chunk of generated bits.[14]

Since there is a trade-off between increasing the accumulator size and the stall frequency, it is necessary to come up with a reasonable target for maximum length of outstanding bits expected to be handled in a stall-free fashion. The maximum length of outstanding bits sequences and the rate of occurrence of such sequences for a few test contents are demonstrated in Table 4.2. This data is collected through modifying

[13]Section B.4 discusses how these limits are derived.

[14]Note that each stall cycle will release a full word of outstanding bits (e.g. 32 bits for a word size of 32) so the stall duration is not expected to continue for long.

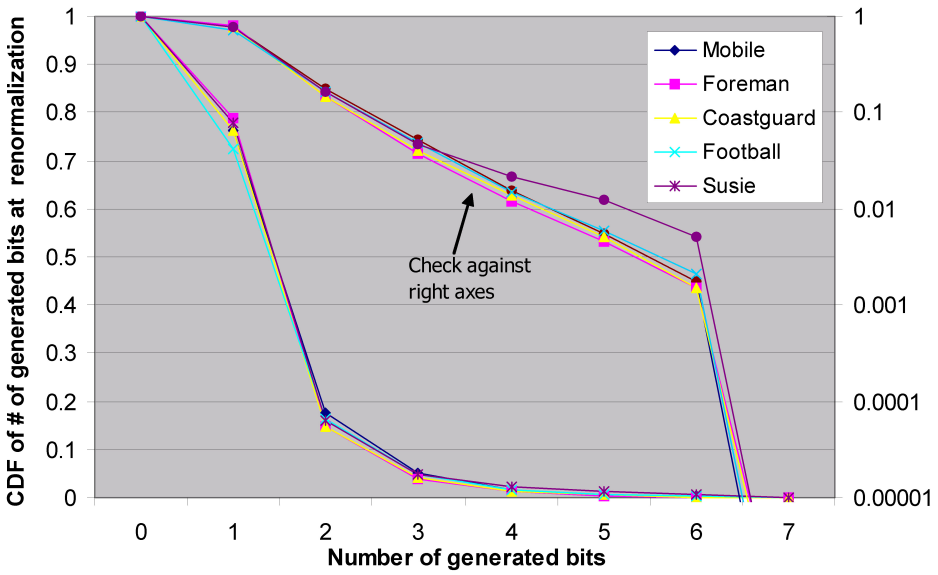


Figure 4.7: CDF of number of bits generated at renormalization.

the reference software [12] and encoding a few standard test contents. Figures 4.7 and 4.8 respectively show the empirical results for size of the generated bits at the renormalization process and length of the outstanding bits at resolve time.

Figure 4.7 reflects a cumulative distribution function (CDF) for lengths of outstanding bit sequences from zero to the maximum possible value for different video contents. Since the frequency of longer sequences goes down sharply, the graphs are also shown in logarithmic scale. As the curves show, the probability of encountering an outstanding sequence of length 12 or more is less than 0.1% while it is less than 0.001% for sequence of 25 or longer. This data suggests that stall-free support for outstanding sequences up to 96 bits provided by a 64-bit intermediate buffer is possibly an overdesign. Figure 4.8 shows the cumulative distribution of number of bits generated within a single renormalization. The exponential decrease of the graph means that in each renormalization step, probability of generating high number of bits is much lower compared to shorter bits. For all sequences except *Susie*, less than

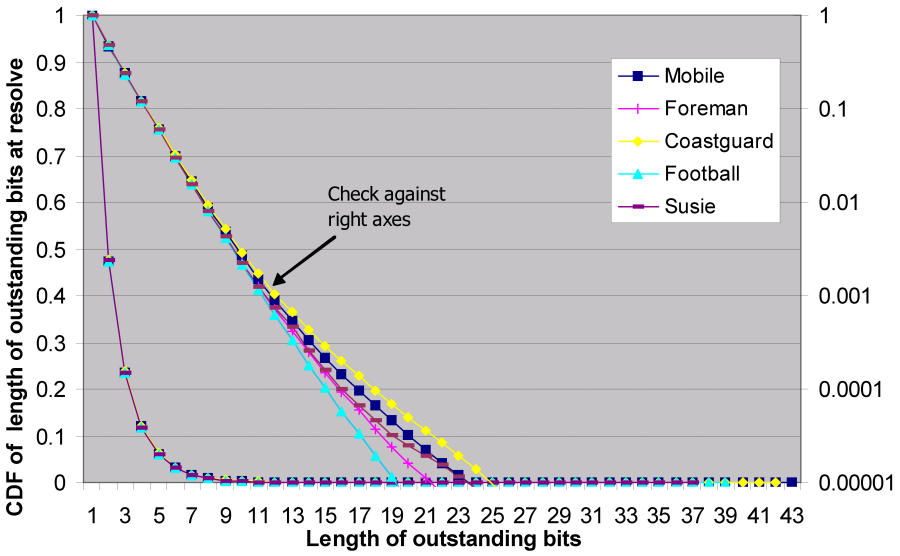


Figure 4.8: CDF of length of outstanding bits at resolve time.

one percent of renormalizations generate five or more bits. This suggests the worst-case calculation for handling stall-free outstanding sequences can relax its assumption based on statistical data. For example, an intermediate buffer size of 32 can handle *mobile* content that has a longest sequence of size 42 without stall. Based on the worst case calculation, stall-free operation for outstanding sequences longer than 32 can not be guaranteed with an intermediate buffer size of 32. Experimental results comparing hardware complexity of different buffer sizes are provided in chapter 7.[15]

4.3 Other issues

Here, some other architectural issues not covered in the previous sections of this chapter are discussed.

[15]Note that the buffer size affects other circuits like size of pointer registers, append logic and generated masks besides the intermediate buffer size itself.

4.3.1 Context table initialization

As mentioned in section 2.5.1, the (u_γ, v_γ) pair and the quantization factor $SliceQP$ are used to derive the initial value of probability state p_{LPS} and MPS for each context entry with index γ. Considering $SliceQP$ can be any integer within [0, 51] and the fact that some context entries have up to four different pair of (u_γ, v_γ)[16], storing all different combinations of initialization tables in ROM will not be possible. The context table need to be restored with its proper initialization value before start of encode of every new slice. Precalculation of the table for the next slice while encoding the current slice is not possible either because $SliceQP$ and the model number are part of the slice header and will not be known till encode of the next slice is started.

To reduce the potential 399 cycles required for initialization of the table, the following approaches can be useful. Since not all of the 399 contexts are used for encode of each slice, the initialization mechanism should calculate only the contexts needed for the current slice considering the slice type and MBAFF mode (see section 2.4.2). Also, if the context RAM data port is wider than a single context (e.g., of 4 words), multiple contexts can be calculated in parallel and written to the RAM at the same time. The extra complexity of such approaches might not gain much after all as the typical number of slices for HD contents is around 30 for frame slices and 60 for field slices suggesting that context table initialization does not happen very often.

[16]This four possibilities depend on the slice type, e.g., I/SI, P/SP and B, and the *model number* parameter used for the current slice. The model number is used for slice dependent adaptation discussed in section 2.5.1 and is stored as a syntax element in the slice header. It is named as *cabac_init_idc* in the context of standard document [3] and specifies the index for determining the initialization table used in the initialization process for context variables. This value can change within the range of 0 to 2, inclusive. Note that in the standard document literature, (m, n) pair is used instead of (u_γ, v_γ). For example initialization pairs of (m, n), refer to Table 9-18 of [3].

4.3.2 First generated bit at every slice

The very first bit generated in every slice using the CABAC engine must be ignored according to the standard document [3]. This means that the encoder does not add this first bit to the output bitstream and the decoder implicitly takes this into account when starting decoding the received bitstream. To do so, the decoder reads only the first 9 bits of the bitstream to initialize its *codIOffset* coding state instead of 10 bits (see section 9.3.1.2 of the standard document [3]).

Ignore of the first generated bit by CABAC is because this bit always has the known polarity of zero. This fact is dictated by the initial values of arithmetic coding states $codILow = 0$ and $codIRange = 510$ and the proof is given in the appendix at section B.3. It is important that the *Bit Generator* block of CABAC architecture be capable of detecting and then dropping the first generated bit within the slice. Otherwise, the host processor receiving the bitstream through the FIFO would need to shift the whole bitstream of the encoded slice to take this into account. Obviously, such shift will be a time consuming operation for the host processor.

4.3.3 Maximum length of outstanding bits

In this design, it was assumed that the longest sequence of outstanding bits can not go over 1023 bits so up to a maximum of 32 outstanding words (assuming output FIFO word of 32 bits) can be tracked through a 5-bit $OutsWordsCounter_{reg}$ register (1024/32 bits FIFO width = 32 words). Based on our empirical results, the longest observed sequence of outstanding bits was 42 (section 4.2). Of course, it is expected to observe longer sequences using a wider range of contents encoded with different set of encoding parameters. Because cost of supporting a wider $OutsWordsCounter_{reg}$ is negligible, a safe approach can assume a maximum length equal to the maximum slice size as suggested by the standard document [3] and discussed in section 2.5.1.

The maximum slice size depends on the target profile level. Since each outstanding bit will be resolved to an output bit after its resolve, the maximum sequence of outstanding bits can not be longer than the maximum size of "encoded" slice. The encoded slice is smaller than the input slice because of the compression achieved by the entropy coder. Furthermore, the per second maximum bitrate limits defined by Annex A of the standard document [3] for each profile level can be used to derive the maximum size of the encoded slice.[17] As a result, the standard document should have pointed to the maximum size of encoded slice as the longest possible sequence of outstanding bits instead of maximum size of input slice fed to the arithmetic coder.

Since such system level constraints were not available, the maximum slice size could not be derived and a maximum length of 1023 seemed a conservative enough choice for the type of test contents used in this work.[18] After all, it is very easy to update the implementation after knowing the system-level constraints and the final bitrate targets. Such change will affect only the size of $OutsWordsCounter_{reg}$ and $PendingOutsWordsCounter_{reg}$ registers and their associated count up/down logic.

4.3.4 Reduction of Parser ROM size

The parser ROM can be reduced to 256 entries from the original 512 entries presented in section 3.4.2. This can be done by removing "status of pending outstanding bits in accumulator", *PendingOuts*, as a bit of the parsing area. Its effect can be generated on the fly by some extra logic applied to the output of parser ROM. This idea was later dropped after it was realized that PLA (Programmable Logic Array) minimization tools like *Berkeley*'s *Espresso* does a wonderful job in optimizing ROM blocks. The ASIC design presented in chapter 7 will discuss the use of this tool further.

[17] Some of the factors affecting the maximum size of encoded slice include the highest target profile level, the minimum slice rate per second, size and combination of group of picture (GOP), etc.

[18] The other option was to assume the very unrealistic case of one slice per second which translates to a maximum slice size of 50 mega bits at Level 4.2. Then $24 - 5 = 19$ bits will be required.

Chapter 5

Binarizer implementation

The binarization process is a pre-processing step employed in CABAC to allow more efficient operation of the subsequent context modeling stage through assigning a unique intermediate binary codeword, *bin string* or *bin*s, to each non-binary syntax element (section 2.4.1). Figure 2.4 shows the stage this process happens within the CABAC. The binarizer associates a context index to each generated bin which is to be encoded in regular mode. This context index points to an entry in the context table where the statistical model used for regular encode of the related bin is stored. As a result, the encoding method (regular or bypass) needs to be signalled for each bin. The interfaces of binarizer within CABAC architecture is shown in Figure 5.1.

The high level binarizer model assumed so far in this work is based on the model presented in [5] where the binarizer process is hardware accelerated rather than fully hardware implemented. In this model, the hardware accelerator is a thin layer implementing the four core binarization schemes of *Unary coder*, *Truncated unary coder*, *Exponential Golomb coder* and *Fixed-length coder* (Figure 5 of [5]). The upstream block (e.g., a host processor) controls the actual binarization schemes required for each syntax element and sends stream of *binarization commands* to the block.

Such interface can not suit all system architectures as the upstream block can

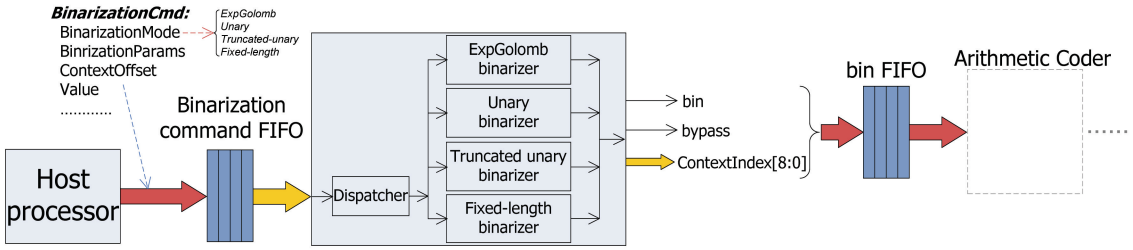


Figure 5.1: Binarizer stage as the first stage of CABAC block and its input/output interfaces.

have different level of processing power based on the overall design. The next section argues for an improved binarizer interface. The later two sections suggest methods to improve the performance of the base accelerator by adding higher degree of hardware support. The difficulties that context index calculation imposes on binarizer will be discussed too. The last section of this chapter evaluates the tradeoffs and compares the pros and cons of each approach.

The system architecture and capabilities of the upstream block interfacing to the binarizer will affect the final decision on choosing the suitable architecture. For the rest of this discussion, *binarizer interface* term is specifically used for its input interface to the upstream block which actually forms the CABAC input interface too.

5.1 Arguing for a better binarizer interface

The thin-layer binarizer model assumed so far was based on the work of Sudharsanan, et al. which advised a model with an upper-level host processor creating one or several binarization commands and sending them to the binarization accelerator [5]. Later, the controller dispatches each command to the right encoder. As discussed earlier, each generated bin by the binarizer needs to carry a proper context index, *ContextIndex*, if the bin is supposed to be encoded using regular arithmetic coding.

The main contribution of [5] was to come up with the idea of a ROM lookup mapping scheme to generate the right sequence of context indices needed for encode of the bin string resulting from each binarization command. In their work, each binarization command carries two fields of *CxtData* and *CxtOffset*. These fields and the bin index of each generated bin by the binarizer are used to calculate the context index through a ROM lookup followed by an addition operation (Figure 6 of [5]). The ROM is prepared in a form that resembles Table 9-24 of [3] with all possible combinations of entries specified through different subclauses are allocated with their own memory lines in the mapping ROM. This increases the size of the ROM beyond what seen in the table as some rows carry several possibilities through different subclauses where all possible combinations need to be allocated their own rows.[1]

This approach streamlines the binarization logic as it simply processes core binarization requests without being concerned about the syntax element type. But the problem is that it actually off-loads a portion of context index calculation to the host processor as the software/firmware needs to come up with its own mechanism for calculating *CxtData* based on the conditions of each subclause. It reflects some redundancy between the software and the mapping operation through ROM because they could have been combined together. This suggests that more of the binarization procedure can be brought into the hardware side (departing from the idea of thin hardware for accelerated binarization) without adding too much "overall" complexity.

Also, many implementations highly desire a self-contained CABAC with full hardware binarizer as either they do not have a dominant host processor to run the software portion of binarization, or the processor can not spare extra cycles for such task. Of course, extra hardware for binarizer would require longer verification phase. On the other hand, the next section will show that the improved binarizer can be designed

[1]Note that Table 9-24 of [3] includes the assignment of *ctxIdxInc* based on *ctxIdxOffset* and *binIdx* for a subset of syntax elements. The process for other syntax elements like *coded_block_flag* is explained in other tables as they have other dependencies like context category, *ctxBlockCat*.

using simple modules which can be verified separately. At the end, the next two designs will be more complex than the thin layer model. In return, they take some load off the upstream logic with their extra hardware support. Section 5.4 at the end of this chapter will evaluate the tradeoffs between different degree of hardware support in the presented designs, the incurred design complexity, memory usage and the overall effect on the system architecture.

5.2 Syntax element binarization in hardware

To eliminate the redundancy in calculation of the context index between the higher level syntax element binarization and the hardware accelerator, these two stages can be combined for majority of the syntax elements where derivation of context index increment, *ctxIdxInc*, for different subclauses of Table 9-29 of [3] can be done with simple condition tests. For more complicated *ctxIdxInc* derivation (like cases of *mb_type* and *sub_mb_type* syntax elements), a ROM lookup still is a good choice. A high-level architecture of this idea is shown in Figure 5.2 where no longer it is assumed that a higher-level processor runs the binarization process for syntax elements.[2]

Now individual blocks manage the high-level binarization for each syntax element and feed the core binarization blocks. The core binarization schemes are reduced to a *ExpGolomb* coder and a combined *Unary/Truncated-unary coder*. A *Fixed-length coder* block is really not necessary as almost all uses of this coder is limited to the simple case of single-bit coder as indicated by Table 9-24 of [3]. The only exceptions are a 3-bit fixed-length code (*cMax=7*) used for *rem_intra4x4_pred_mode* syntax element and a 4-bit fixed-length code (*cMax=15*) for the prefix portion of *coded_block_pattern*

[2] *Hi-level H-264 coder* block here reflects the hardware/software combination executing all portions of H.264 algorithms (Fig. 2.2) up to the *Entropy encoding* level. However, this layer is still responsible for keeping the history of syntax elements belonging to the previous neighboring macroblocks (*MB state tracker* block). This history is used to calculate proper *context index hints* required for binarization of certain syntax elements (*Context hint calculator* block).

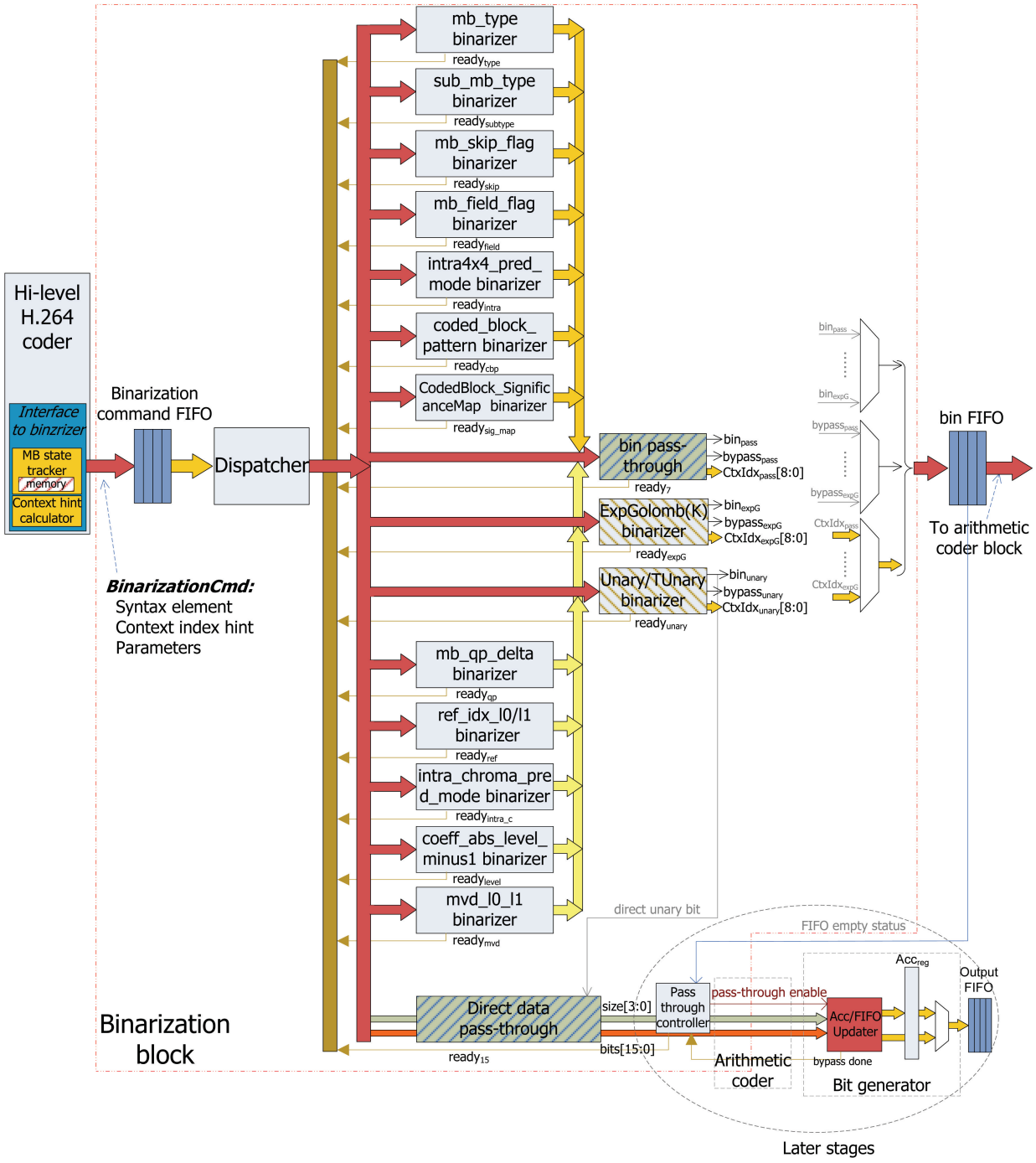


Figure 5.2: The suggested binarizer architecture for improved performance.

syntax element which are very simple to implement inside the corresponding binarizers rather than including another core block. Depending on the syntax element type, the high-level binarizers drive the two core binarizers or simply generate the right bin and use the *bin pass-through* path to the output bin FIFO directly. The syntax element binarizers wait till the core binarizers have finished their process and then indicated their readiness for accepting a new syntax element (through their individual *ready* signal) after the whole binarization process is finished.[3]

With this method, each syntax element binarizer block will manage generation of both context indices and bins at the same time. This new method reduces the required binarization interface bandwidth significantly as the previous "core" binarization commands are now replaced with high-level "syntax element" binarization commands. For the case of motion-vector difference (MVD) as an example, now a single high-level command is sent to the binarizer while between one to three core commands were required in the previous approach.[4] Table 5.1 shows the statistics for binarization of MVD and transform coefficient level indicating the reduction in number of binarization commands. Less number of binarization commands will require smaller depth FIFO for buffering the commands.

Just to get a sense of the rate of syntax elements issued to the binarizer, let us add up the total number of MVDs and transform coefficients for the most complex content which is *Mobile*. The resultant 2,628,051 issued syntax elements over 90 frames ($\sim$3 seconds) is equivalent to a rate of 876,017 syntax elements per second. Considering that MVDs and transform coefficients consist the large majority of the generated syntax elements, the final rate of binarization commands will be in the low orders of

[3]Note that binarization of some syntax elements might drive the core binarizers several times. An example is motion vector difference which drives *Unary/Truncated-unary coder* first and then potentially drives *ExpGolomb coder*.

[4]The first command was for truncated-unary code of the difference value which was potentially followed by an Exp-Golomb command depending on the value of MVD. For non-zero MVD values, a third command (pass-through bin command) was also necessary to encode the sign of the value.

Table 5.1: Observing the reduction in number of issued binarization commands after using the new scheme for *MVD* and *coeff_abs_level_minus1*

		MVD			
Sequence	Size and number of frames	Syntax elements encoded	# of ExpG(3) commands	# of sign bin commands	Reduction in binarization commands (%)
Foreman	CIF 300	347524	29467	159127	54.26
Mobile	SD 90	684386	55952	279862	49.07
Football	SD 90	634446	144767	385977	83.65
Susie (grey)	SD 150	271010	28882	133662	59.98

		Transform coefficient level			
Sequence	Size and number of frames	Syntax elements encoded	# of ExpG(0) commands	# of sign bin commands	Reduction in binarization commands (%)
Foreman	CIF 300	461239	124739	4232	27.96
Mobile	SD 90	1943665	515510	9009	26.99
Football	SD 90	1729755	460184	2445	26.75
Susie (grey)	SD 150	295419	41006	155	13.93

mega commands per second and several times lower than the rate of generated bins.[5]

The new interface also leads to a more compact 32-bit format for the binarization command compared to the work of Sudharsanan, et al. [5]. The command format is shown in Figure 5.3-a. Figure 5.3-b lists the required binarization commands.

Direct access to the core binarizers (i.e., *ExpGolomb* and *Unary/TUnary*) is allowed to facilitate encode of other elements in the slice header or *pcm_byte* data which are not arithmetic encoded. The binarized string of these data will be directly sent to the output stream using *Direct data pass-through* interface. This interface effectively allows the bin string to jump over the arithmetic coder and reach the accumulator register of *Bit Generator* block. Alternatively, these bin string data could have been

[5]Table 7.4 shows the total number of generated bins for the full set of syntax elements of the same test contents as Table 5.1. If assuming 80% of all syntax elements are MVDs and transform coefficients, then the estimated total number of syntax elements will be $(1/0.8) * 2,628,051 \sim 3.29$ mega syntax elements for *Mobile* content which results in almost 17.15 mega bins per Table 7.4.

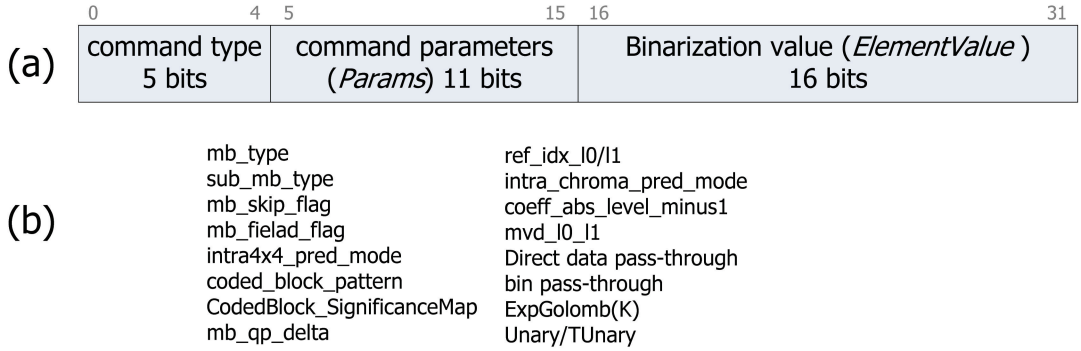


Figure 5.3: (a) The suggested format for binarization command (b) List of binarization commands.

communicated through a dedicated interface instead of the normal output FIFO.

It is important to note that derivation of context index for several syntax elements requires information about the state of same syntax elements in neighboring macroblocks (left and top macroblocks[6] adjacent to the current macroblock) as discussed in section 2.4.2. Since the binarizer does not store any information beyond the life time of executing a single binarization command, it can not resolve this dependency on the neighboring macroblocks on its own. It is still required that some "hint" about the evaluation of context index be provided to the binarization hardware by a "higher-level entity" (software or a front-end hardware) which keeps track of history of macroblock information. As a result, even this improved scheme is not completely self-contained and still relies on the upstream block though the amount of information dependence is minimal. The next section will address this issue and shows that the cost of such full-fledged binarizer will increase significantly.

The format of each binarization command of Figure 5.3 is given in appendix C. Also, calculation of the required context index hints by the upstream logic and the

[6]The definition of top macroblock is different when MBAFF (macroblock field/frame adaptive) mode is enabled. Also, sometimes the dependency is on the state of neighboring blocks "within" the same macroblock and not necessary on the neighboring macroblocks.

mapping of context index range for each command are described in the appendix.

5.3 A self-contained binarizer block

The previous section brought the CABAC interface to a higher level by passing "syntax elements" to the binarizer. But the suggested interface is still not completely self-contained and relies on the pre-binarizer stage (processor software or special hardware) to provide some "hints" for calculating context indices for generated bins which are supposed to be arithmetically encoded in the regular (also known as context-based) mode. Here the goal is to strip the upstream block from any knowledge about context index calculation, i.e., removing the hints from the interface discussed in previous section by only passing the syntax elements to the binarizer.

As section 5.1 discussed briefly, a fully self-contained implementation of CABAC which does not expose the pre-binarizer stage to CABAC details and its difficulties is favorable in some implementation scenarios. In these cases, the pre-binarization stage issues only the syntax elements to the binarizer and the binarizer will manage context index retrieval fully by itself.

This new functionality can be implemented by extending the architecture of section 5.1 and bringing in the *MB state tracker* and *Context index hint calculator* blocks of Figure 5.2 from the upper layer into the binarizer itself as shown in Figure 5.4. Now this block provides the same sort of *context hint* information previously provided as part of the binarization command parameters to each individual binarization block. As described in section 2.4.2, CABAC utilizes four different context types, *context templates*, to calculate conditional probabilities of encoded symbols. For *mb_type* and *mb_sub_type* syntax elements, a binary tree is used to derive the bin sequence and their associated context indices. For syntax elements related to significance map of the residual data, the position of the encoded syntax element in the scanning path

determines the context index. For level information of the residuals, the accumulated number of certain level values decides the context index. And for the final case, properties of the neighboring syntax elements are used to derive the context indices used for encoding the current syntax elements. These neighboring syntax elements are at the left and at the top of the current syntax element as depicted in Figure 2.5.

Out of the four discussed context types, it is easy to derive the context for all types except the neighbor-dependent type. For the position-dependent type, a simple counter is enough to track the position of each encoded element of the significance map. The counter does not need to live beyond a single *CodedBlock_SignificanceMap* binarization command call (see appendix C for description of the command) as one call encodes all elements of one batch of residual data for any block type. Even for the more difficult type of the residuals level information, implementing a couple of counters to track occurrence of certain level values would suffice.[7] The difficulty arises for the neighbor-dependent type where some syntax elements used in encode of the macroblock to the left or to the top of the current macroblock need to to be used.[8]

As a result, some of the encoded syntax elements need to be cached for further reuse in calculation of context index belonging to some future macroblocks. In the Main Profile of H.264, a slice has a rectangular shape and all of its macroblocks are encoded row by row, from top to bottom and from the left macroblock to the right macroblock of each row.[9] As Figure 5.4 shows, at encode of the n-th macroblock of

[7] At encode of current level value, the number of leading level values equal to 1 met before the current level needs to be tracked. Similarly, the number of levels with values greater than one encountered before the current level needs to be tracked. These counters were referred to as `NumT1` and `NumLgt1` in section 5.3. Note that the residual level information are encoded in the reverse scanning order and the mentioned counters track the occurrence of level data in the same reverse scanning order.

[8] This is not the case for all syntax elements. For example, for *coded_block_flag* the dependency will be on the left and upper blocks of the "same" macroblock if the current block belongs to neither the left column nor the top row of the current macroblock.

[9] Advanced slicing features of H.264 like *slice groups*, *arbitrary slice ordering (ASO)* and *redundant slices* are not supported for the Main Profile as shown in Figure 2.1 and discussed in [2]. These features are geared towards the applications with unreliable transmission and limited bandwidth

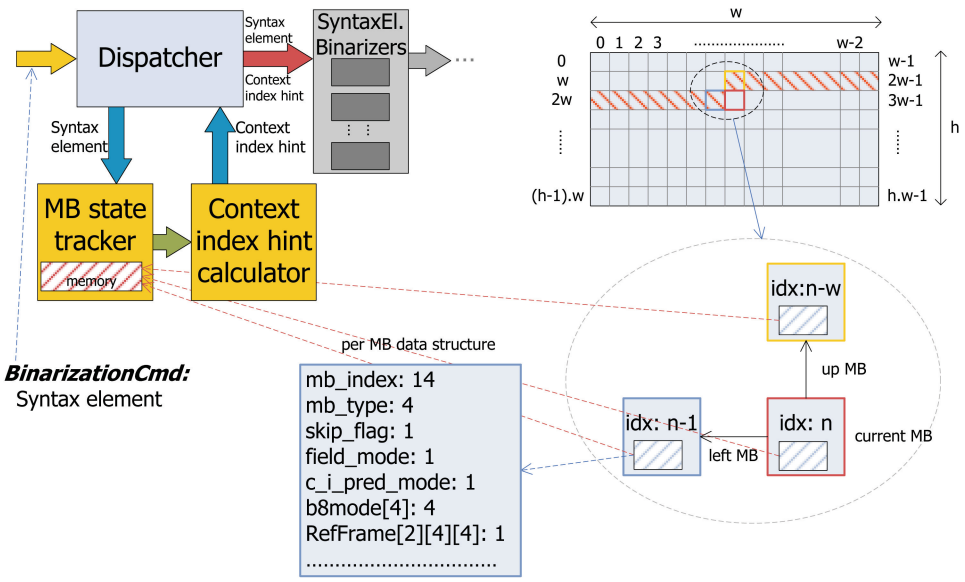


Figure 5.4: Dependency of encode of the current macroblock on the neighboring left and top macroblocks requiring cache of previously coded syntax elements.

the slice (with width and height of w and h respectively), some of the syntax elements belonging to the $(n-1)$-th and $(n-w)$-th macroblocks need to be available. Some of the cached per macroblock syntax elements and data are shown in Figure 5.4. The required memory area is referred to as *per macroblock cached data* or *per-MB structure* and indicated with blue hatched pattern (in / direction). After encode of the n-th macroblock, the $(n-w)$-th macroblock will no longer be on the left or top of any other macroblock so the per macroblock memory allocated for that macroblock can be reused for per macroblock cached data of the n-th macroblock itself. This totals to w such per macroblock data for the last w encoded macroblocks which are shown with red hatched pattern (in \ direction) in the macroblock grid.

The combined required memory will be the memory area shown in *MB state tracker* block. Instead of relying on "context index hints" calculated in the upstream block, *Dispatcher* consults *MB state tracker* block to resolve the dependencies on the

using Extended and Baseline profiles of H.264.

neighboring blocks/macroblocks and provide equivalent hints to individual binarizer blocks. The *Dispatcher* forwards a copy of received binarization command to *MB state tracker* block. The state tracker caches the syntax element info contained in the command in a "reduced form" (to be explained later), calculates the context index hint from the cached data of previous macroblocks and returns the hint to the *Dispatcher*.

The next section describes the content of the *per-MB structure* to be cached by *MB state tracker* for each of the last w macroblocks.

5.3.1 Per macroblock cached data

For context-based encode of bins of some syntax elements belonging to a particular macroblock at any time, the same syntax elements belonging to the previously coded left and top neighboring block/macroblock affect the context index calculation. This requires caching of these syntax elements to have them available as long as they can be used by future macroblocks. It will be shown that significant memory overhead is required for caching w of such per macroblock data. The complete data structures will be like an array of such per macroblock data with a size equal to a full row of macroblocks in the slice, w. But coding the whole syntax element is not efficient since in many occasions, the test for context index retrieval is only concerned with particular range of values and not the whole dynamic range (as an example, refer to *ctxIdxHint* calculation of *mb_qp_delta binarizer* in appendix C). Then, a "reduced form" of the syntax element will result in equivalent condition evaluation for related context index.

The profiles along with different levels of H.264 standard give the complete specifications about the exact requirements of an implementation like the maximum frame size, processing rate, possible range of motion vector difference, etc. as defined in Annex A of the standard document [3]. These parameters affect the allocated size for

some of the fields of per macroblock data structure. Level 4.2 is the highest level for HD contents which are the most common application of the Main Profile. Level 5 targets contents at movie quality level which limits it to very professional applications. For the below discussion, it is assumed that Level 4.2 is the highest target for this binarizer implementation and the data fields of the per macroblock data structure are allocated based on that. The list of per macroblock data and their possible form with smaller footprint is discussed below:

***mb_index*:** *13-bit wide: No allocated bit*

Since Level 4.2 supports up to 8192 macroblocks, a 13-bit value is required for macroblock index. Because the w-sized array of per macroblock data structures are for a continuous sequence of macroblocks, it can be sufficient to know the index range of per macroblock array. It is also important to know which array index is associated with the first macroblock in a macroblock row of the frame. This is because such macroblock does not have a left neighbor. In summary, it is enough to store the start index of the macroblock range at anytime and the slice width, w (in macroblock unit). The rest of required indices can be calculated on the fly. As a result, it is assumed for the rest of this discussion that storing a *mb_index* field for *per-MB structure* will not be required.

***mvd_hor*:** *6-bit wide[2][16]: 192 bits*

The horizontal range for motion vector difference in Level 4.2 is [-2048, 2047.75] which leads to $16 * 1024$ combinations for quarter-pixel accuracy requiring a 14-bit value for representation. But a 6-bit value is enough for the purpose of calculation of context hint as only *mvd* component values up to an absolute value of 33 (in integer non-quarter unit) affects the context index calculation.[10] Because each macroblock can contain up to 16 sub-blocks (with one *mvd* for

[10]Refer to derivation of `NeighboringPartitionsMVD_Sum` in *mvd_l0l1 binarizer* of appendix C for exact calculation of the hint value.

each sub-block) for each of the two reference lists (list 0 and list 1), an array of *[2][16]* of 6-bit values is needed here.

***mvd_ver*:** *6-bit wide[2][16]: 192 bits*

The maximum vertical range of *mvd* for Level 4.2 (which also applies to Level 5 and Level 5.1) is [-512, 511.75]. This leads to $4 * 1024$ combinations in quarter-pixel accuracy requiring a 12-bit value to represent. Similar to the discussion for horizontal *mvd*, an array of *[2][16]* of 6-bit values is sufficient for the purpose of context index calculation.

***ReFrame*:** *2-bit wide[2][16]: 64 bits*

The reference value index takes the range of [0, 63]. But for context hint calculation, it is enough to track the three states of equal to zero, equal to one or more than one so a 2 bit value is sufficient to record this state. Similar to *mvd* values, an array of *[2][16]* is required for the two reference lists each having up to 16 sub-blocks.

***b8mode*:** *4-bit wide[4]: 16 bits*

Each *8×8* luma block of the macroblock can be encoded in different modes like *8×8*, *8×4*, *4×4*, ... requiring 4 bits to store the mode. Each of the four *8×8* blocks of the macroblock needs such a mode value totalling 16 bits.

***cbp*:** *6-bit wide*

Each of the four *8×8* luma and the two chroma blocks of a macroblock carry a bit to reflect if the block is coded or not (i.e., skipped because a close match is found in motion estimation). This totals to six bits.

***cbp_bits*:** *51-bit wide*

This value is used for coding status of luma and chroma portions of all different block size combinations (e.g., *8×4*, *4×8*, ...) within a macroblock. It is

implemented as a 51-bit wide bit field in the reference software [12]. Refer to `write_and_store_CBP_block_bit` function in the reference software [12] for the details on usage of this bit field.

***luma_transform_size_flag*:** *1-bit wide*

This value keeps the intra prediction size for intra *8×8* and *4×4* modes.

***chroma_intra_pred_mode*:** *1-bit wide*

Though intra prediction mode for chroma blocks can take several values, for the purpose of context index calculation it would be enough to store whether this value is zero or non-zero for an encoded macroblock.

***skip_flag*:** *1-bit wide*

A single bit is required to store whether the macroblock was skipped or not for inter-predicted macroblocks.

***field_mode_flag*:** *1-bit wide*

This single bit reflects whether the macroblock was encoded in the field mode or not when MBAFF coding is active.

***mb_type*:** *4-bit wide*

This 4-bit value stores the type of the macroblock.

5.3.2 Overhead and issues

The required memory for the above fields adds up to 529 bits with almost 73% of them is related to motion vector differences. Obviously, this number would be several times higher if the syntax elements were not replaced with the reduced forms.

The specifications of H.264 levels do not specify the maximum horizontal and vertical sizes of the frame in pixels unit. Instead, the total number of macroblocks

comprising the picture is mentioned as the limiting factor. Because the number of *per-MB structures* required for caching is equal to the picture width w in macroblocks unit, a realistic upper limit for width needs to be defined. Though technically a picture of 8192 macroblocks width with a hight of one macroblock is possible based on H.264 specifications, it is assumed here that the width can not be more than 256 macroblocks which suggests a height of no less than 32 macroblocks in this case. Such large width to height ratio of 8 is a reasonable maximum assumption in almost all realistic applications. This suggests a width of $256 * 16 = 4096$ pixels which is an acceptable maximum width limit for an HD content.

There are some data of global nature that need to be tracked at anytime (e.g. the index range of cached macroblocks) but the majority of required memory is related to w copies of *per-MB structure*. The new binarizer needs to be able to allocate up to 256 *per-MB structure* which totals to 135,424 bits. Though caching a single row of *per-MB structure* is enough for pure frame and pure field encoding modes, it will not be enough for MBAFF mode where the top macroblock of a macroblock could be two macroblocks above. As a result, two full rows of *per-MB structure* is required when MBAFF mode can be on.[11] This requires a total of $2 * w = 512$ *per-MB structure* being cached which results in 270,848 ($\sim$ 265 kilo) bits.

In many HD encoding scenarios, the maximum width of the encoded pictures is 1920 pixels or 120 macroblocks. Then a history of $2 * 120 = 240$ macroblocks will suffice. The memory requirement will be $240 * 529 = 126,960$ ($\sim$ 124 kilo) bits which is much more affordable than the previous case. This emphasizes the importance of assuming sound limits on the target application to make this design feasible.

[11] In the reference software [12], MBAFF mode is automatically set to on when rate-distortion optimization mode is active.

Also, new issues come up by having the rate-distortion optimization feature enabled.[12] In current implementation of rate-distortion optimization mode in the reference software [12], speculative encode of different possibilities are done all way down to the entropy-coding level. This has the advantage of coming up with the exact cost for the final output stream to decide about the best choice for encoding mode. In real-life implementations (especially a hardware implementation), this is not a realistic implementation as it could increase the workload several hundred times. From the point of view of the new standalone hardware binarization scheme, this would require extra interfaces for save and restore of some of the fields in a data-chunk fashion. As such hardware implementation of rate-distortion optimization is highly unlikely, this is more an issue for development of the proof-of-concept software model as discussed in the next chapter.

5.4 Picking the right architecture

The original thin-layer approach makes the binarizer simpler shortening the verification phase but transfers the complexity to the firmware of the higher-level block. The syntax element binarization scheme of section 5.2 makes the binarizer aware of the core binarization steps need to be taken for each individual syntax element. This brings more complexity to the binarizer but at the same time eliminates the implicit redundancy of context index calculation between the binarizer and the upstream block unlike the earlier thin-layer binarizer approach. Now the only dependence of binarizer on the upstream block is the context index hints provided for some syntax elements.

The fully hardware-based binarizer of section 5.3 goes further and even supports calculation of context index hints by itself. The obvious advantage of this scheme is that it relieves the upstream block (firmware or hardware) from having any knowledge

[12]This feature can be enabled for the reference software [12] by setting *RDOptimization=1* in the encoder configuration file.

about the complexities and details related to context index calculation for binarization process. Here, the binarizer interface is elevated to a higher level by operating at pure syntax element level.

The main disadvantage is the extra memory overhead of around 124 to 265 kilo bits depending on the maximum width of picture. Because of the added complexities of the binarization hardware, more rigorous verification process is necessary now. This could have been delegated to the firmware in a mixed hardware-software approach if a programmable processor was available in the upstream. After all, the memory size might not be a real issue depending on the overall behavior of the encoder at higher level. This is because even a processor-based implementation using the thin layer binarizer still needs to cache the same type of *per-MB* data as it needs to provide hints about context indices. Memory waste only happens if both the upstream block and the binarizer end up caching the same data in the self-contained scenario.

On the other hand, the upstream block needs to at least pack the syntax element data and its associated parameters in binarization commands and issue them to CABAC. Even in a self-contained binarizer scenario the upstream block is still a relatively sophisticated piece of hardware as it needs to synchronize between the syntax element data it receives from other blocks (e.g., motion vector differences from motion estimator and transform coefficients from DCT block) and then issue them to the binarizer. As a result, the suitable design for a particular system architecture will depend mainly on the available computational power and programmability of the upstream block that can be spared for binarization.

Chapter 6

Hardware/Software co-design

New hardware designs increasingly take advantage of software modeling and simulation for different stages of their design these days. This varies from high-level approaches like algorithmic exploration to different types of modelings and verifications even including transistor-level simulations.

This work significantly employed software modeling in developing different models for test and development of the initial ideas and later towards testing the validity of implementation. The reference software [12] was the major tool to use as the basis of the combined hardware/software design approach used in this work.

Video standards are usually difficult and complex technical documents that could lead to different interpretations. As a result, the standard developers provide *reference software* that functions as proof-of-concept for the standard. This software is not optimized for speed or space and can not be used for real applications when hard limits in performance delivery are needed. Not only a good starting point for further development, the reference software serves as a very important learning tool when accompanied with the standard document. Its value is more appreciated after realizing that the standard document is vague in many cases and there exist discrepancies between different publications in their effort to explain the standard.

Table 6.1: Standard test contents used in different stages of this work.

Sequence	Frame size	Sequence length (frames)
Foreman	CIF (PAL)	300
Mobile	SD (PAL)	90
Coastguard	CIF (PAL)	300
Football	SD (NTSC)	90
Susie (grey)	SD (NTSC)	150

For the case of H.264 reference software in this work, several standard test sequences listed in Table 6.1 were used to evaluate the new explorations and their associated models.[1,2] These test contents are the standard test contents used by the research community [25] for development and evaluation of video coding algorithms. First, the original reference software is used in the encode mode to generate some *reference H.264 streams* using these test contents. These reference streams serve as the golden references for later simulations of the hardware models and HDL implementations. Then any new idea or model is implemented in a modified version of the reference software. To test validity of the new model, the output stream of the modified encoder software is tested against the reference stream.[3] Any discrepancy between the streams would reflect a problem with the new implementation or model.

The following sections give details about the main parts of this work which took advantage of the mentioned software modeling scheme.

[1]CIF stands for Common Intermediate Format. A CIF image is of *352x288* or *352x240* pixels depending on the sequence's video standard type, PAL or NTSC respectively. SD stands for Standard Definition which is also known as D1 size. An SD image has a size of *720x576* or *720x480* pixels depending on PAL or NTSC standards respectively.

[2]All test contents were frame-based and encoded with three B frames (e.g., the encoded frame pattern was IBBBPBBBP...).

[3]Needless to say, the same encoding parameters are used for both cases.

6.1 Design of the arithmetic coding block

Renormalization and its iterative nature was identified as one of the most challenging parts of a hardware implementation for arithmetic encode part of CABAC. As a result, context-based (regular) coding mode became the first focus of this work.

The first idea was use of a ROM as a fixed and single-cycle latency mechanism for implementation of the renormalization process as discussed in section 3.3.2. The table content was calculated based on the suggested logic of section 3.3.2 at start of the encoder. Then, instead of the regular renormalization loop inside `biari_encode_symbol` function, a new function was called to renormalize and update the coding states through ROM table lookup.

Similarly, new ideas like formation of the parsing area described in section 3.3.3 were modeled and explored. Here the parsing table was constructed once. Then, at each context-based encode of the bin, the parsing area was formed through the described rules. By accessing the parser table, the generated output bits and pointers to the potential outstanding bits area were retrieved. Packing of the generated bits was modeled based on what described in section 3.4. In all different stages of the development, the model tried to resemble a real hardware implementation like variables representing registers and functions representing functional units.

The goal of above modeling was to verify the validity of ideas at algorithmic level. Though not cycle accurate, the models were developed in such a way to make the migration to HDL code and real hardware easy. They focused more on the functional behavior of difficult parts of the design for exploring ideas and were not meant to be complete cycle accurate models.

The software model of the arithmetic coder helped a lot in speeding up the verification phase. The reference software with plug-in models was thoroughly tested

for very long sequences of test contents. To be able to increase the burden, the rate-distortion optimization mode was turned on which resulted in a bin sequence of 566 million bins for the *foreman* sequence. Most of the bugs related to functionality of the idea were caught in the model simulation which was much easier to investigate than at HDL simulation level. Mainly the bugs related to the timing and pipelining issues remained to be caught at HDL simulation phase. Enabling the model to handle rate-distortion optimization introduced many complexities. For this scenario, the complete state of the arithmetic coder including the *intermediate buffer* at output bit-packing stage needed to be saved and restored. This save and restore process required several modifications in different part of the reference software which were not directly related to CABAC functioning.

Having the models also helped to make architectural decisions. An example is in making decision about the *intermediate buffer* size and the possibility of its overflow because of very long sequence of outstanding bits. A buffer size of 32 bits can not handle an outstanding bits sequence of longer than 32 in a worst case scenario. As an example, the simulation showed such overflow scenario could not happen for *mobile* sequence which generates sequence of outstanding bits up to sizes of 42. This suggested that for real test contents, chance of stall will be less than the worst case scenario.

It should be emphasized that debugging the software model is a very cumbersome process. Because of the serial nature of arithmetic coding, if a bin is wrongly encoded, it would result in wrong encode of all later bins belonging to the current slice. Because the bugs are detected based on comparing the output bits of two different streams, occurrence of a mismatch will be the starting point to locate the bin which its encode is not correctly done. But usually the problem is started with encode of a much earlier bin before propagating to the output stream. Further, after fixing any problem, the whole simulation needs to be started from scratch as states of all data set belonging

to the previous slices (e.g., the reference frames) need to build up again from the beginning of the encode process. This is a time consuming process as running the debug version of the reference software takes several hours for a typical test content.

6.2 Collection of statistical information

Modifying the reference software at proper points allowed gathering very useful statistics about different test contents. These statistics were crucial in making design decisions because they were representative of type of the load faced by the target hardware implementation in real applications. The following is a list of such statistics.

Distribution of number of renormalization iterations:

This distribution is shown in Figure 4.8 reflecting the number of iterations it takes to renormalize coding states after encoding a bin. This distribution also implies the number of outputs bits generated at encode of a bin. Opposed to what originally thought, this distribution did not show a length of 8 output bits for the maximum size of number of iterations at renormalization. This observation initiated an investigation which proved that the generated bits at each encode can not be more than 7 bits (section B.2). This fact helped to come up with the right number of bits in the data path, parser table, and other parts of the design.

Distribution of number of outstanding bits at resolve time:

This distribution is shown in Figure 4.7 reflecting number of pending outstanding bits resolved with arrival of a non-outstanding bit. The maximum of this value was used to decide the right size of the intermediate buffer. Low frequency of long sequence of outstanding bits suggested very low probability of stalls (Table 4.2).

Percentage of bypass coded bins:

This measure (shown in Table 4.1) helped with deciding the effectiveness of an implementation based on issue of bins at non-fixed intervals rather than bin issue at fixed three cycle intervals as discussed in section 4.1.3.

Design of binarization commands:

Table 5.1 shows how different designs of binarizer interface and their associated binarization commands affect the number of commands sent to the binarizer. This results in less strict requirements for the depth of input FIFO.

6.3 Interface design for hardware binarization

The architecture suggested in section 5.3 is relatively a complex architecture affecting the binarizer interface and how the upstream block issues binarization commands to the CABAC block. Since this involves divergence from the original simpler approach, thorough test of the idea was necessary to prevent overlooking difficult points.[4] As a result, a new model based on the suggested architecture and its accompanying interface were developed for the reference software.

Both global and per macroblock structures were defined in the model. All the original functions of `cabac.c` for encode of different syntax elements were modified to route the calls to a new set of functions which used the new scheme for context index calculation. Instead of relying only on comparison of the output bitstream against the reference stream, the calculated context indices based on the new scheme were compared to the original algorithm on the fly so the discrepancies can be detected earlier.

[4]In the simplest scheme, the binarizer was considered as thin as possible delegating all context-related issues and syntax elements binarization to the upstream. Subsequently, the binarization commands were geared towards the core binarizer components discussed in section 2.4.1.

Though time consuming, the implementation of the base model went straightforward. But when rate-distortion optimization was turned on, several unexpected issues started to show up. The major problem was related to speculative coding of the rate-distortion optimization mode which required sudden save and restore of previously coded elements. This required update of global and per macroblock structures outside the normal interface of individual syntax element encode. This had to be resolved by extending the interface and accommodating special save/restore commands.[5]

This model helped significantly with developing the suggested interface.

6.4 Generation of test vectors and initial tables

For executing simulation of the hardware design in both FPGA and ASIC platforms, the input bitstream for the block was generated using the reference software. The software was modified to log the bin value, bin type and its associated context index generated at encode of a *foreman* sequence into output text files to be used by HDL simulation testbench code. Similarly, the initialization files required for different ROM tables in the design (multiplier table, next probability state, parser table) and also for the initialization of the context table RAM were all generated by modifying the reference software. The initialization files were generated in *.mif* (memory initialization file) format and the rest were generated in text format.

[5]Note that the rate-distortion optimization scheme implemented within the reference software is required to be turned on if macroblock-adaptive frame/field (MBAFF) mode is used. MBAFF mode itself requires extra work to map the neighboring left and top macroblocks to the right per macroblock structure as now two rows of per-MB data needed to be kept for this mode.

Chapter 7

Experimental results

This chapter presents and discusses the implementation results. The first implementation platform was *Altera*'s *Stratix EPS180* FPGA (Field Programmable Gate Array) with *Quartus II* version 4.2 as the design environment. This FPGA is the largest one from the older *Stratix* series with almost 80K of logic cells, 7M bits of on-chip memory and equipped with DSP blocks and hardware multipliers. The size of this FPGA is much bigger than what is required for the design of this work. Neither the DSP blocks nor the hardware multipliers were used for the design.

For the second platform, a 0.18 μm ASIC (Application Specific Integrated Circuit) design of the architecture was carried out using *Synopsys*'s *Design Compiler* and *Virage* memory compiler for a generic standard cell *TSMC* process. Both designs were written in *Verilog* hardware description language. Table 7.1 presents summary of implementation results.

The next two sections discuss the implementation results in details (including the design size and expected speed) for the main two explored architectures of sections 4.1.2 and 4.1.4. The relation of the achieved throughput for the arithmetic coding engine and the bitrate requirements of H.264 profile levels is discussed too. Finally, the simulation approach and its environment setup are discussed.

Table 7.1: Summary of implementation results.

Design	FPGA				ASIC			
	Size (cells)	Memory (bits)	Clock (MHz)	Rate (Mbps)	Size (mm^2)	Power (mW)	Clock (MHz)	Rate (Mbps)
3-cycle arch.	1373	13,056	141	47	0.423	48	263	87
Fully pipelined	1261	13,056	97	97	0.209	28	190	190

7.1 Results for 3-cycle throughput architecture

The very first implemented architecture used the ROM-lookup method of section 3.3.2 for its renormalization combined with output bit packing similar to the one of section 3.4.1. After the idea of architecture of section 4.1.2 emerged, the initial implementation was modified based on the improved architecture.

The synthesis report of this architecture is given in Table 7.2. The number of logic cells used for design is 1373 cells which is equal to 1.7% of the total cells available on *Stratix 1S80*. The majority of cells (almost 80%) were consumed by *Bit Generation* block. Most of the cells were consumed by its four-stage pipelined implementation, generated masks and shift/logical operations of the append process on the 64-bit wide intermediate buffer.

A total of 13,056 memory bits were consumed. The memory breakdown is given below.

***Context State RAM*:** Each context entry consists of 7 bits (6 bits for probability state and one bit for MPS bit) as explained in section 2.4.2 and Figure 2.9. A total of 399 entries are required for this table but 512 of them has to be allocated since the memory blocks of *1S80* must be allocated in sizes of power of two like most other FPGAs. As a result, the memory size totals to $512 * 7 = 3584$ bits. In a customized environment with exact selection of memory depth, this can be reduced to 2793 bits.

Table 7.2: Synthesis report of the initial 3-cycle throughput architecture.

Design block	Logic Cells	LC Registers	Memory Bits
Cabac	1373(5)	331	13056
ArithEncoder	242(163)	31	6400
ContextStateRAM	0(0)	0	3584
RangeRenormalizer	58(58)	0	0
RangeTableLPS	0(0)	0	2048
TransIdxMPS_LPS	0(0)	0	768
ParseShiftedLowBits	21(0)	0	0
ParseShiftedLowBits_bypass	0	0	0
Bit generator (Renormalizer)	1126(998)	300	6656
GetAppendingMasks	123(26)	0	0
CodILowParserROM	0(0)	0	6656
GetResolveBit	5(0)	0	0
BShifter_7:LeftAlignedShifter	0	0	0
BShifter_7:ResolvedBitsShifter	0	0	0

***RangeTableLPS ROM*:** This is the multiplier table which carries the content described in Table 9-33 of [3] with its addressing layout shown in Figure 2.9. It consists of 256 entries, each of 8-bit width.

***TransIdxMPS_LPS ROM*:** This ROM is used for derivation of the probability state in LPS and MPS paths. Its content is based on Table 9-34 of [3]. It takes the 6-bit value of the current probability state so it has a depth of 64 bits. The next states are two 6-bit values for LPS and MPS paths (Figure 2.9). As a result, the total size will be $64 * 12 = 768$ bits. The next state for the MPS path can be derived through

$$TransIdx_{MPS} = (ProbStateIdx! = 63)?(ProbStateIdx + 1) : 63$$

based on Table 9-34 of [3]. Though this can reduce the table size to half, this optimization was not considered here. In the ASIC's final implementation, all

ROM tables will be replaced with optimized combinational logic derived through PLA (Programmable Logic Array) minimization tools like *Berkeley*'s *Espresso*.

***Parser ROM*:** This ROM is used to process the *parsing area* (a 9-bit value) as described in section 3.3.3 and returns a 13-bit value as shown in Figure 3.6-a. This totals to $512 * 13 = 6656$ bits.

Since *Stratix* family FPGAs provide only synchronous type memories and the context-adaptive coding path requires several memory accesses, the longest path would take several clock cycles. The bottleneck for processing a bin takes 3 cycles starting from T_0 and ending at T_3 in Figure 4.1. This FPGA design currently achieves a speed of 141 MHz with the longest path of coding and renormalization taking three cycles.[1] This effectively achieves an encoding rate of 47 Mbps which is significant on an FPGA implementation and already capable of handling the maximum rate of Level 4 which is equivalent to medium quality HD contents per Table A-1 of [3].[2]

Since the bulk of the design is consumed by *Bit Generator* block, the effect of reducing the output FIFO width was investigated. Decreasing the FIFO width implies reduction of the intermediate buffer which is twice as long as the FIFO width, and its associated appending logic. If FIFO width is reduced to 16 from the original 32, *Bit Generator* size will be reduced by 34% which reduces the total design by almost 30% to 963 logic cells. As discussed in section 4.2, the disadvantage of reducing FIFO width is that in a worst-case scenario, the longest sequence of outstanding bits for stall-free encode will be reduced to 32 from 96 increasing the possibility of stall. Since this experiment has not been fully verified using proper testbench codes, its further

[1] An earlier version of this design was running up to 163 MHz as reported in [23]. A later bug fix caused a drop in the maximum speed.

[2] Note that the maximum video bit rate, *MaxBR*, is the bitrate of the output bitstream generated by CABAC. As a result, CABAC needs to be capable of higher encoding rates as entropy coding achieves compression rates in 1.24% to 1.44% range (Table 7.4). Anyway, the achieved encoding rate of 47 Mbps is easily capable of handling Level 4 with its *MaxBR* of 20 Mbps.

Table 7.3: Synthesis report of fully pipelined architecture.

Design block	Logic Cells	LC Registers	Memory Bits
Cabac	1261(29)	257	13056(14080)
ArithEncoder	382(330)	132	6400
ContextStateRAM	0(0)	0	3584
RangeRenormalizer	25(14)	0	0
RangeTableLPS	0(0)	0	2048
TransIdxMPS_LPS	0(0)	0	768
ParseShiftedLowBits	27(4)	0	0
ParseShiftedLowBits_bypass	0	0	0
DataCounter	24(0)	0	0
Bit generator (Renormalizer)	826(645)	101	6656 (7680)
GetAppendingMasks	175(64)	0	0
CodILowParserROM:inst0	0(0)	0	3328)
CodILowParserROM:inst1	0(0)	0	3328
GetResolveBit:inst0	2(0)	0	0
GetResolveBit:inst1	2(0)	0	0
BShifter_7:LeftAlignedShifter	2(0)	0	0
BShifter_7:RBitsShifter_inst0	0	0	0
BShifter_7:RBitsShifter_inst1	0	0	0

discussion will remain a future work. Anyhow, this results reflect the significance of FIFO width on the design size.

An ASIC synthesis and simulation using a 0.18 μm generic TSMC technology resulted in a 263 MHz circuit, thus enabling 87 Mbps encoding rate.[3] The circuit occupied a total of 0.423 mm^2 with an estimated power consumption of 48 mW.

7.2 Results for the fully-pipelined architecture

After the previous experience, the idea of fully-pipelined architecture was developed (presented in section 4.1.4). The synthesis report of this architecture is given in Table 7.3. The number of logic cells used in this design is 1261 cells which is equal to 1.6% of

[3]Note that this design uses the original 32-bit FIFO width.

the total cells available on *Stratix 1S80*. Similar to the previous design, the majority of cells (almost 66%) were consumed by *Bit Generation* block. The reason for this reduction in *Bit Generation* share is mainly because its pipeline stages are reduced to three from the original four. On the other hand, the fully pipelined implementation of *Arithmetic coding* block[4] increases its size to 382 cells from the original 242 cells. Furthermore, the extra new logic required for data forwarding of the updated context entry to stage T_1 of Figure 4.4 adds up to the new extra logic required for new registers between the pipeline stages.

The memory size remains at 13,056 bits though couple of changes in the organization of memory has happened. First, *RangeTableLPS* is now configured as a 64 entry ROM with a width of 32 bits for concurrent read of all four $Range_{LPS}$ values associated with current probability P_{LPS} (as explained in section 4.1.4). Second, *Parser Table* is now divided to two parts since new timing requirements of *Bit Generation* block (discussed in section 4.1.4) requires partial processing of two entries first and later selection of the target one. Again, the total size of parser ROMs does not change and remains at 6656 bits. The only difference between the two designs from memory point of view is that the new design requires a two-port memory for the context RAM while the previous design used a single-port memory. The two-port memory is required because both read and write accesses to the context RAM can happen in parallel as a result of fully-pipelined approach.[5]

The implemented architecture achieved an speed of 97 MHz which means 97 million binary symbols per second can be encoded.[6] This encoding rate is almost twice as that of the design of section 7.1 due to the fully pipelined approach. As expected,

[4]It is the combination of *Context-adaptive arithmetic coding* block and *Bypass coding* block. After making these blocks pipelined, now the whole design is fully pipelined.

[5]Note that a full two-port memory is not required as one port is always for read accesses and the other one dedicated to write accesses.

[6]An earlier version of this design [24] was capable of running at 104 MHz. Fixing a bug and completing the design for context memory initialization resulted in this minor drop in maximum clock frequency.

the longest path is now the T_1 stage of pipeline (Figure 4.4) for calculation of updated *codIRange*. The proposed fully pipelined design clearly outperforms the first design.

A 0.18 μm ASIC design of the above architecture was carried out for a generic standard cell *TSMC* process. Based on synthesis results, the design runs at 155 MHz occupying an area of 0.321 mm^2 where the memories take 76.6% of it. Here the *Bit Packer* block, stage T_4 of the pipeline, of *Bit Generator* block becomes the bottleneck. By breaking it into three stages, the speed can further be improved to 190 MHz with an area increase to 0.355 mm^2 of which 69.3% is taken by the memory elements. The estimated power consumption is 28 mW.

Also, another experiment was conducted to evaluate the effect of intermediate buffer size on the ASIC design size. If the buffer size is reduced to 32 bits using a 16-bit FIFO interface from the original size of 64 bits, the FPGA design sees a reduction of 26% in number of logic cells with the memory elements staying constant. The ASIC design is reduced in area by only 7.5% as the major portion of the design area is taken by the memories. Depending on the probability of the event of very long outstanding bits sequences, a designer could choose either a 32-bit or a 64-bit intermediate buffer. Since this experiment was not fully verified, its further investigation remains as a future work.

In another experiment, *Berkeley*'s *Espresso* tool for PLA minimization was used to reduce all ROM blocks of the design to combinational logic. This significantly reduced the design size from the original 0.355 mm^2 to 0.209 mm^2. This improvement was tested with the 6 million bin test vector from *foreman* content.

It is very important to understand the relation between input bin rate of CABAC's arithmetic coding engine and the generated output bitrate, and the implications of this relation in matching the architecture throughput with requirements of H.264 profile levels. The maximum video bitrate, *MaxBR*, column of Table A-1 of the standard document [3] defines the maximum bitrate of encoded video bitstream, i.e.,

Table 7.4: Observed compression rate of arithmetic coding in CABAC for standard test contents with disabled rate-distortion optimization.

Sequence	Total input bins	Generated output bits	Input to output ratio
Foreman	5884955	4685947	1.26
Mobile	17150489	13875007	1.24
Coastguard	10635458	8343924	1.27
Football	17638761	13789526	1.28
Susie (grey)	4871692	3392278	1.44

the output bitstream of CABAC. Because the arithmetic coding engine of CABAC is the entropy coding tool of H.264 standard, it achieves significant compression rate implying that the input stream of bins has much higher rate than what is defined for output stream in *MaxBR* column. Also, the binarization process itself achieves some level of compression but since it processes stream of syntax elements through higher level binarization commands (like the interface defined in section 5.2), its input command rate is not as high as its output bin rate (equal to the bin rate of arithmetic coding engine).

Table 7.4 shows the average compression rate achieved at the arithmetic coding engine of CABAC for standard test contents. Further study is required to find a proper bound on the maximum compression rate of arithmetic coding in CABAC. If a maximum compression rate of 1.5 is assumed for example, a throughput of $1.5 * 50Mbps = 75Mbps$ needs to be supported till a content rate of 50 Mbps targeted for Level 4.1 or Level 4.2 can be encoded safely.[7] Note that *MaxBR* measure of Table A-1 of [3] is the maximum bit rate in a second. The maximum "instantaneous" bitrate delivered to the arithmetic coding engine within a short period of time could be much higher unless it is equipped with a large input FIFO. The right FIFO size and the proper extra margin for the throughput of arithmetic coding engine is a subject for

[7]Note that the unit of *MaxBR* column of Table A-1 of [3] is either in 1000 or 1200 bits/s, e.g., it means that the value of 50,000 for Level 4.2 can translate to either 50 or 60 Mbps. Refer to VCL HRD and NAL HRD explanations in Annex A of [3] for further information.

further study.

7.3 Test and simulation scheme

Both FPGA and ASIC designs were simulated using *ModelSim* with two test vectors of sizes 1000 (for the architecture of section 4.1.2) and 6 million (for the architecture of section 4.1.4) generated from the *foreman* test content of Table 6.1. Chapter 6 discusses the choice of test contents and the process of generating test vectors from the standard test contents. Lower level timing simulation and also a final FPGA implementation integrated with *Nios* soft-processor were part of the original roadmap which were later dropped because of lack of time. Instead, the software model simulation and functional Verilog model simulation were emphasized with a long test bitstream of 300 frames of *foreman* test content.

Each row of the input vectors were associated with encode of a single bin carrying the bin polarity, bin type (regular or bypass mode) and the context index associated with the bin (if coded in context-based mode). The expected output bitstream was tested word by word as generated by the block and compared against the reference output stream generated through the golden reference. The size of the input and expected output test vectors total to more than 640 mega bytes and each execution of simulation takes hours to complete.

Initialization of the context table requires special attention. Since the context modeling is for life cycle of a slice only, it needs to be reinitialized at start of each new slice (i.e., each new frame for a sample test content). But there is more than one set of initialization values for the context table. As discussed in section 2.5.1, quantization parameter (QP) and a slice-dependent mechanism determine the initialization values of the context table. The reference software did not change QP in encode of our *foreman* test content but the slice-dependent mechanism was used. As a result, an

extra vector added to the input vectors to signal change of initialization table (see *model_number* definition in footnote 16 of chapter 4). This vector is referenced at start of every new slice and determines which initialization set to be used for resetting the context table. The initialization values are written one by one to the table within a special initialization phase at every new slice. All this process is handled in the top file of the design, `tb_top.v`.

Chapter 8

Concluding Remarks

8.1 Conclusion

This thesis explored several architectures for efficient and complete implementation of different portions of CABAC. It presented a comprehensive encoding engine and provided the tools for efficient implementation of it. To the best of our knowledge, this work is the most complete solution proposed for CABAC encoding.

Algorithms of CABAC building blocks were closely examined to exploit any venue for design of a fast and efficient architecture targeting real-time encoding rates for HD contents. In-dept analysis of related design issues were provided and improvement possibilities were explored. The well-known serial nature of arithmetic coding makes it very difficult to use the traditional parallel architecture techniques for CABAC. As a result, a high throughput pipelined architecture became the natural choice. Different blocks were decoupled from each other and pipelining used at each level. After tackling many issues, an efficient pipelined design emerged at the end. Several of such issues are listed below.

- The variable-sized iterative *renormalization process* of CABAC's arithmetic

coding was flattened by forming and processing a *parsing area*. The renormalization process was decoupled to two stages: rescale of the coding states and bit generation.

- The *bypass coding* process was rearranged to share the renormalization process with the *context-based coding*. This technique streamlined the renormalization stage of arithmetic coding.
- The issues related to the generated *outstanding bits* (e.g., tracking, resolving, buffering and handling long sequence of them) were addressed thoroughly to eliminate overflow conditions. Also, the design decisions significantly reduced the probability of potential stalls due to long sequence of outstanding bits.
- Several solutions for binarization were proposed to support different system architectures and upstream blocks with different level of programmability and processing capability. Each design provided a different level of abstraction with different degree of hardware support. The self-contained design fully relieved the higher level hardware from getting involved in the binarization process.

An efficient architecture for CABAC's arithmetic coding and bit generation evolved through several designs. They were implemented on *Altera*'s *Stratix 1S80* FPGA and also on a 0.18 μm ASIC design for a generic standard cell *TSMC* process. The final fully-pipelined design of this work, which we believe has not been reported in the literature by others, achieved a throughput of 97 Mbps on the FPGA design and 190 Mbps on the ASIC design. A 190 Mbps rate for encode of input bins can easily handle maximum output bitrate of 50 Mbps for Level 4.1 and Level 4.2 of H.264. Even it can be marginally within the safe range for Level 5 with its 135 Mbps bitrate requirement. Higher throughput is expected using more advanced processes like 0.13 μm which can make Level 5.1 with its 240 Mbps output bit rate within reach.[1] An

[1]Refer to section 7.2 for a discussion on the relation between bin encoding rates of arithmetic coding engine of CABAC and the rate of generated output bitstream.

example commercial implementation by *Ateme* boasts an encode rate of one bit per 4 cycles achieving 37.5 Mbps at a clock rate of 150MHz [26].

The H.264 reference software [12] was significantly used in investigation and development of hardware-friendly models for different blocks. These models were plugged into the reference software to simulate full-fledged test contents. This approach is a good example of hardware/software co-design for high performance SoC design and can be used as a template for similar design scenarios. In addition, the reference software was modified to probe different portions of the H.264 algorithm and gather statistical information which significantly helped in making design decisions.

8.2 Future work

The core work for an efficient implementation of H.264's CABAC is completed in this work. Some suggestions for future work include:

- The size of the intermediate buffer of the bit generation block (implying the width of output FIFO) has considerable effect on the size of the design. Study of more test contents will provide better statistics about the maximum length of outstanding bits which will eventually help to allocate the right size of buffer.

- With the experience gained through this research, a design for CABAC decoding should be more straightforward, however, with its own challenges. Update of coding states and the renormalization process are much simpler in the decoding side. The outstanding bits are not an issue here and each context-based or bypass decode process generates a bin after comparing *offset* and *range* values.[2] The main challenge here is feeding the arithmetic decoder with a continuous stream of decoding requests. The type of each decoding request, i.e.,

[2] *Offset* and *range* are the coding states of arithmetic decoder in CABAC.

context-based or bypass, and the context index for each context-based request can depend on the outcome of previous decoding requests, i.e., generated bins. Similar problem also exists at a higher level when the decoding algorithm issues a syntax element decode command to CABAC. The type of next syntax element to be decoded can depend on the value of previous syntax elements.

Unlike the encode path of Fig. 2.8, syntax element decode commands and bin decode requests can not be issued and buffered in FIFOs uninterruptedly. This will lead to long idle cycles for the arithmetic decode core because it has to wait for the result of partial de-binarization after each bin decode or partial syntax element decode at the higher level. One possible solution is to use *predicated instructions* [27] for bin decode requests. Then, decode instructions are issued speculatively by the de-binarizer and conditioned on the value of previous decoded bin. Although more difficult, similar scheme could be extended to speculative decode commands at syntax element level which are issued by the high-level decoding algorithm. Further, the arithmetic decoding block should not use a deep pipeline because of the dependency on the end result. Also, the command FIFOs should be capable of skipping over one or several commands when some predicated commands are nullified after failing their condition.

- Considering the achieved throughput in this work, it is expected to reach higher performance even for H.264's Level 5.1 by using standard cells geared for finer manufacturing processes like 0.13 μm and 90 nm.

Depending on the system-level architecture that employs this design, some extra engineering and deployment issues need to be addressed.

- It is important to have a more solid study of both the average rate and the maximum instantaneous rate of compression achieved by arithmetic coding engine.

Simulation of encode of a wider range of test contents (with varied encoding parameters) can be a good start. These measures will help to clock the design proper to the target profile levels without incurring the costs of an over-design. Similarly, more study of the syntax element and input bin bursts reaching respectively the binarizer and arithmetic coding blocks will help to come up with the right depth of FIFO for these stages. Larger FIFO depths can damp the instantaneous bursts decreasing the required maximum throughput. Such study will help to find the sweet spot for combination of FIFO depths and the design clock rate to minimize the design size, power, cost, etc.

On the other hand, recent advances in adaptive clocking techniques [28] can make the above issues less critical by making it possible to adapt the clock frequency of binarizer and arithmetic coding pipes according to the portion of their input FIFOs filled by commands. When the binarization command FIFO or bin FIFO get nearly filled up, they can ramp up the clock frequency of the subsequent binarization or arithmetic coding pipes respectively to handle the burst period quickly.

- The context table initialization at each slice needs to be implemented efficiently. The large number of combinations make it infeasible to simply load the precalculated table from ROM. A possible solution is to store parameter pair $(\sigma_\gamma, \varpi_\gamma)$ of each context in a ROM and calculate the initial values on the fly at every slice (section 4.3.1).

- The CABAC block needs to have its own set of programmable registers for configuration and status reporting purposes. For example, a sample status register can report the number of valid bytes in the last generated word. The motivation for such register is that at the end of each slice, padding zeros will byte-align the final output bits. Because the FIFO word size is several bytes, e.g., 32 bits,

not all bytes of the last word carry valid data. An status register can allow the upstream block to query CABAC block about this. Other status registers can report status of different FIFOs to allow the upstream block interact with CABAC's input/output interface effectively. CABAC block can be equipped with interrupt generation capability to make this notification mechanism more efficient.

- In real implementation scenarios (unlike the reference software), it is not probable that the rate distortion optimization scheme employed by the higher-level encoding algorithm will use the actual number of post entropy-coded bits as a measure of rate distortion optimization. But in case such approach is implemented, the whole state of CABAC (e.g., register set, intermediate buffer, output FIFO) need to be saved and restored since speculative encoding happens. This will be very much like the context-switch process in a programmable processor.

Bibliography

[1] Peter Symes. *Digital Video Compression*. McGraw-Hill, 2004.

[2] Iain Richardson. *H.264 and MPEG-4 Video Compression*. John Wiley & Sons, 2003.

[3] ITU. *ITU-T Recommendation H.264: Advanced video coding for generic audio-visual services*, May 2003.

[4] D. Marpe, H. Schwarz, and T. Wiegend. 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.

[5] S. Sudharsanan and A. Cohen. A hardware architecture for a context adaptive binary arithmetic coder. In *Proc. of the SPIE, Embedded Processors for Multimedia & Communications II*, pages 104–112, Mar. 2005.

[6] J. L. Núñez and V. A. Chouliaras. High-performance arithmetic coding VLSI macro for the H.264 video compression standard. *IEEE Transactions on Consumer Electronics*, 51(1):144–151, Feb. 2005.

[7] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, August 2003.

[8] Tung-Chien Chen, Yu-Wen Huang, and Liang-Gee Chen. Analysis and design of macroblock pipeline for H.264/AVC VLSI architecture. In *Proc. International Symposium on Circuits and Systems*, volume II, pages 273–276, 2004.

[9] Y. Hu, A. Simpson, K. McAdoo, J. Cush. A high definition H.264/AVC hardware video decoder core for multimedia SoCs. In *Proc. International Symposium on Circuits and Systems*, pages 385–389, 2004.

[10] R. Osorio and J. Bruguera. Arithmetic coding architecture for H.264/AVC CABAC compression system. In *Proc. Euromicro Symposium on Digital System Design*, pages 62–69, 2004.

[11] W. Pennebaker and J. Mitchell. *JPEG: Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[12] ITU. *H.264/AVC Reference Software*. http://iphome.hhi.de/suehring/tml, ver. JM 8.2, July 2004.

[13] T. Wiegand, G. Sullivan, G. Bjntegaard, 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.

[14] Iain Richardson. *H.264 Variable Length Coding tutorial*. http://www.vcodex.com/h264.html, 10 2002.

[15] W.B. Pennebaker, J.L. Mitchell, G.G. Langdon, and R.B. Arps. An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, 32:717–726, 1988.

[16] D. Taubman and M.W. Marcellin. *JPEG2000 Image Compression: Fundamentals, Standards and Practice*. Kluwer, Boston, MA, 2002.

[17] D. Marpe and T. Wiegand. A highly efficient multiplication-free binary arithmetic coder and its application in video coding. In *Proc. IEEE Int. Conf. Image Proc. (ICIP)*, Barcelona, Spain, September 2003.

[18] A. Moffat, R.M. Neal, and I.H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998.

[19] I.H. Witten, R.M. Neal, and J.G. Clearly. Arithmetic coding for data compression. *Commuincation of ACM*, 30(6):520–540, 1987.

[20] ITU. *JPEG2000 image coding system: Core coding system.* ITU-T Rec. T.800, *E* edition, August 2002.

[21] M. Dyer, D. Taubman, S. Nooshabadi. Improved throughput arithmetic coder for JPEG2000. In *Proc. IEEE International Conference on Image Processing (ICIP '04)*, volume 4, pages 2817–2820, October 2004.

[22] Yu-Wei. Chang, Hung-Chi Fang and Liang-Gee Chen. High performance two-symbol arithmetic encoder in JPEG2000. In *Proc. IEEE International Symposium on Consumer Electronics IEEE Int. Conf. Image Proc. (ICIP)*, pages 101–104, Reading, UK, September 2004.

[23] H. Shojania and S. Sudharsanan. A high performance CABAC encoder. In *Proc. of the 3rd International IEEE Northeast Workshop on Circuits and Systems (NEWCAS'05)*, June 2005.

[24] H. Shojania and S. Sudharsanan. A VLSI architecture for high performance CABAC encoding. In *Proc. of the SPIE, Visual Communications and Image Processing 2005 (VCIP2005)*, pages 1444–1454, July 2005.

[25] Iain Richardson. *Video Codec Design.* John Wiley & Sons, 2002.

[26] Ateme SA. *Ateme MPEG-4 AVC-H.264 CABAC/CAVLC IP Datasheet*. http://extranet.ateme.com/download.php?file=447, 2005.

[27] John Hennessy, David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.

[28] Augustus K. Uht. Uniprocessor Performance Enhancement through Adaptive Clock Frequency Control. *IEEE Transactions on Computers*, 54(2):132–140, Feb. 2005.

[29] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufman Publishers, 2nd edition, 2000.

[30] Thomas M. Cover, Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[31] R. G. Gallagher. Variations on a Theme by Huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, November 1978.

Appendix A

Entropy coding review

This appendix briefly reviews entropy coding concepts, and Huffman and arithmetic coding techniques. A quick comparison of Huffman coding and arithmetic coding is provided too. The main reference of this review is [29].

A.1 Entropy coding

Entropy is a measure of unpredictability and can be used to represent the amount of information carried in a message. A perfectly predictable event does not carry any information. Shannon quantified this measure of information by defining *self-information* quantity. Assume an event X is a set of outcomes of some random experiment with $P(X)$ representing its probability. Then the self-information associated with X will be defined by

$$I(X) = \log_2 \frac{1}{P(X)} = -\log_2 P(X) \tag{A.1}$$

which shows the number of bits[1] required to represent information carried with a particular experiment [29].

Assume a *discrete* source S randomly generates a sequence of *symbol*s from a limited known set, *alphabet*, $A = \{a_1, a_2, ..., a_N\}$ with size of N. The entropy of such source is defined as the average self-information associated with this random experiment X which has generated the sequence $\{X_1, X_2, ..., X_m\}$ of length m. Then the entropy can be presented by the measure $H(X)$ [29] as

$$H(X) = \lim_{m\to\infty} \frac{1}{m} \sum_{i_1=a_1}^{a_N} \sum_{i_2=a_1}^{a_N} .. \sum_{i_m=a_1}^{a_N} P(X_1 = i_1, X_2 = i_2, .., X_m = i_m) \log_2 P(X_1 = i_1, X_2 = i_2, .., X_m = i_m). \tag{A.2}$$

For a *discrete memoryless source* (DMS), the probability of any generated symbol is independent of previously generated symbols. This means for every $a_i, a_j \in A$ there will be $P(X = a_i, X = a_j) = P(X = a_i)P(X = a_j)$. If besides the independence property, the sequence elements have the same distribution too, i.e., they are generated from an *independent and identically distributed* (iid) distribution, the source entropy will be reduced to

$$\begin{aligned} H(X) &= \sum_{i_1=a_1}^{a_N} P(X = i_1) I(X = i_1) \\ &= -\sum_{i_1=a_1}^{a_N} P(X = i_1) \log_2 P(X = i_1) = -\sum_{1}^{N} P(X) \log_2 P(X). \end{aligned} \tag{A.3}$$

Entropy can also be treated as a measure of the average number of binary symbols needed to code the output of the source. Shannon showed that, in average, a lossless

[1] In general, a base of b is used in the above formula instead of 2 and the unit of self-information will be in base b instead of bits.

compression method can not achieve any better than entropy of the source [29] so entropy can serve as the lower bound of the average bitrate.

In many sources, there exists sample-to-sample correlation between the samples of sequence generated by the source. Depending on the assumptions about the structure of the sequence, a *model* can be built for the sequence. For example, a *residual* sequence can be built through

$$r_i = x_i - x_{i-1} \tag{A.4}$$

where r_i is the i-th element of the residual sequence and x_i is the i-th element of the original sequence [29]. Then encoding the residual sequence might "reduce the entropy" measure. Note that here the "estimate of entropy" is reduced not the actual entropy since the entropy does not change as long as the information generated by the source is preserved [29].

In practice, it is difficult to know the real structure of the sequence but "learning" from the sequence can improve estimation of the source entropy. This can be achieved by picking larger size of block of data and letting the block size m grow to infinity as Equ. A.2 shows. But this approach is not applicable in practice and an accurate model of the source is preferred [29]. For some sources, the physics of the data generation is known and can be used for constructing a *physical model*. But the physics of the problem can be too complicated to allow building such model. In that case, models based on empirical observation of the statistics of the data are preferred [29].

If symbols generated by a source are independent and there can be a *probability model* defined like $P = \{P(a_1), P(a_2), ..., P(a_N)\}$ for them, then entropy of the source can be computed using Equ. A.3. There are efficient coding techniques for such source but if the source symbols are dependent (i.e., source has memory), there exists other compression schemes performing better than the ones designed with the independence assumption [29]. This is because the dependence structure carries information itself

which can result in better "estimate of the entropy".

Markov models are a very popular method of representing dependence in data. A particular type of Markov process called *discrete time Markov chain* is commonly used for models in lossless compression [29]. If $\{x_m\}$ is a sequence of generated symbols by a source, the source follows a kth-order Markov chain if

$$P(x_m|x_{m-1}, ..., x_{m-k}) = P(x_m|x_{m-1}, ..., x_{m-k}, ...). \tag{A.5}$$

This means that knowledge of the past k symbols is equivalent of the knowledge of the entire past history of the source [29]. The first-order Markov model ($k = 1$) is the most commonly used order of the Markov chain. A sequence that follows a Markov chain, for example of order of one, can have different forms of dependence. If the dependence is linear, it can be represented by

$$x_m = \rho_m x_{m-1} + \varepsilon_m. \tag{A.6}$$

Of course, the dependence of a sequence modeled by a Markov chain can be nonlinear in general. If the model parameters, e.g. (ρ_m, ε_m) in above case, are not changing with m, the model is called *static* like the model of Equ. A.4. But the model is *adaptive* if its parameters adapt with m to the changing characteristics of the sequence [29].

H.264 uses Markov model in its probability estimation and adaptivity in model determination. Two widely used entropy coding techniques are Huffman coding and arithmetic coding which are the basis of CAVLC and CABAC employed in H.264 standard. The following sections briefly discuss each technique.

A.2 Huffman coding

A common method of entropy coding defines a *codebook* through assigning a code to each symbol. By assigning smaller codes to the more frequent symbols, average size of each coded symbol can be minimized. This leads to compression over sufficiently large number of encoded symbols [2]. This technique is known as *variable length coding* (VLC) and different variations of it are widely used in compression standards. Generally, VLC shows a better performance than fixed-length codes where same-sized codes are assigned to all symbols.

Huffman coding assigns a VLC to each input symbol where the code and its size are based on the probability of occurrence of the associated symbol. Before code assignment and constructing the codebook, it is necessary to calculate the probability of symbols. Then the codes can be derived by constructing a code tree through a recursive algorithm [2, 29]. If the probability distribution used for deriving the codes is accurate, compression can be achieved. Huffman coding generates an optimal *variable-length code* for the related input sequence but it suffers from several major deficiencies.

First, Huffman coding requires knowledge of probability distribution of the symbols. Because this distribution is not known generally, it needs to be calculated. This means that the full sequence needs to be processed first (e.g. through an initial pass) only to derive the probability distribution. Then the second pass performs the encoding based on the calculated distribution. This two pass process may introduce unacceptable delay and prevent on the fly encoding. Second, besides the encoder, also the decoder needs to know the codebook to be able to decode the bitstream. Transmitting the codebook to the decoder is an extra overhead which reduces the compression efficiency especially for short sequences.

Third, Huffman coding, similar to other variable-length coding techniques, assigns

an integer number of bits to each symbol. This generally leads to less efficient compression as the code size will be more than the amount of information carried by the symbol which is a fractional number. For distributions with skewed probabilities for some symbols, this issue becomes more problematic as there will be lots of waste for high probable symbols, e.g., with more than 50% probability.

The average code length, also known as *coding rate*, of a coding technique for a source with alphabet $\{a_1, a_2, ..., a_N\}$, probability model of $\{P(a_1), P(a_2), ..., P(a_N)\}$, and the code length set of $\{l_1, l_2, ..., l_N\}$ corresponding to the alphabet set will be defined [29] by

$$R = \bar{l} = \sum_{i=1}^{N} P(a_i) l_i. \tag{A.7}$$

The upper bound of the coding rate is a useful measure in comparison of coding performance of different techniques. It is known that optimal codes including Huffman guarantee a coding rate R of within 1 bit of the entropy $H(X)$ [30] or

$$H(X) \leq R_{Huffman} < H(X) + 1. \tag{A.8}$$

Another bound for Huffman coding rate (usually a tighter bound) can be expressed by

$$H(X) \leq R_{Huffman} < H(X) + p_{max} + 0.086 \tag{A.9}$$

where p_{max} is the probability of the most frequently occurring symbol [31]. For applications which the probability of occurrence of different symbols are skewed (usually with small alphabet size), the value of p_{max} can be quite large and the code becomes less efficient as the upper bound will be larger.

To reduce upper bound of the coding rate, *Extended Huffman Codes* form a new alphabet by assigning a "supersymbol" to each sequence of m symbols of the original

alphabet effectively encoding a block of m symbols in each step. These codes can achieve closer rate to entropy of

$$H(X) \leq R_{ExtendedHuffman} < H(X) + \frac{1}{m} \tag{A.10}$$

for a block size of m [29]. By increasing the block size m, the coding rate can be made arbitrarily close to the entropy [30]. But this brings its own implementation problems which are discussed in the next section.

To address the first and second issues of Huffman Coding, recent image and video standards define sets of codeword tables based on the probability distribution of "generic" video materials using a set of test video contents [2]. The third issue is compensated by *run-length coding* [29]. These remedies improve the compression efficiency partially but can not fully mask the problem. Arithmetic coding presented in the next section does not suffer from these issues.

A.3 Arithmetic coding

The variable length coding technique described in the previous section suffers from the sub-optimality of assigning an integer number of bits to each symbol since the optimal number of bits for a symbol is generally a fractional number based on the information content of the symbol. This is especially the case for symbols with probabilities greater than 0.5 as the shortest code to assign a symbol can not be "less than one bit".

Arithmetic coding is a practical alternative to Huffman coding and converts a "sequence of symbols" into a single number that can approach more closely the optimal fractional number of bits required to assign to each symbol [2]. Each sequence is "tagged" with a unique identifier. An interval like the unit interval of [0,1) is a valid range of numbers to tag all possible sequences as it holds infinite number of

numbers. Then a function maps every sequence of symbols to a number in this interval. The *cumulative distribution function* (cdf) of the random variable associated with the source is the tagging function used in arithmetic coding [29].

First, the interval is divided proportional to the probability of symbols and each subinterval marked with its associated symbol. In other words, the subinterval positions $I_0, \ldots, I_N$ can be derived through the cumulative density function as [2]

$$F_X(k) = \sum_{i=1}^{k} P(X = i) = \sum_{i=1}^{k} P(a_i) \qquad k = 1, ..., N \qquad \text{(A.11)}$$

$$I_0 = \begin{cases} 0, & \text{if first encoded symbol} \\ I_{prev\_symbol\_index-1}, & \text{otherwise.} \end{cases} \qquad \text{(A.12)}$$

$$I_N = \begin{cases} 1, & \text{if first encoded symbol} \\ I_{prev\_symbol\_index}, & \text{otherwise.} \end{cases} \qquad \text{(A.13)}$$

$$I_k = (I_N - I_0)F_X(k) + I_0 \qquad k = 1, ..., N-1 \qquad \text{(A.14)}$$

$$subinterval_k = [I_{k-1}, I_k) \qquad k = 1, ..., N. \qquad \text{(A.15)}$$

With encode of each symbol, the subdivision related to that symbol is picked as the new interval and subdivided again proportional to the probability of symbols. This operation is recursively continued till the full sequence is encoded. Then a number from the final sub-division is picked to represent the encoded sequence.

Equations A.11 to A.15 perform the described process. First, the *cdf* associated with each symbol in the alphabet list is calculated based on Equ. A.11. In a fixed symbol probability scenario, the *cdf* calculation needs to be done only once. Then for each to be encoded symbol, Equations A.12 to A.15 are calculated iteratively. For the first encoded symbol, the interval range is $[0, 1)$ so the initial interval is set

[2] Equations A.11 and A.14 are from [29].

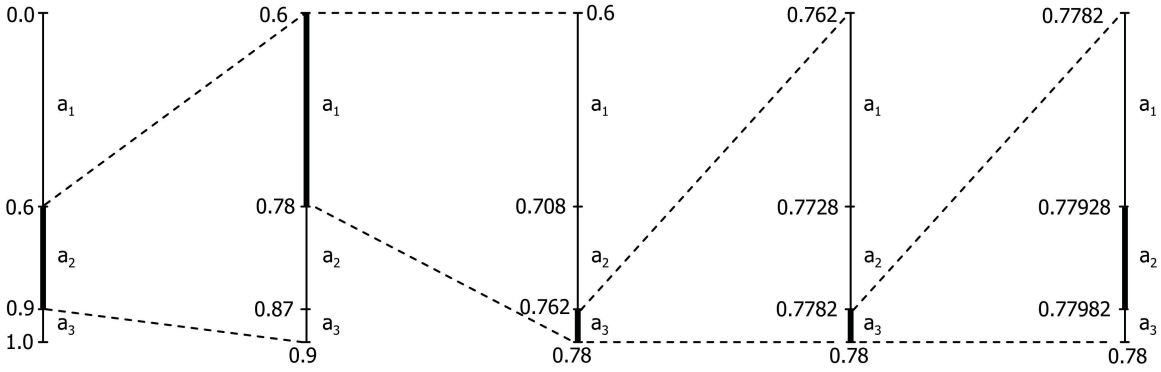


Figure A.1: Arithmetic encode of sequence $\{a_2, a_1, a_3, a_3, a_2\}$ in a turnery alphabet.

accordingly (Equ. A.12 and A.13). For other symbols, I_0 will be equal to the start of the subinterval associated with the previous encoded symbol. Similarly, I_N will be equal to the end of the subinterval associated with the previous encoded symbol. *prev_symbol_index* is the index of the previous encoded symbol in the alphabet set, e.g. if the symbol was a_p, then $prev\_symbol\_index = p$.

Figure A.1 shows a sample turnery arithmetic encode of a sequence with the following properties:

$$alphabet = \{a_1, a_2, a_3\}$$
$$p(a_1) = 0.6,\ p(a_2) = 0.3,\ p(a_3) = 0.1$$
$$sequence = \{a_2, a_1, a_3, a_3, a_2\}$$

At each iteration, the subinterval marked with ticker line is associated with the encoded symbol and picked as the new interval. At encode of the last symbol, any number between $[0.77928, 0.77982)$ subinterval can be picked as the sequence representative. The start of subinterval, 0.77928, is usually picked. To simplify the presentation, this example ignored update of the symbol probabilities after encoding each input symbol. Otherwise, statistics and probability of the symbols would have been updated at each iteration. The ratio of the subinterval associated with the encoded

symbol would have grown because of the statistics update. This means recalculation of Equation A.11 at every iteration unlike the previous simplified assumption of fixed probabilities. In a real scenario, symbol probabilities are initialized with predefined generic values and adapted based on statistics of encoded symbols. As a result, there is no need to communicate the symbol probabilities to the decoder side as the decoder is initialized with the same predefined values and updates its statistics after decode of each symbol similar to what happens at the encoder side.

An adaptive q-ary arithmetic coder operating on a q-ary source alphabet is in general a computationally complex operation for $q > 2$ requiring at least two multiplications for each symbol encode as well as update of probability estimation [4]. CABAC opted for adding a *binarization* pre-processing stage to assign a binary string to each incoming symbol and then encode each binary symbol using a *binary arithmetic coder*. More details on binarization stage and binary arithmetic coder employed in CABAC are given in section 2.4. The overhead of coding multiple bins of the binary string instead of using one pass in an q-ary arithmetic coder can be compensated by using a fast binary coding engine [4].

The upper bound for arithmetic coding is

$$H(X) \leq R_{Arithmetic} < H(X) + \frac{2}{m} \tag{A.16}$$

when a sequence of length m is encoded [29]. Equation A.10 similarly showed that the upper bound of extended Huffman coding can be arbitrarily made close to the entropy by increasing the block size m. At first glance, comparing Equations A.10 and A.16 might lead to the conclusion that Huffman coding is superior to arithmetic coding though its advantage decreases with increasing m [29]. However, building Huffman codes for block size of m requires building the entire code for "all possible sequences of length m" which means a codebook size of N^m for alphabet size of N.

Such a codebook is not a viable solution even for reasonable values [29] of $N = 16$ and $m = 20$ which require a codebook size of 16^{20}. This issue is not the case for arithmetic coding where only the code for the tag corresponding to "the given sequence" of length m is needed and not the entire code for "all possible sequences". This means when design limitations exist in practice, arithmetic coding can achieve rates closer to the entropy than Huffman coding for most sources.

Also, for arithmetic coding the code is generated on the fly and there is no need to generate the entire codebook a priori. This allows much higher flexibility in adaptation to the input statistics compared to Huffman coding. These facts allow separation of modeling and coding stages of the compression process and its design advantages is discussed in section 2.4.2.

Appendix B

Proofs

Here the proofs for some statements made in earlier chapters regarding the coding states and their validity are given. All the proofs are based on our work though not all were required for the hardware implementation.

The relations given in section B.1 were implied in the standard document. The standard document had already set limits regarding coding state size implying their validity after all coding paths. Nevertheless, we decided to prove them to make sure their limits are right since in a hardware implementation of arithmetic coding, overflow and loss of even a single bit can not be tolerated. The proofs given in later sections helped us to design the hardware in more efficient way.

B.1 Proofs for validity of coding states

Section 3.2 mentioned that the coding states of the arithmetic coder are represented by a 10-bit value for *codILow* (start of the coding interval) and a 9-bit value for *codIRange* (size of the interval). As suggested there and in section 3.3, the subtraction operation for update of *codIRange*, the addition for update of *codILow*, and also the renormalization process always generate a new interval that can still be represented

with a 10-bit *codILow* and a 9-bit *codIRange*. This means that the updated *codIRange* never exceeds $2^9 - 1 = 511$. Also, both start and end of the region never exceed $2^{10} - 1 = 1023$. These can be shown as below:

$$0 < codIRange < 512 \qquad \text{range size} \tag{B.1}$$

$$codILow < 1024 \qquad \text{start point} \tag{B.2}$$

$$codILow + codIRange - 1 < 1024 \qquad \text{end point} \tag{B.3}$$

The coding states are updated in regular coding (Fig. 2.12), bypass-coding (Fig. 2.15) and renormalization (Fig. 2.13). By looking into the flowcharts of mentioned figures, it can be easily verified that the below condition also holds after each of the above processes.

$$256 \leq codIRange \tag{B.4}$$

It will be shown here that update of the coding states will not cause underflow or overflow and represent a valid interval (Equ. B.1) that both its start and end points can be represented by a 10-bit value (Equ. B.2 and B.3). *codILow* and *codIRange* are initialized with 0 and 510 respectively (section 2.5.1) before encode of the first bin in every slice and the above conditions hold at the beginning. As each of the three mentioned processes (regular and bypass coding, and renormalization) generates a valid interval and the fact that the interval is not updated by any other part of the arithmetic coding algorithm, it can be assumed that before start of each process the coding states are already within the valid range defined by Equations B.1 to B.3.

B.1.1 Regular coding

Proof. The coding states remain in valid ranges after encoding a regular-coded bin. As the flowchart of Fig. 2.12 shows, step 2 of the chart divides the interval range by

retrieving $codIRange_{LPS}$ from the multiplier table and calculating $codIRange_{MPS}$ through $codIRange_{MPS} = codIRange - codIRange_{LPS}$. This subtraction does not cause an underflow as this is just the division of the range into two sub-region. It can also be verified by noting that (per Equ. B.4) $codIRange \geq 256$ holds all the time before start of regular (or even bypass) coding and the maximum value of $codIRange_{LPS}$ defined in the *Multiplier Table* is 240 (Table 9-33 of [3]).

No matter if *MPS* or *LPS* path is taken in Fig. 2.12, update of $codILow$ and $codIRange$ in step 4 still generates valid range size, start and end points as the new range is just a subset of the original range. Equations B.1 to B.3 will be still valid for the new subrange because they were valid for the original range. This can be seen in Figure 2.6 too.

□

B.1.2 Renormalization

Proof. The coding states remain in valid ranges after the renormalization process.
As shown in Fig. 2.13, the renormalization process involves several left shift of $codIRange$, equal to n, till it reaches a minimum of 256. Since the number of shifts depends on the leading zero of $codIRange$, Equ. B.1 is valid at the end. At each iteration, either the top bit or the bit next to the top bit of $codILow$ (bits 9 and 8) is reset and a lift shift happens. As a result, Equ. B.2 will hold at the end of renormalization too. Proving that the region end point after the renormalization still obeys Equ. B.3 is a bit more difficult though.

It is shown here that the following stronger condition holds after renormalization

$$codILow + codIRange < 1024. \tag{B.5}$$

Here two separate scenarios are considered. In the first case, the top n bits of $codILow$

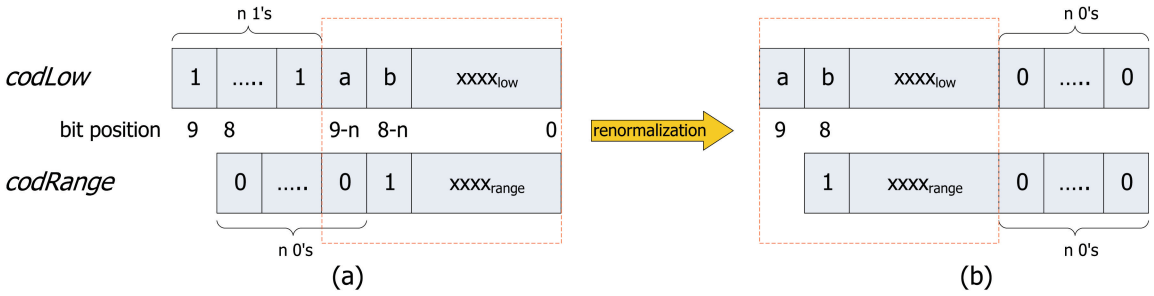


Figure B.1: Scenario 1: Top n bits of *codILow* are all 1's.

are all ones before renormalization so all of the n iterations in the renormalization process go through the right branch (Fig. B.1-a).[1] Because the right branch is taken at all n iterations, the top bit of $codILow$ is set to zero before the left-shift at each iteration. This results in Fig. B.1-b. We need to show that the renormalized coding states $\{a, b, xxxx_{low}, n\{1'0\}\}$ and $\{1, xxxx_{range}, n\{1'0\}\}$ will not result in an interval with its end exceeding 1023 (Equ. B.5). Since the n trailing bits of both coding states are all 0, this is equivalent to checking overflow for addition of $\{a, b, xxxx_{low}\}$ and $\{1, xxxx_{range}\}$. But it is obvious that such an overflow can not happen by comparing the marked rectangles of Fig. B.1-a and Fig. B.1-b. Otherwise, overflow was unavoidable in Fig. B.1-a since all the top n bits of $codILow$ before renormalization are all 1's. Then a carry bit resulting from addition of the lower $9-n$ bits would have resulted in an overflow at the first place. This is not possible as it would be against our assumption that the interval is valid before renormalization and Equ. B.5 holds.

In the second scenario, not all the top n bits of $codILow$ are 1. In that case, the right branch of the renormalization process is taken consecutively for number of leading 1's and then the middle or left branch is taken for the rest of iterations. Note that the branch choice depends on the polarity of $codILow$'s second top bin

[1] *xxxx* and similarly *yyyy* used from now on represent an arbitrary bit string with their size implied within the context. For example, in Fig. B.1 the size of $xxxx_{low}$ is *8-n* bits because *codILow* register has 10 bits and *2+n* bits of it is shown by other fields.

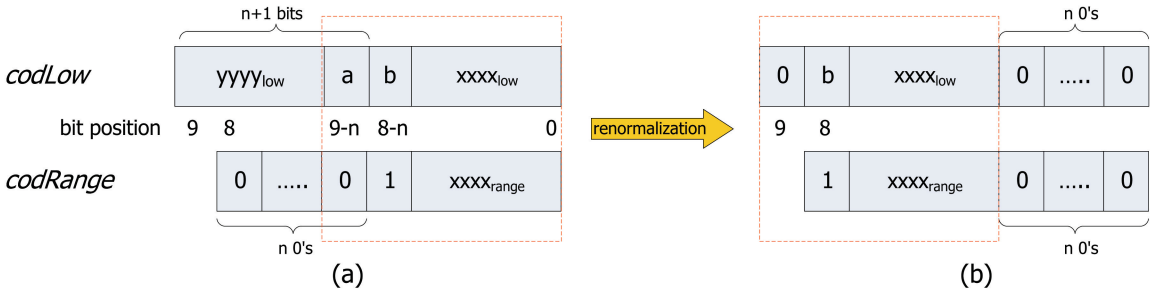


Figure B.2: Scenario 2: Not all top n bits of *codILow* are 1's.

at each iteration. This would effectively result in $n + 1$ leading zeros if the top bit was not thrown away at each left-shift (assuming an extended $codILow$ register with a size of *10+n* bits). In other words, the top bit of $codILow$, i.e., $codILow[9]$, will be guaranteed to be zero after renormalization in this scenario (Fig. B.2). Because of this zero bit, no overflow can happen when adding up the renormalized $codIow$ and $codIRange$ no matter what values b, $\{xxxx_{low}\}$ and $\{xxxx_{range}\}$ hold at that time and Equ. B.5 will be valid here too. For more detailed information about the renormalization operation refer to section 3.3.

□

B.1.3 Bypass coding

Proof. The coding states remain in valid ranges after encoding a bypass-coded bin. As mentioned in section 3.5, bypass coding divides the range into two equal subdivisions and does a regular coding as if the $valMPS = 0$ and $codIRange_{LPS} = codIRange >> 1$. It is effectively a special case of regular coding (with equiprobable subintervals). Therefore, the proofs of sections B.1.1 and B.1.2 are valid and Equations B.1 to B.3 holds for this special case too.[2] □

[2]To be exact, there is a minor difference between bypass coding and an equiprobable form of regular coding if $codIRange$ is an odd value. As described in section 3.5 and step 5 of Figure 3.8, a

B.1.4 Termination bit coding

Proof. The coding states remain in valid ranges after encoding the termination flag. As mentioned in section 3.6, encode of the termination flag is a special case of regular coding. It is as if a fixed $codIRange_{LPS} = 2$ is used with the right branch of Fig. 2.16 taken for the MPS path and the left branch taken for the LPS path. As a result, Equations B.1 to B.3 holds for this special case too.

□

B.2 Number of iterations in renormalization

Proof. Maximum number of iterations in the renormalization process is seven. The renormalization process defined by the standard document is an iterative process (Fig. 2.13). Although a hardware implementation that takes a cycle per iteration is not efficient at all, it is essential to come up with proper bounds on minimum and maximum of possible number of iterations because it affects the architecture exploration phase discussed in section 3.3.

The number of iterations, *iter*, depends on the incoming *codIRange* value from the previous coding stage (e.g., regular coding stage). $codIRange \geq 256$ holds before start of coding process because the renormalization process for the previous bin guarantees

post-processing is required to adjust the interval size resulting from equiprobable regular coding to the original $codIRange_{pre-bypass}$ value. This adjustment either increases or decreases *codIRange* by 1 for the MPS and LPS scenarios respectively. As the post-renormalization coding states have to satisfy Equ. B.5, even a decremented *codIRange* in the LPS path still satisfies Equ. B.5. Here we continue to show incrementing *codIRange* in the MPS case satisfies Equ. B.5 too. If the MPS path is taken, updated *codILow* (before renormalization) remains as the original *codILow*. We know that the renormalization loop iterates only once in bypass coding. It is easy to verify that the renormalized *codILow* becomes smaller if the right branch ($codILow \geq 512$) or the middle branch ($256 \leq codILow < 512$) is taken (because of the reset of $codILow[9]$ or $codILow[8]$ and the fact that the left-shift brings in a 0 bit). Because *codILow* is decreased and the end effect of the adjustment of *codIRange* restores it to its original value, Equ. B.5 still holds when the right or the middle branch is taken. If the left branch is taken ($codILow < 256$), we will have $codILow < 512$ after renormalization. Even if *codIRange* was originally at its maximum size of 511, we will have $codILow + codIRange < 1023$ and Equ. B.5 still holds.

it. Also, $codIRange$ is initialized to 510 before encode of the first bin. Referring to Fig. 2.12 shows that $codIRange$ is either set to $codIRange_{LPS}$ or $codIRange_{MPS}$ depending on the polarity of encoded bin and its associated context. $codIRange_{LPS}$ is retrieved through the multiplier table where it can be noted that

$$2 \leq codIRange_{LPS} \leq 240 \tag{B.6}$$

holds (Table 9-33 of [3]). It is not difficult to observe that there exist cases when $codIRange$ is already more than or equal to 256 after regular coding and no renormalization iteration will be needed. Such a case happens when the MPS path is taken for a bin and $codIRange_{LPS}$ coming from the Multiplier table is small. Then the MPS interval calculated through

$$codIRange_{MPS} = codIRange - codIRange_{LPS} \tag{B.7}$$

will result in

$$codIRange_{MPS} \geq 256. \tag{B.8}$$

The standard does not allow $codIRange$ be reduced to a range size of zero because rescale of the range will not be possible through renormalization anymore. This is guaranteed by defining $p_{min} = 0.01875$ in Equ. 2.3 and preventing the next probability state p_{LPS} from reaching zero (Fig. 2.11). A common mistake (like the work of Osorio, et al. [10]) is to assume $codIRange$ can be reduced to 1 after coding which will require the renormalization loop to iterate 8 times for reaching $codIRange \geq 256$. After update of the coding states the minimum value that $codIRange$ can reach will be 2 as shown below.

If the MPS path is taken for the current bin, $codIRange = codIRange_{MPS}$ will

be at least 16 per Equ. B.7, Equ. B.8 and $max(codIRange_{LPS}) = 240$ per Equ. B.6.[3] If the LPS path is taken, $codIRange = codIRange_{LPS}$ will hold which its minimum will be 2 as $min(codIRange_{LPS}) = 2$ per Equ. B.6. As a result, not more than seven iterations in the renormalization process are needed to rescale $codIRange$ to a value with minimum of 256.

□

B.3 First generated bit at each slice

Proof. The first generated bit in any slice is always a zero bit.
As section 4.3.2 discussed, the first generated bit from encode of every slice is ignored because it is known to be always zero. The proof for this fact is given here.

At the beginning of each slice, the coding states of the arithmetic coder is initialized to $codILow = 0$ and $codIRange = 510$. If the first encoded bin within the slice is regular-coded, the coding states can be constrained as below based on the flowchart of Figure 2.12.

$$Initialization: \begin{cases} codILow = 0 \\ codIRange = 510 \end{cases} \tag{B.9}$$

$$2 \leq codIRange_{LPS} \leq 240 \qquad \text{based on Table 9-33 of [3]} \tag{B.10}$$

$$270 \leq codIRange_{MPS} \leq 508 \qquad \text{by Equ. B.9 , B.10} \tag{B.11}$$

Let us consider the below equations for update of the coding states according to Fig. 2.12.[4]

[3] *min* and *max* operators used here return the minimum and maximum of their arguments.
[4] The subscript *pre-renorm* indicates that the updated coding states are not renormalized yet.

$$codILow_{pre-renorm} = \begin{cases} codILow & \text{if } bin == valMPS \\ codILow + codIRange_{MPS} & \text{if } bin \; != valMPS \end{cases} \quad \text{(B.12)}$$

$$codIRange_{pre-renorm} = \begin{cases} codIRange_{MPS} & \text{if } bin == valMPS \\ codIRange_{LPS} & \text{if } bin \; != valMPS \end{cases} \quad \text{(B.13)}$$

Now the followings can be derived when $bin == valMPS$:

$$codILow_{pre-renorm} = 0 \qquad \text{by Equ. B.9, B.12} \quad \text{(B.14)}$$

$$270 \leq codIRange_{pre-renorm} \leq 508 \qquad \text{by Equ. B.11, B.13.} \quad \text{(B.15)}$$

Because of the above value of $codILow_{pre-renorm}$ and the range of $codIRange_{pre-renorm}$, the renormalization process of Fig. 2.13 will generate a single output bit of zero value. Since this is the first generated bit and it is a zero bit, the case is proved when $bin == valMPS$.

For $bin \; ! = valMPS$, the followings can be derived:

$$270 \leq codILow_{pre-renorm} \leq 508 \qquad \text{by Equ. B.11, B.12} \quad \text{(B.16)}$$

$$2 \leq codIRange_{pre-renorm} \leq 240 \qquad \text{by Equ. B.10, B.13.} \quad \text{(B.17)}$$

Because $codILow_{pre-renorm} > 256$ and $codIRange_{pre-renorm} < 256$, the renormalization process of Fig. 2.13 generates one outstanding bit which is potentially followed by up to six bits of zero or outstanding type (any combination of zero and outstanding bit might happen). Using a semi-Verilog notation[5], the generated bit sequence can be shown as $\{1+, \{r\{1'g\}\}\}$ where $r \in \{0, 1, \ldots, 5, 6\}$ and $g \in \{1+, 0\}$. Now the following two scenarios are possible.

- *Case 1- At least one of the r generated bits after the initial outstanding bit*

[5] *1+* is considered a single bit value as *0* and *1*.

is a zero bit: By assuming r_1 is the number of outstanding bits (excluding the initial bit) generated before the the first zero bit, then the sequence of generated bits can be shown as $\{\ 1+,\ \{r_1\{1+\}\}, 0,\ \{r_2\{1'g\}\}\ \}$ where $r_2 = r - r_1 - 1$. The first zero bit will resolve the preceding pending outstanding bits as $\{\ 0,\ \{(r_1+1)\{1\}\},\ \{r_2\{1'g\}\}\ \}$. While potential resolve of the other outstanding bits is possible if they are followed by any zero bit, the very first generated bit is resolved to 0. In summary, if there is any 0 bit generated in the renormalization of the first bin (regardless of its location in the bit sequence), the first outstanding bit will be resolved to 0 and the case is proved.

- *Case 2- All generated bits associated with renormalization of the first bin are outstanding bits*: The bit sequence can be shown as $\{1+,\ \{r\{1+\}\}\}$. Then the resolve of the first bit needs to wait for the generation of the first non-outstanding bit associated with encode of the following bins. There could be more outstanding bits generated from encode of future bins but as long as the first non-outstanding bit is a zero bit, it will resolve the first outstanding bit to zero as shown in the discussion of *Case 1* above. To prove the remaining case, it is enough to show that the first generated non-outstanding bit after a sequence of outstanding bits can never be a 1 bit.[6]

  Sometimes it is more helpful to show the interval associated with arithmetic coding with its low and high points[7] instead of its low point and its range as shown in Fig. B.3. Per Equ. B.9, the upper limit of the subinterval associated with the LPS area (which is the focus of the discussion here) follows $high_{pre-renorm} \leq 510$. Each iteration of the renormalization loop updates the low and range of the sub-interval which can be equivalently represented with

[6]Note that such a non-outstanding bit with a value of 1 can only be generated at the first iteration of the renormalization associated with encode of a bin (section 3.3.1).

[7]Shown by variables *low* and *high*.

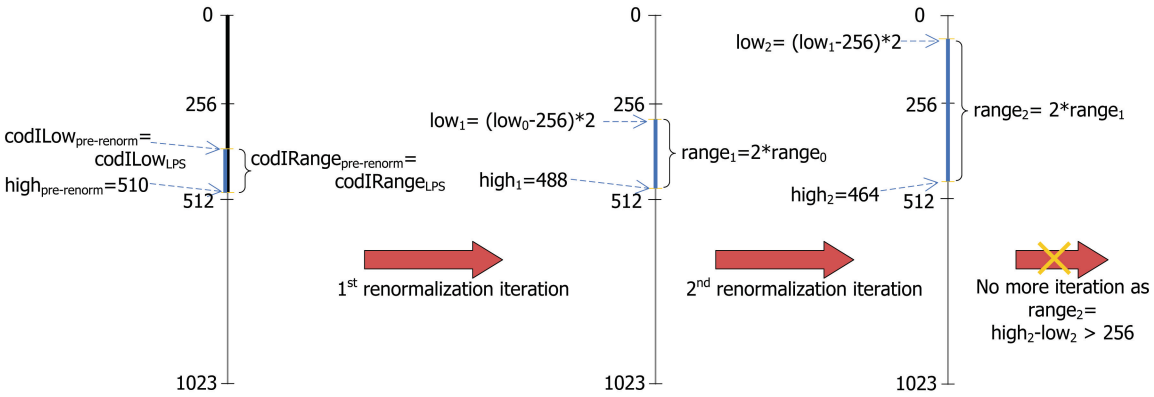


Figure B.3: A sample encode of the first bin of a slice taking the LPS path and generating two outstanding bits.

update of the low and high points of the subinterval. Since such a first non-outstanding bit with value of 1 could only be generated after a sequence of outstanding bits, only renormalization through the middle branch of Fig. 2.13 is considered. The update of low and high points will happen based on the following relations.[8]

$$low_{n+1} = (low_n - 256) * 2$$
$$high_{n+1} = (high_n - 256) * 2$$

Fig. B.3 shows low and high points of the subinterval after two iterations of the renormalization. The LPS subinterval was chosen in a fashion to allow two iterations generating two outstanding bits. No matter what exact value the low point carries, the high point decreases though the interval size is doubled at each iteration. This means that no matter how many outstanding bits are generated, $high < 510$ will be valid.

[8] n indicates the iteration number.

Because update of the coding states at encode of the next bin divides the interval into two subinterval, the high point of either of the two subintervals of LPS and MPS will be still lower than 510. With similar argument, the low point will be lass than 510 too, i.e., $codILow_{pre-renorm} < 510$. This means the right branch of Fig. 2.13 will never be taken for the first non-outstanding bit and a 1 bit can never be generated.[9] As a result, the first non-outstanding bit must be a zero bit. This 0 bit resolves the first outstanding bit to zero and finishes the proof for this case too.

□

B.4 Limits on the longest outstanding bits sequence

As discussed in sections 4.2 and 3.4.2, a long sequence of outstanding bits can cause overflow in the intermediate buffer. Such scenarios need to be handled by stalling the front-end pipes and preventing the new internally resolved bits ($ResolvedBits_{reg}$ in Fig. 4.5) from being appended to the intermediate buffer (Acc_{reg}). The stall frequency depends on the frequency of long sequence of outstanding bits, size of such sequences and size of the intermediate buffer, $2w$.[10] The first two factors are discussed in section 4.2. Here, size of the longest sequence of outstanding bits that is guaranteed to be handled without stall is of our interest. This worst-case scenario is not a typical case. But combined with the statistics of outstanding bits, it can be helpful in making decision about the right size of the intermediate buffer.

[9] To generate a non-outstanding one bit, $codILow_{pre-renorm} \geq 512$ must be valid before renormalization. This could happen by having $codILow \geq 512$ before encode of the current bin so the subinterval for sure falls into the upper half of the range. The other possibility is that the interval spans the midpoint, 512, meaning $codILow < 512$ and $high > 512$. Now if the LPS subinterval is selected in a way that the subinterval is fully in the upper half ($codILow_{pre-renorm} \geq 512$), then the generated bit will be a one bit.

[10] The intermediate buffer size is twice of the FIFO width, w.

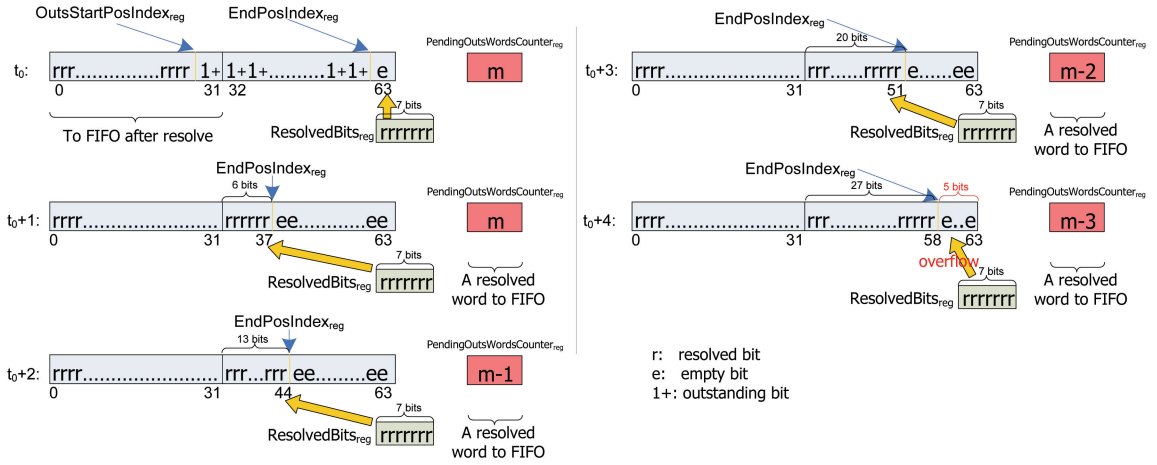


Figure B.4: Maximum outstanding bits tolerated stall-free for $w = 32$ bits.

B.4.1 Longest stall-free sequence for FIFO width of 32

In the worst-case scenario, the longest sequence of outstanding bits that can be handled without overflow is 96 bits for a FIFO with of 32 bits.

When the second word of the intermediate buffer, $Acc_{reg}[32 : 63]$ is fully filled up with outstanding bits, the area is tracked by $OutsWordsCounter_{reg}$ instead and the word is freed for incoming generated bits. The worst-case scenario happens when an outstanding bits area starts from bit 31 of the intermediate buffer, goes beyond several words, continues up to $Acc_{reg}[62]$ bit and is finally resolved by the longest possible generated bits of 7 bits. Now the first word is fully resolved and sent out at the same cycle the resolve has happened. At every next cycle, a word of resolved outstanding bits will be sent out to the FIFO decrementing $OutsWordsCounter_{reg}$ till it reaches zero. But at the same time, lots of newly generated bits (with the maximum possible size of 7) could arrive at the buffer at every cycle. Because of the limited bandwidth of access to the output FIFO (one word per cycle), this can quickly fills up the whole intermediate buffer leading to an overflow.

Such scenario is shown in Figure B.4 with the initial state shown at t_0. The state

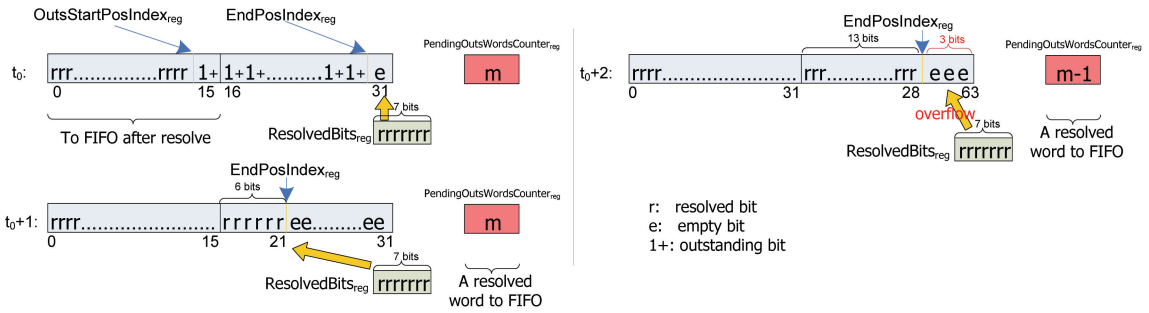


Figure B.5: Maximum outstanding bits tolerated stall-free for $w = 16$ bits.

of the buffer at the next four cycles is shown from $t_0 + 1$ to $t_0 + 4$ which finally leads to an overflow. This shows that if more than $m = 2$ words were accumulated in $OutsWordsCounter_{reg}$, the overflow can not be prevented. Considering the other 32 bits already existing in $Acc_{reg}[31 : 62]$ at t_0, this results in 96 bits for a FIFO width of 32, i.e. $w = 32$.

□

B.4.2 Longest stall-free sequence for FIFO width of 16

In the worst-case scenario, the longest sequence of outstanding bits that can be handled without overflow is 32 bits for a FIFO with of 16 bits.

For a FIFO width of 16 bits similar argument can show that the longest sequence of outstanding bits that can be handled without stall can not be more than two words, i.e., 32 bits since the word size is 16 bits. Figure B.5 shows the intermediate buffer states at the three cycles leading to an overflow.

□

Appendix C

Binarization commands format

This appendix provides the description of syntax element binarization commands defined in section 4.1.3. It is assumed that a minimal hint for derivation of context index is provided by a higher-level entity (e.g. upstream block or processor) when information about the neighboring macroblocks is needed. This hint, *cxtIdxHint*, is passed to the binarization command as part of the parameters field, *Params*. The format[1] of binarization commands in the new scheme are described below.[2] Similar to the notation of Figure 2.5, the left, top and current macroblocks are indicated by A, B, and C respectively.

***UnaryTUnaryCoder binarizer*:**

ElementValue: { {8{1'0}}, value_in[7:0] }

Params: { {2{1'0}}, TruncateMode, MaxValue[7:0]}

This binarization command is internally translated to a call like the following one to interface to the corresponding binarizer block:

*UnaryTUnaryCoder(value_in /*value to binarize*/,*

[1] See Figure 5.3 for bit allocation of binarization command.

[2] To simplify the demonstration of calculation of *CtxIdxHint* here, only the main condition that covers the range of possible context index increments is shown and other conditions (like neighboring block not being available) are skipped. Refer to [3] and [12] for full set of conditions.

*DIRECT_MODE /*ctxIdxOffset: 511 to signal special direct mode*/,*
X /$CtxIdxIncr_1$: Not to be regular coded; don't care*/,*
X /$CtxIdxIncr_2$: Not to be regular coded; don't care*/,*
X /$CtxIdxIncr_3$: Not to be regular coded; don't care*/,*
*TruncateMode /*TruncateMode*/,*
*MaxValue /*MaxValue: Cutoff value for truncated unary binarization*/).*

The command allows unary or truncated-unary binarization of a value up to a width of 8 bits[3] to be sent to the hardware block and the resultant binary bits be added directly to the output stream. The generic interface for *UnaryTUnaryCoder* block is used but with the special mode of *DIRECT_MODE* which carries value of 511, an unused context index value. This notifies the block to pass the unary coded bits directly to the output stream through *Direct data pass-through* interface rather than arithmetically coding them. This command is especially useful for elements of slice header which are supposed to be unary coded. Another important case is *pcm_byte* data which is part of *macroblock layer* when macroblock is of *I_PCM* type (section 7.3.5 of [3]).

Binarization of several syntax elements will drive the unary coder block by feeding the block "internally" rather than through a unary/truncated-unary binarization command. The data provided to the block follows the below format:

*UnaryTUnaryCoder(value[7:0] /*value to binarize*/,*
*ctxIdxOffset[8:0] /*Context index offset for the first bin*/,*
*$CtxIdxIncr_1[3:0]$ /*Context index increment for the second bin*/,*
*$CtxIdxIncr_2[3:0]$ /*Context index increment for the third bin*/,*
*$CtxIdxIncr_3[3:0]$ /*Context index increment for rest of the bins*/,*
*TruncateMode /*Whether the value is to be truncated-unary coded*/,*
*MaxValue[7:0] /*Cutoff value for truncated unary binarization*/).*

[3]The widest data to be unary/truncated-unary coded is the 8-bit *pcm_byte* data.

If *DIRECT_MODE* is not active, all generated bits will be arithmetically encoded in the regular mode with *ctxIdxOffset* carrying a context index in [0,399] range within the context table. This context index will be used for encode of the first generated bin resulted from unary/truncated-unary coding of *value*. $CtxIdxIncr_1$ is the context index increment added to *ctxIdxOffset* for deriving the context index used with regular coding of the second generated bit. Similarly $CtxIdxIncr_2$ will be used for deriving the context index for the third generated bin (if any). And finally, $CtxIdxIncr_3$ is the context index increment to be used for calculating the context index for encode of the forth and subsequent generated bins (if any).[4]

***ExpGolombK binarizer*:**

ElementValue: { {1'0}, value_in[14:0][5] }

Params: { {5{1'0}}, direct_mode, k[4:0][6] }

This binarizer command allows k-th degree exponential Golomb encode of an input value. *direct_mode* signals the binarizer to directly send the generated bits to the output stream without arithmetic encode of them. Otherwise, the generated bins will be sent for arithmetic encode in the bypass mode like for *coeff_abs_level_minus1* and *mvd_l0_l1* syntax elements. The parameters of the binarization command will drive the block like the following function call which is similarly used by other binarization blocks:

*ExpGolombK(value[14:0] /*value to binarize*/,*

*k[3:0] /*k = 0 ExpG of degree k */,*

[4]Because the largest context increment used is 9 (for *coeff_abs_level_minus1*), a 4-bit value for context index increments is sufficient.

[5]The largest value to be encoded using *ExpGolombK* binarizer is the 15-bit transform coefficient value. In general, the input value is signed (though not for the cases of *MVD* and *coeff_abs_level_minus1*) to allow signed exponential Golomb encode of some elements of the slice header as marked with *se(v)* in [3].

[6]The highest degree of *ExpGolombK* used for binarization of syntax elements is $k = 3$ which is used for *coeff_abs_level_minus1*. A wider range is allowed here in case extended degree is needed for encode of header information elements.

*direct_mode/*if not active, the bits will be bypass coded*/).*

***BinCoder interface (bin pass-through path)*:**

ElementValue: { {15{1'0}}, bin_value }

Params: { {1'0}, coding_mode, ctxIdx[8:0] }

Rather than an actual binarizer block with explicit hardware implementation, this is an interface that directly passes the input bin to the arithmetic coder at the following stage along with the context index *ctxIdx* and the mode for arithmetic coding. That is why it is also called *bin pass-through* path. The format of use of this interface by other syntax element binarizers is like the following function call:

*BinCoder(bin_value /*value to encode*/,*
*ctxIdx /*context index for regular coded mode*/,*
*bypass_mode /*1 if bypass mode; 0 if regular mode*/).*

***mb_type binarizer*:**

ElementValue: { {10{1'0}}, mb_type[5:0] }

Params: { {6{1'0}}, SliceType[2:0], ctxIdxHint[1:0] }

ctxIdxHint $\in \{0, 1, 2\} =$ (SliceType == `I_SLICE`) ?
(((`MB_Type(A)` != `I_4x4` && `MB_Type(A)` != `I_8x8`) ? 1 : 0) +
((`MB_Type(B)` != `I_4x4` && `MB_Type(B)` != `I_8x8`) ? 1 : 0)) :
(((`MB_Type(A)` != `B_Direct_16x16`) ? 1 : 0) +
((`MB_Type(B)` != `B_Direct_16x16`) ? 1 : 0)).[7]

The implementation of this binarizer uses the provided *ctxIdxHint* for calculating the context index of the first generated bin when the image slice is of I or B types. The other generated bins are associated with fixed context indices.

[7] `MB_Type(X)` returns the type of macroblock pointed by *X*. Also, note that the reference software [12] has recently added the comparison against I_8x8. Compare version JM8.2 vs. JM9.6.

Generation of the bin string and sequence of context indices can be simplified through using a ROM lookup table as several conditions are tested. All generated bins are sent to the bin FIFO using the *bin pass-through* path.

***sub_mb_type binarizer*:**

ElementValue: { {11{1'0}}, sub_mb_type[4:0] }

Params: { {8{1'0}}, SliceType[2:0] }

The implementation of this binarizer uses fixed assignments of context indices to generated bins. The generated bin string depends on the macroblock sub-type, *sub_mb_type*, and the slice type of encoded picture, *SliceType*. Similar to the case before, a ROM lookup mechanism can make generation of the bin string and context indices easier. All generated bins are sent to the bin FIFO using the *bin pass-through* path.

***mb_skip_flag binarizer*:**

ElementValue: { {15{1'0}}, skip_flag }

Params: { {8{1'0}}, IsB_SliceType, ctxIdxHint[1:0] }

ctxIdxHint $\in \{0,1,2\} =$

$$((\texttt{MB\_SkipFlag(A)} == 0)\,?\,1:0) + ((\texttt{MB\_SkipFlag(B)} == 0)\,?\,1:0).$$[8]

The implementation of this binarizer uses the provided *ctxIdxHint* and the slice type of the picture for calculating the context index of the only generated bin. The generated bin is sent to the bin FIFO using the *bin pass-through* path.

***mb_field_decoding_flag binarizer*:**

ElementValue: { {15{1'0}}, field_flag }

Params: { {9{1'0}}, ctxIdxHint[1:0] }

[8] `MB_SkipFlag(X)` returns the skip flag status of the macroblock pointed by *X*. Also, note that there is a change from version JM8.2 to JM9.6 of the reference software [12] for calculation of the context index increment. Now a skip flag of 0 for the neighboring macroblocks will result in increment of the context index.

ctxIdxHint $\in \{0,1,2\} = ((\texttt{MB\_FieldFlag(A)} != 1)\,?\,1:0) + ((\texttt{MB\_FieldFlag(B)} != 1)\,?\,1:0)$.[9]

The implementation of this binarizer will use the provided *ctxIdxHint* above for calculating the context index of the only generated bin. The generated bin is sent to the bin FIFO using the *bin pass-through* path.

***mb_qp_delta binarizer*:**

ElementValue: { {10{1'0}}, mb_qp_delta[5:0] [10]}

Params: { {10{1'0}}, ctxIdxHint }

ctxIdxHint $\in \{0,1\}$ = ((`MB_QP_Delta(PreviousMB)` != 0) ? 1 : 0).[11]

The implementation of this binarizer only needs to employs the *UnaryTUnary coder* to binarize the delta value of the current macroblock's quantization parameter. The provided *ctxIdxHint* is passed to the unary coder as the first parameter to be used as *ctxIdxIncr* for the first generated bin. The second generated bin will use context increment of 2 and the rest will use context increment of 3. The parameters provided to the unary coder will like the function call below:

*UnaryTUnaryCoder(mb_qp_delta/*value to binarize*/,*
*60/*ctxIdxOffset*/,*
*ctxIdxHint /*CtxIdxIncr*$_1 \in \{0,1\}$**/,*
*2 /*CtxIdxIncr*$_2$**/,*
*3 /*CtxIdxIncr*$_3$**/,*
*0 /*TruncateMode*/,*
*X /*MaxValue: Don't care as truncate mode is off */).*

***ref_idx_l0/l1 binarizer*:**

ElementValue: { {10{1'0}}, ref_idx_l[5:0] [12]}

Params: { {9{1'0}}, ctxIdxHint[1:0] }

[9] `MB_FieldFlag(X)` returns whether the macroblock pointed by X is coded in field mode or not.
[10] Note *mb_qp_delta* $\in [0, 51]$ based on page 76 of [3].
[11] `MB_QP_Delta(X)` returns the QP delta associated with the macroblock pointed by X.

ctxIdxHint $\in \{0,1,2,3\} = a + 2 * b$ where a and b are derived as below:[13]

if (`MBAFF_mode` && !`MB_FieldFlag(C)` && `MB_FieldFlag(B)`)

a = ((`LeftMB_RefIdx(C)` $>$ 1) ? 1 : 0) ;

else

a = ((`LeftMB_RefIdx(C)` $>$ 0) ? 1 : 0) ;

if (`MBAFF_mode` && !`MB_FieldFlag(C)` && `MB_FieldFlag(A)`)

b = ((`UpMB_RefIdx(C)` $>$ 1) ? 1 : 0) ;

else

b = ((`UpMB_RefIdx(C)` $>$ 0) ? 1 : 0) ;

The implementation of this binarizer only needs to drive the *UnaryTUnary coder* to binarize the reference index of the picture used for motion estimation of the (sub)block. The provided *ctxIdxHint* is passed to the unary coder as the first parameter to be used as *ctxIdxIncr* for the first generated bin. The second generated bin will use context increment of 4 and the rest will use context increment of 5. The parameters provided to the unary coder will be similar to the function call below:

*UnaryTUnaryCoder(num_ref_idx_10_active_minus1/*value to binarize*/,*
*54/*ctxIdxOffset*/,*
*ctxIdxHint /*CtxIdxIncr*$_1 \in \{0,3\}$**/,*
*4 /*CtxIdxIncr*$_2$**/,*
*5 /*CtxIdxIncr*$_3$**/,*
*0 /*TruncateMode*/,*
*X /*MaxValue: Don't care as truncate mode is off */).*

[12]Per page 57 of [3], the maximum index value is $num\_ref\_idx\_l0\_active\_minus1$ for the decoding of frame macroblocks and $2{*}num\_ref\_idx\_l0\_active\_minus1{+}1$ for the decoding of field macroblocks. Because $num\_ref\_idx\_l0\_active\_minus1 \in [0, 31]$, the index value shall be in the range of 0 to 63, inclusive.

[13]`LeftMB_RefIdx(X)` and `UpMB_RefIdx(X)` respectively return the reference index associated with the left and upper blocks of the macroblock pointed by X. These left and upper blocks might be in macroblock X or its neighboring macroblocks.

***intra_chroma_pred_mode binarizer*:**

ElementValue: { {14{1'0}}, intra_chroma_pred_mode[1:0] [14]}

Params: { {9{1'0}}, ctxIdxHint[1:0] }

ctxIdxHint $\in$ {0,1,2} =

((`MB_IntraChromaPredMode(A)` != `Intra_Chroma_DC`) ? 1 : 0) +

((`MB_IntraChromaPredMode(B)` != `Intra_Chroma_DC`) ? 1 : 0). [15]

The implementation of this binarizer only needs to employ the *UnaryTUnary coder* to binarize the value of chroma intra prediction mode which is in [0,3]. The provided *ctxIdxHint* is passed to the unary coder to be used as *ctxIdxIncr* for the first generated bin. The second generated bin will use context increment of 3.[16] The parameters provided to the unary coder will be like the function call below:

*UnaryTUnaryCoder(intra_chroma_pred_mode/*value to binarize*/,*

*60/*ctxIdxOffset*/,*

ctxIdxHint /$CtxIdxIncr_1 \in$ {0,1,2}*/,*

3 /$CtxIdxIncr_2$*/,*

X /$CtxIdxIncr_3$: Not used; don't care!*/,*

*1 /*TruncateMode*/,*

*3 /*MaxValue: As the value is in [0,3] */).*

***intra4x4_pred_mode binarizer*:**

ElementValue: { {12{1'0}}, intra4x4_pred_mode[3:0] }

Params: { {7{1'0}}, prevMB_intra4x4_pred_mode[3:0] }

[14]Note *mb_qp_delta* $\in [0, 51]$ based on page 76 of [3].

[15]`MB_IntraChromaPredMode(X)` returns the chroma intra prediction mode of the macroblock pointed by *X*. Also, note that the explanation given in page 628 of [4] for calculation of context index increment is wrong and does not match [3] and [12].

[16]Page 628 of [4] suggests that binarization of the intra prediction mode for chroma will result in three bins which is a wrong statement! *intra_chroma_pred_mode* takes only integer values between 0 and 3 based on Table 7.13 of [3] and truncated-unary encode of it with a $cMax = 3$ will not result in more than two bins per Table 9-24 of [3].

The implementation of this binarizer first derives *prev_intra4x4_pred_mode_flag* and *rem_intra4x4_pred_mode[2:0]* values based on Equ. 8-24 of [3]. Then simply *prev_intra4x4_pred_mode_flag* is coded with the context index of 68 and in case it does not carry a 1 value, the three bits of *rem_intra4x4_pred_mode[2:0]* will be coded using fixed-length coding with the context index of 69 for the three bins. The fixed-length coding of the three bits will be done inside the block starting from the least significant bit.[17] The generated bin(s) (either one or four) will be sent directly to the *bin FIFO* through the *bin pass-through* path.

***coded_block_pattern binarizer*:**

ElementValue: { {8{1'0}}, $ctxIdxHint_{luma_0}[1:0]$, coded_block_pattern[5:0] }

Params: { {1'0}, $ctxIdxHint_{chroma_1}[1:0]$, $ctxIdxHint_{chroma_0}[1:0]$,
$ctxIdxHint_{luma_3}[1:0]$, $ctxIdxHint_{luma_2}[1:0]$,
$ctxIdxHint_{luma_1}[1:0]$}

$ctxIdxHint_x \in \{0,1,2,3\} = a + 2 * b$.

For each one of the four luma blocks blk_i of the current macroblock, values of a and b are derived as below to calculate the related hint value $ctxIdxHint_{luma_i}$:[18]

```
if ( (IsLeftLumaBlockInC(blk_i)) )
    a = (MB_GetCodedBlock(C, GetLeftBlockIndex(blk_i)) == 0) ? 1 : 0 ;
else
    a = (MB_Type(A) == I_PCM) ? 0 :
        (MB_GetCodedBlock(A, GetLeftBlockIndex(blk_i)) == 0) ? 1 : 0 ;
if ( (IsUpperLumaBlockInC(blk_i)) )
    b = (MB_GetCodedBlock(C, GetUpperBlockIndex(blk_i)) == 0) ? 1 : 0 ;
else
    b = (MB_Type(B) == I_PCM) ? 0 :
        (MB_GetCodedBlock(B, GetUpperBlockIndex(blk_i)) == 0) ? 1 : 0 ;
```

[17] A fixed-length coder block is not part of the binarizer hardware implementation anymore.

For each of the two chroma blocks, the proper context increment value is similarly calculated using section 9.3.3.1.1.4 and Table 7.12 of. Note that the coded block pattern is a 6-bit value which the luma and chroma portion of it are organized based on Equ. 7.22 of [3].

This binarizer simply utilizes the *bin pass-through* path for the four luma bits of the pattern using appropriate context index increment for each luma bit and context offset of 73.[19] The first chroma flag is similarly encoded using the *bin pass-through* path with context offset of 77 and if the flag is non-zero, the second chroma bit is encoded similarly but with context offset of 81.[20]

***CodedBlock_SignificanceMap binarizer*:**

ElementValue: { significance_map[15:0] }

Params: { {6{1'0}}, ctxCategory[2:0], ctxIdxHint[1:0] }

ctxIdxHint $\in \{0,1,2,3\}$ = `coded_block_flag(A,i)` + 2 * `coded_block_flag(B,i)`.[21]

This blocks handles binarization of three syntax elements of *coded_block_flag*, *significant_coeff_flag* and *last_significant_coeff_flag* at once. The significance map of a whole block is passed through *ElementValue* field of the binarization command where each bit represents the significance (nonzero or zero) of the corresponding coefficient in the scanning order.[22] The category which the encoded block belongs to is passed through *ctxCategory* which specifies one of the five

[18]The four luma blocks (of $8 * 8$ size) of a macroblock are indexed from 0 to 3 associated with top-left, top-right, bottom-left and bottom-right blocks. `IsLeftLumaBlockInC(i)` returns whether the block to the left of block i is within the same macroblock or not. `GetLeftBlockIndex(i)` and `GetUpperBlockIndex(i)` respectively return the block index of the block to the left or to the top of block i. `MB_GetCodedBlock(X,i)` returns whether the luma block i of the macroblock X is coded.

[19]All four luma bits use a shared set of context indices from 73 to 76. But each of the chroma bits will use their separate context index range one from 77 to 80 and the other from 81 to 84.

[20]Note that the two chroma bits within the coded block pattern does *not* correspond to individual chroma blocks, u and v blocks. The meaning of these two bits are explained in Table 7.12 of [3].

[21]`coded_block_flag(X,i)` returns whether the block i of macroblock X is coded or skipped. Note that it applies to both luma and chroma block types.

[22]The maximum number of coefficients in an encoded block is 16 based on Table IV of [4].

possible block/context categories (Table IV of [4]). Based on the received category, the binarizer would derive the number of valid bits within the significance map. First, *coded_block_flag* syntax element is derived based on existence of at least one nonzero coefficient in the map. *ctxIdxHint* combined with *ctxCategory* points to one of the twenty context indices in the range of 85 to 104 to be passed directly to *bin FIFO* for encode of *coded_block_flag* syntax element.

Then for every bit of the significance map, the right context increment is derived from one of the 61 possible indices based on the context category *ctxCategory* the coded block belongs to and the index of significant coefficient.[23] The significance bit is directly sent to the *bin FIFO* along the context index. After each significance bit, *last_significant_coeff_flag* is derived based on if the last significant bit is met or not. Each *last_significant_coeff_flag* is directly sent to the *bin FIFO* with a context index similarly derived using the context category and the coefficient position in the scan order.[24] A simple state machine in the binarizer can manage this process. By binarizing the whole significance map for a block through a single binarization command, the communication between the higher level entity and the binarizer is significantly reduced here too.

***coeff_abs_level_minus1 binarizer*:**

ElementValue: { coeff_sign, coeff_abs_level_minus1[14:0][25]}

Params: { {1'0}, ctxCategory[2:0], $ctxIdxHint_{bin1-13}$[3:0], $ctxIdxHint_{bin0}$[2:0] }

These partial context index increment values are calculated by the higher level entity

[23]Adding up *MaxNumCoeff* column of Table IV of [4] for the five categories will result in 66 contexts. Because no *significant_coeff_flag* needs to be encoded for the last coefficient of each category (it is implied by *last_significant_coeff_flag*), the total number of contexts will be reduced to $66-5=61$ which is covered by contexts 105 to 165 for frame coded mode and 277 to 337 for field coded mode. Depending on whether macroblock adaptive field/frame mode, pure field or pure frame modes are active, both or either of the context index ranges are used.

[24]The context range for *last_significant_coeff_flag* covers 61 indices from 166 to 226 for frame coded mode and 338 to 398 for field coded mode.

[25]Based on different subsections of section 8.5 of [3], *coeff_abs_level_minus1* can not be outside

through the following calculations. Here i is the index of the current coefficient in reverse scan order of coefficients as transform coefficients are encoded in reverse scan order.[26]

$$ctxIdxHint_{bin0} \quad \in \{0,1,2,3,4\} = \begin{cases} 4 & \text{if } \texttt{NumLgt1(i)} > 0 \\ \min(3, \texttt{NumT1(i)}) & \text{otherwise.} \end{cases}$$

$$ctxIdxHint_{bin1-13} \in \{5,6,7,8,9\} = 5 + \min(4, \texttt{NumLgt1(i)}).^{27}$$

This blocks handles binarization of a single transform coefficient and its sign. Binarization of *coeff_abs_level_minus1* consists of a 14-bit truncated-unary coded prefix and an exponential Golomb coded suffix with $k = 0$. $ctxIdxHint_{bin0}$ and $ctxIdxHint_{bin1-13}$ specify the partial context index increment for the first and the following 13 bins within the indices range for the context category associated with the type of block being coded. Through Table 9-30 of [3], $ctxCategory$ specifies the indices range to be added to the partial context index increment provided by $ctxIdxHint$. As this table shows, 10 contexts are used for each category because $0 \leq ctxIdxHint \leq 9$. The forth category ($ctxCategory = 3$) is an exception since a Chroma DC block can only carries up to four coefficients so `NumLgt1()` will not go higher than 3 implying a maximum of 8 for $ctxIdxHint_{bin1-13}$ value for this category of block type. As a result, the total number of contexts are 49 which are covered in the indices range of 227 to 275 of the context table.

For binarization of the prefix portion, the unary coder will be driven as the below function call:

$[2^{-15}, 2^{15} - 1]$ even for the widest range like the case of Luma DC. As a result, a 15-bit value and a sign bit will suffice for presentation of all different types of transform coefficients.

[26] Note that both references to *max* function in page 631 of [4] are wrong as they contradict the explanation given in the text. Instead, *min* function should have been used.

[27] `NumT1(i)` returns the number of trailing transform coefficients with a value of equal to one encountered till position i in reverse scan order of the coded block. Similarly, `NumLgt1(i)` returns the number of coefficients having a value greater than one encountered till position i in reverse scan order.

*UnaryTUnaryCoder(max(coeff_abs_level_minus1, 14)/*value to binarize*/,*
*227/*ctxIdxOffset*/,*
$ctxIdxHint_{bin0}$ /$CtxIdxIncr_1 \in \{0,1,2,3,4\}$*/,*
$ctxIdxHint_{bin1-13}$ /$CtxIdxIncr_2 \in \{5,6,7,8,9\}$*/,*
$ctxIdxHint_{bin1-13}$ /$CtxIdxIncr_3$: Used for all later bins too!*/,*
*1 /*TruncateMode*/,*
*14 /*MaxValue: As the prefix value is in [0,14] */).*

If *coeff_abs_level_minus1*$>$ 14/, the suffix portion is binarized by driving the *ExpGolomb(k)* block as below. Note that no context index needs to be provided to *ExpGolomb(k)*as all of its generated bins are to be arithmetically encoded in the bypass mode.

ExpGolombK(coeff_abs_level_minus1$-$14 */*value to binarize*/,*
0 */*k = 0 ExpG of degree k */,*
0 */*direct_mode is off).*

And finally, the sign of the transform coefficient is directly sent to the *bin FIFO* to be encoded in the bypass mode.

***mvd_l0_l1 binarizer*:**

ElementValue: { {2{1'0}}, mvd_l0_l1[13:0][28]}

Params: { {8{1'0}}, direction, ctxIdxHint[1:0] }

ctxIdxHint $\in$ {0,1,2} = (`NeighboringPartionsMVD_Sum(C, direction)` $\leq$ 2) ? 0 :
(`NeighboringPartionsMVD_Sum(C, direction)` $\leq$ 32) ? 1 :
2 ;

NeighboringPartionsMVD_Sum is defined as below for the current macroblock or submacroblock partition C:

[28]Based on section A.3.1 of [3], the horizontal range of motion vector difference is independent of the active H.264 profile and is within [-2048, 2047.75] range. Considering the quarter sample accuracy of the motion vector difference, $16 * 1024$ possibilities exist which can be represented by a 14-bit value. The vertical direction of motion vector difference is profile-dependent but does not exceed [-512, 511.75] so a 14-bit value still suffice.

`NeighboringPartionsMVD_Sum(C,direction)` =

|`LeftPartitionMVD(C,direction)`| +

|`TopPartitionMVD(C,direction)`| ;[29]

This blocks handles binarization of a motion vector difference value in vertical or horizontal direction for one of the macroblock or submacroblock partitions of the current macroblock. Binarization of *mvd_l0_l1* consists of a 9-bit truncated-unary coded prefix and an exponential Golomb coded suffix with $k = 3$ forming a *UEG3* with a cutoff value of 9 for the truncated-unary prefix. *ctxIdxHint* specifies the context index increment for the first of the prefix. Each of the second, third and the fourth bins will be coded with a fixed context index increment (3 to 5 respectively). The fifth to ninth bins of the prefix are all coded with a context index increment of 6. As a result a total of 7 context index increments from 0 to 6 are used for each *MVD* component (in horizontal or vertical direction). Indices of 40 to 46 are used for horizontal direction while 47 to 53 used for vertical direction (Table 9-11 of [3]).

l0 and *l1* indicate whether the reference picture belongs to *L0* list (for P/SP and B slices) or to L1 list (for B slices). Because no distinction between *mvd_l0* and *mvd_l1* syntax elements needs to be made at binarization, *mvd_l0_l1* is used for representing both syntax elements for binarization purposes.

For binarization of the prefix portion, the first bin is directly sent for encoding through the *pass-through* path as below:

*BinCoder((mvd_l0_l1 != 0) ? 1 : 0 /*value to encode*/,*
*((direction == 0) ? 40 : 47) + ctxIdxHint /*ctxIdx*/,*
*0 /*regular-bypass mode: to be regular coded*/).*

[29] `LeftPartitionMVD(p, d)` and `UpPartitionMVD(p, d)` respectively return the MVD component at direction d for the partition to the left or to the top of the partition p.

The first bin is encoded separately because *UnaryTUnaryCoder* handles up to four different context indices (through a context offset and three context increments) which is not enough to handle the five context indices for *mvd_l0_l1* (one index for the first, second, third and the fourth bins and another index for the fifth to ninth bins) unless its interface is extended. The rest of the prefix bins are sent to the unary coder as below if *mvd_l0_l1* is non-zero:

*UnaryTUnaryCoder(|mvd_l0_l1| − 1 /*value to binarize*/,*
*((direction == 0) ? 40 : 47)+3 /*ctxIdxOffset*/,*
*1 /*CtxIdxIncr$_1$ = 4*/,*
*2 /*CtxIdxIncr$_2$ = 5*/,*
*3 /*CtxIdxIncr$_3$ = 6*/,*
*1 /*TruncateMode*/,*
*8 /*MaxValue: As cutoff is 9 but first bin is already coded*/).*

The suffix portion is binarized when $|mvd\_l0\_l1| \geq 9$ by driving *ExpGolomb(k)* block as below. Note that the suffix bins will be encoded in the bypass mode.

*ExpGolombK(|mvd_l0_l1| − 9 /*value to binarize*/,*
*3 /*k = 3 ExpG of degree k */,*
*0 /*direct_mode is off).*

And finally, if the motion vector difference is non-zero, its sign is directly sent to the *bin FIFO* to be encoded in the bypass mode as below:

*BinCoder((mvd_l0_l1 ≺ 0)? 1 : 0 /*value to encode*/,*
*X /*ctxIdx: Don't care value as the bin is bypass coded*/,*
*1 /*regular/bypass mode: to be bypass coded*/).*

***Direct data pass-through interface*:**

ElementValue: { value_in[15:0] }

Params: { ValueWidth[3:0] }

This interface allows direct append of some raw data directly to the output

stream. Its main function is to allow insertion of data like some header elements that do not require arithmetic coding (e.g., requiring simple fixed-length coding) to the output stream to streamline process of retrieving the output stream no matter if the data is arithmetically coded or not. Another important use of this interface is for *pcm_byte* data which are unary coded using the binarizer and do not require arithmetic coding stage. Since the *bin FIFO* might still have some pending bins belonging to previous syntax elements, the insertion of bits passed to this interface will not happen until the *bin FIFO* is empty and the pipelines of the following arithmetic coding and bit generation stages are cleared up.

Of course, if the data does not require use of binarization or arithmetic coding blocks, the driver of binarizer block might opt to directly write to the output bitstream memory location rather than using this interface and retrieving these data mixed with arithmetically coded data through the output FIFO of CABAC. The input data *value_in* could be up to a width of 16 bits and its size is signalled through *ValueWidth* field.