



NTNU – Trondheim
Norwegian University of
Science and Technology

Design and analysis of a High Efficiency Video Coding(H.265) compliant Context-Adaptive Binary Arithmetic Coding Architecture

Lars Erik Songe Paulsen

04/08/2017

Master Thesis

Department of Electronics and Telecommunications
Norwegian University of Science and Technology

Supervisor 1: Professor Kjetil Svarstad
Supervisor 2: Milica Orlandic

Preface

After the introduction of the new High Efficiency Video Coding(H.265), the standard has been subject to significant academic research. A major evolution in this revision of the video coding standard is the upgraded entropy coding scheme. Because of the serial nature of this algorithm, designing an efficient implementation is vital for high throughput encoding. This formed the motivation for this project, where a HEVC CABAC encoder is designed and analyzed. This Master's thesis was written for readers with knowledge of both software and hardware design, preferably with prior knowledge of video coding standards. It was completed in cooperation with the Department of Electronic Systems at the Norwegian University of Science and Technology, during the autumn semester of 2017.

Trondheim, 04/08/2017

Lars Erik Songe Paulsen

Acknowledgment

The amount of progress achieved in this project would not have been possible without the countless consultations I have had with Milica Orlandic. Her input and knowledge of the HEVC standard has been vital to my understanding of the subject. For this I am very grateful. I would also like to thank Kjetil Svarstad for his invaluable guidance he has given me, regarding both hardware theory and the Hardware Description Languages.

L.E.S.P.

Summary and Conclusion

The Context-Adaptive Binary Arithmetic Coding(CABAC) used in High Efficiency Video Coding(HEVC/H.265) is a near optimal entropy coding method. As a consequence of this coding efficiency, CABAC implementation is a complicated and highly serialized algorithm. With the CABAC becoming a bottleneck in encoder and decoder performance, a major innovation has taken place in the binarization scheme of the transform-coefficient level values. HEVC introduces an adaptive binarization scheme that allows more data to be encoded using a high throughput bypass mode. This adaptive binarization scheme utilizes three different coding methods, Truncated Unary(TrU), k-th order Truncated Rice(TRk) and k-th order Exp-Golomb(EGk). By exploiting the properties of the video-coding data structure, as well as the properties each of these coding methods hold, this binarization scheme is able to achieve a near optimal code.

Thorough analysis of the binarization scheme has been performed, with a main focus on finding an efficient hardware implementation. A major challenge was finding an efficient way of coding the remaining absolute transform-coefficient level(ALRem). ALRem is coded using an truncation of TRk and EGk, with an adaptive level(k). A finite state machine approach was found, that proved to be a very efficient at coding the absolute remaining level. This approach was implemented in hardware.

The Context Index Calculator, that form an integral part of the HEVC CABAC system was not implemented. When this module is designed, it is proposed to combine the Binarizer and Context Index Calculator. This is due to the large amount of shared data dependencies.

A simplified version of an actual Context-Adaptive Binary Arithmetic Coding encoder architecture is implemented. It performs CABAC encoding as specified in by the HEVC standard, but is limited to the encoding of a subset of the transform-coefficient level data. Verifying the correctness of this hardware encoder required the development of a software model. This software encoder was expanded to also include a decoder, which allowed for additional functional verification.

Because of the inconsistent throughput of the encoder modules, an asynchronous fifo was developed to simplify data flow, and improve performance. Due to the unfinished state of both the binarizer and context index calculator, the completed system was not implemented.

Contents

Preface	2
Acknowledgment	3
Summary and Conclusion	4
1 Introduction	10
1.1 Background	10
1.2 Objectives	11
1.3 Limitations	11
1.4 Approach	12
1.5 Features and Contributions	12
1.6 Structure of the Report	13
2 Entropy and Arithmetic Coding	14
2.1 Shannon Entropy	15
2.2 Entropy for binary strings	16
2.3 Arithmetic Coding	17
3 HEVC System and Data Structure	19
3.1 Syntax Elements	20
3.2 CABAC Encoding	21
3.3 CABAC Decoding	22
4 Binarization	23
4.1 Binarization Processes	23
4.2 Unary, Truncated Unary (TrU) Fixed-Length (FL)	23
4.3 Truncated Rice (TRk)	24
4.4 Exp-Golomb	24
4.5 Scan Direction	25
4.6 Transform-coefficient level data	26
4.7 Coding of First Non Zero Element Coordinate	27
4.8 Coding of Absolute Level	28
4.9 Coding of Sign	33
4.10 Grouping of Bins	33
4.11 Complete Example	34
5 Context Modeling	35
5.1 sig_coeff_flag	35
5.2 sig_coeff_greater1_flag and sig_coeff_greater2_flag	35
5.3 Probability model	36
5.4 Initialization	36
6 HEVC CABAC Algorithm	37
6.1 Overview	37
6.2 Variable Initialization	37
6.3 Encoding a Decision	38
6.4 Renormalization	39

6.5	Writing to Bitstream	40
6.6	WriteBits	40
6.7	read_bits	40
6.8	Decoding a Decision	41
7	Software Model	42
7.1	Binarization and Context Index Calculation	43
7.2	Encoder and Decoder	43
7.3	Interfacing With TestBenches	44
8	Hardware Implementation	45
8.1	Modules	45
8.2	Parameters	45
8.3	Byte Packing and Alignment	45
8.4	Binarizer Implementation	46
8.4.1	Syntax Frames	46
8.4.2	last_sig_coeff	46
8.4.3	sig_coeff_flag	47
8.4.4	coeff_abs_level_greater1_flag	48
8.4.5	coeff_abs_level_greater2_flag	49
8.4.6	coeff_abs_level_remaining	50
8.4.7	coeff_abs_level_sign_flag	53
8.5	Context Index Calculator	54
8.6	CABAC Encoder	55
8.6.1	Interface	57
8.6.2	Parameters	58
8.6.3	Transition Tables	58
8.6.4	Context Table	58
8.6.5	Context Handling	59
8.6.6	BitsOutstanding Loop	60
8.6.7	Termination	60
8.6.8	Register precision	60
8.6.9	Utilization	61
8.6.10	Frequency	62
8.6.11	Performance	63
8.7	Fifo Buffer	64
9	Results and Discussion	65
9.1	Binarizer and Context Index Calculator	65
9.2	CABAC Hardware Encoder	65
9.3	Achieving Correctness	65
9.4	Future Work	65

List of Figures

1	Simple Unary coding based compression of byte data.	14
2	Encoding of a single symbol.	17
3	Encoding of a single symbol.	17
4	Sequential Encoding of Three Symbols	18
5	AVC vs HEVC Picture Partitioning	19
6	Syntax Element Bitrate Distribution	20
7	CABAC encoding process for a 4×4 Transform Block.	21
8	CABAC decoding process for a 4×4 Transform Block.	22
9	Scan direction for Diagonal scan.	25
10	Scan direction for vertical and horizontal scans.	25
11	Residual data in the Transform Block to be Binarized	26
12	Binarization of last_sig_coeff_x/y_prefix in 4×4 Transform Blocks.	27
13	Binarization of Z using the SIG, ALG1, ALG2 and ALRem.	29
14	Binarization of Z using the SIG, ALG1 and ALRem.	30
15	Binarization of Z using the SIG and ALRem.	31
16	ALRem code length for each value of the adaptive variable k	32
17	Binarization of SIGN.	33
18	Complete Example Binarization of a 4×4 Transform Block.	34
19	SIG context assignment for 4×4 Transform Blocks.[9]	35
20	FSM based probability estimation in CABAC.	36
21	Encoding of a binary decision.	38
22	Renormalization during encoding of a binary decision.	39
23	PutBit procedure	40
24	Decoding of a binary decision.	41
25	Software model user interface.	42
26	Hardware and Software Interfacing	44
27	Planned Hardware Modules	45
28	Binarization of sig_coeff_flag	47
29	Binarization of coeff_abs_level_greater1_flag	48
30	Binarization of coeff_abs_level_greater2_flag	49
31	Binarization of coeff_abs_level_sign_flag	53
32	Hardware Encoder State Machine Diagram	55
33	Asynchronous fifo.	64

List of Tables

1	Entropy of Symbols	15
2	Entropy of Binary Strings	16
3	Unary, Truncated Unary and Fixed Length code.	23
4	Truncated Rice(TRk) Code	24
5	Reverse order Exp-Golomb(EGk) code.	24
6	Selection of scan direction in 4×4 Transform Blocks.	25
7	Syntax Elements detailed in this project.	26
8	Initial values for encoding and decoding.	37
9	Alternative representation of ALRem	50
10	Alternative representation of ALRem(calculation)	51
11	Example Binarization of ALRem for $Z = 8$ and $k = 0$	51
12	Example Binarization of ALRem for $Z = 8$ and $k = 2$	51
13	Hardware Encoder State Sequence	56
14	Hardware Encoder State Sequence With Redundancies.	56
15	Pipelined Hardware Encoder State Sequence.	56
16	Encoder ports.	57
17	Hardware Encoder Parameters.	58
18	Syntax Elements supported in the current context table.	59
19	Hardware Encoder Context Table Structure.	59
20	Range variable precision requirements.	60
21	Synthesis results	61
22	Synthesis results	61
23	CABAC Encoder maximum frequency for a few select parameters.	62
24	Performance for bypass encoding.	63
25	Performance for regular encoding.	63

Appendixes:

Appendix A: VHDL Binarizer

Appendix B: VHDL Encoder

Appendix C: Verilog FIFO

Appendix D: C# Encoder

Appendix E: C# Decoder

Appendix F: System Setup

Appendix G: Context Table

Appendix H: Alternative Binarization of coeff_abs_level_remaining

Appendix I: Software Encoder and Decoder demonstration

1 Introduction

1.1 Background

Due to the ever increasing demand for higher quality video content, such as 4k streaming and Virtual Reality 3D video. The Joint Collaborative Team on Video Coding(JCT-VC) set out to improve upon the previous coding standard, H.264/MPEG-4 Advanced Video Coding(AVC). The result of which is the H265/MPEG-H High Efficiency Video Coding(HEVC) standard. The preliminary performance goal for HEVC was a 50% reduction in bit rate compared to AVC at the same subjectively perceived video quality.[5].

In practical terms, HEVC can be viewed as an extension of the concepts utilized in AVC. Although the concepts are very similar, many improvements and optimizations have been explored since the AVC standard was first completed in 2003. Introducing a new standard also allowed for a ground-up redesign of the data structures, that in many ways, set the baseline for potential performance.

HEVC seems to have lived up to its performance goal, as documented by Netflix in its large-scale study on video codecs published in 2016.[7] Using one of the leading open-source HEVC encoders, x265, and comparing it with the leading open-source AVC encoders, x264, as well as the VP9 reference encoder, libvpx. Netflix showed with their advanced video multimethod assessment fusion video quality measurement tool, that x265 offered bit rate savings ranging from 35.4% to 53.3% compared to x264, and from 17.8% to 21.8% when compared to libvpx, at the identical delivered video quality. Even still, 4¹/₂ years after the standard was ratified, adoption rate is still slow. Despite the impressive performance displayed by HEVC, the competition in the royalty free and open-source VP9 has shown to be quite capable. Forcing content providers to consider if HEVC is worth the cost.

HEVC is designed and documented with a focus on a software implementation. The standard document is meant to be understood alongside the HEVC Test Model(HM) written in C++. This makes designing for hardware challenging. As of 2017, hardware implementations are still sparse, with few commercial implementations available[13]. Publicly available hardware implementations of the HEVC CABAC entropy coding scheme is still absent.

HEVC uses the context-adaptive binary arithmetic coding(CABAC) as its single entropy coding method. While AVC also supported the lower-complexity context-adaptive variable-length coding(CAVLC). HEVC CABAC was redesigned to offer higher throughput than its AVC predecessor, while still maintaining a higher compression ratio. This was achieved, in part, by redesigning the binarization scheme for the transform-coefficient level values. This has allowed for an 8× reduction in context coded bins(regular coded bins).[12] Which in turn allow for more bins to be coded using the higher throughput bypass coding method.

The HEVC CABAC entropy coding scheme represents the state of the art lossless compression technology, and has in turn been the focus of intense academic research. CABAC delivers compression close to the theoretical limit(entropy). Because of the complexity of the algorithm, optimizations is still being researched. With a main focus on the coding of the transform-coefficient residual data, that contribute to the largest portion of the compressed video data.[6]

1.2 Objectives

This project aims to:

- Research the HEVC entropy coding scheme. With a main focus on the changes made to the binarization of the transform-coefficient residual data.
- Implement a HEVC compliant CABAC encoder architecture in VHDL, restricted to these residual coding syntax elements.
- Test and characterize the performance of the implemented CABAC encoder modules.

1.3 Limitations

This master thesis did not benefit from a prior semester project, which would have made the extensive scope of this project more manageable. This could also have given a better overview of the challenges involved in developing a HEVC CABAC module, possibly resulting in a smarter approach.

One of the best resources for understanding the HEVC entropy coding scheme, is the open access article on Entropy coding in HEVC from MIT[11]. It does a very good job of introduction the principles utilized in HEVC, but it does not define the data structure and interface needed to base correct module designs upon. Furthermore, it details many versions of the coding scheme, leaving some ambiguity about the current revision functionality. Complete HEVC CABAC documentation is provided by ITU Telecommunication Standardization Sector, in their Recommendation ITU-T H.265 standard document. This documentation is best understood when used with the accompanying HEVC Test Model, a C++ based software model. The standard document along with the HEVC Test Model does cover everything, but does so in an unapologetically complicated manner. This resulted in a major simplification of the first stage binarization, and an incomplete Context Index Calculator. The remaining stages of the hardware CABAC encoder is designed as best understood from the specification, but with some omitted or unverified features.

1.4 Approach

There exists a large amount of research articles related to the HEVC binarization of the transform-coefficient residual data. These articles are very in depth, often assuming the reader has a very good understanding of the subject. Leading to a steep learning curve. Basic understanding of this advanced adaptive binarization scheme is best gained by the combination of figures and clear descriptions. For this reason, a great deal of effort has been made to document the binarization scheme using detailed figures.

A major challenge in this project was testing and verifying the correctness of the designs. All modules comes paired with a simulation TestBench model, this allowed for simple verification of the signaling and state machines. But for functional verification to be achieved, the correctness of the output data had to be established. The best approach would be to use the HEVC Test model to trace the Binarizer and encoder outputs. But with the complexity of this Test model, now approaching 94,000 lines of code[14], this was deemed too demanding for the limited time constraint. Instead, it was decided to develop an independent software model. This model was expanded to include a CABAC decoder, allowing for functional verification with a higher level of confidence. Input test data was however constrained to randomly generated data, making analysis of the encoder performance difficult.

1.5 Features and Contributions

A major contribution of this project is the in-depth analysis of the binarization of the transform-coefficient level data. As well as the efficient encoding method for the `coeff_abs_level_remaining`. The code provided serves as a reference for solving many technical challenges when implementing a correctly sequenced binarizer.

A working regular/bypass encoding module of a HEVC compliant CABAC architecture is written in VHDL. In addition, a software version of both the encoding and decoding is provided.

1.6 Structure of the Report

1 Introduction

Chapter 1 gives an introduction to the project background, objectives and approach.

2 Entropy and Arithmetic Coding

Chapter 2 gives a short introduction to some of the theoretical aspects behind the CABAC video compression system.

3 HEVC System and Data Structure:

Chapter 3 gives an overview of the High Efficiency Video Coding standard.

4 Binarization:

Chapter 4 documents the adaptive binarization of the transform-coefficient level data.

5 Context Modeling:

Chapter 5 gives an introduction to Context Index Calculation and Context modeling in HEVC CABAC.

6 HEVC CABAC Algorithm:

Chapter 6 documents the CABAC algorithm, as it is specified in the HEVC standard document.

7 Software Model:

Chapter 7 details the software model used to verify the hardware encoder.

8 Hardware Implementation:

Chapter 8 documents the hardware architectures implemented in this project

9 Results and Discussion:

Chapter 9 discusses the results of the project, as well as possible future work.

2 Entropy and Arithmetic Coding

A foundation for intuitive understanding of the principles involved in data compression is sometimes best achieved through practical examples. Consider being given the task of reducing the page count for a normal English text. With the condition that all substantive information is preserved, and that the recipient is able to rebuild the whole original text with only the knowledge of the English language and writing system. A straightforward first approach could be to eliminate all redundant spacings. Then continue by removing any unambiguous vowels. Application of these two simple methods results in a substantial page count reduction, while still containing the same amount of information. The compressed text now carry substantially less redundancy. In other words, the compressed text now contain more information per character when compared to the original text. The key to achieving this compression lies in the knowledge of how the English language works, as this compression system would not work for a completely random text pattern. This important principle is applied when designing a compression scheme for video coding. Where predictability of the data leads to improvements in data compression.

A very simple implementation of an entropy coder Fig. 26 shows how compression of binary information can be achieved. Note how this coder is only efficient at low byte values.

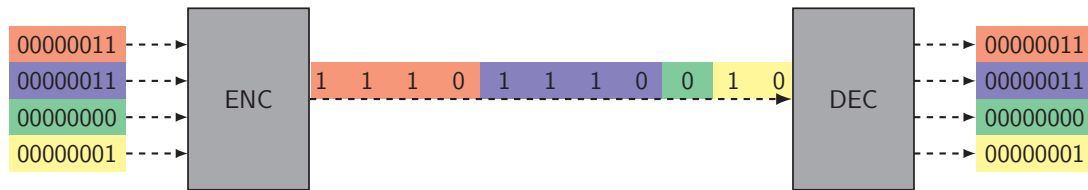


Figure 1: Simple Unary coding based compression of byte data.

This simple unary coder is unable to produce a representation that approaches the theoretical optimal compression ratio(entropy). More advanced entropy coding methods, such as Context-Adaptive Binary Arithmetic Coding, are however when given sufficiently sized information sets able to produce a representation that is arbitrarily close to entropy.

2.1 Shannon Entropy

While generally, entropy is used to refer to the disorder or uncertainty of a system. Shannon entropy provides a mathematical model for the best possible compression of information. When entropy is discussed in the realm of information theory, it is most often referring to Shannon entropy. Where a shannon(sh) is a unit of entropy, which can also be denoted as a bit. One of the most important principles that are applied in complex entropy coders, is the fact that entropy is skewed by the probability for each distinct element in the information set to occur. Generally, $\log_2(n)$ bits are needed to represent a integer variables of n values, given that n is a power of 2. If these variables are equally probable, the entropy is said to be equal to the number of bits. If, however, some variables reliably occur more often in the information set, entropy goes down. This holds even if every possible variable in the information set is present. Understanding this somewhat unintuitive concept requires delving deeper into the theory that Shannon presented. Shannon defined entropy H of a discrete random variable X in the range $\{x_1, \dots, x_n\}$ and with a accompanying probability mass function $P(X)$ as:

$$H(X) = \sum_{i=1}^n P(x_i) I(x_i) = - \sum_{i=1}^n P(x_i) \log_2 P(x_i) \quad (1)$$

Eq. 1 shows how entropy is calculated using \log_2 , resulting in a unit of entropy that can be referred to as bits. If \log_{10} were used the unit of entropy would denote how many decimal symbols that is required to distinguish the variable within the range. Because the number of bits for any binary number is an integer. The actual number of bits for any given entropy is equal to $\lceil H(X) \rceil$.

Table 1 shows how the theoretical entropy of symbols can be calculated if the probability is modeled by observing the Symbol string.

Symbol string	Symbol Probability					H per symbol
	A	B	C	D	E	
AAABCC	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{6}$	0	0	1.459
ABCABC	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	1.585
AAAAAA	$\frac{3}{3}$	0	0	0	0	0
ABCD	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0	2
AABB	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	1
ABCDE	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	2.322

Table 1: Entropy calculated assuming the distribution in the symbol string is representative of the actual probability distribution.

This also illustrates how entropy calculation is relative to your model. Consider the symbol string {AAABCC}. If you could map {AAA} \rightarrow {D} and then instead transmit {DBCC}. This would result in a reduction of entropy. Notice how for the Symbol string {ABCD}, H per symbol is equal to 2. The logical mapping of this symbol string in binary would be {A, B, C, D} \rightarrow {00, 01, 10, 11}. Notice also how for the Symbol string {AAAAAA}, the H is 0. In other words, entropy is zero when the outcome is certain.

2.2 Entropy for binary strings

All digital information is represented in what can be referred to as binary strings. This can be seen as a simplification of the amount of symbols that needs to be modeled with a probability distribution. Table 2 shows how probability can be calculated by the observed bin string. And how the entropy is affected by the observed probability distribution.

Bin string	Probability		H per bin
	1	0	
00001111	$\frac{1}{2}$	$\frac{1}{2}$	1
00110011	$\frac{1}{2}$	$\frac{1}{2}$	1
01010101	$\frac{1}{2}$	$\frac{1}{2}$	1
11111110	$\frac{7}{8}$	$\frac{1}{8}$	0.543
00000011	$\frac{1}{4}$	$\frac{3}{4}$	0.811
11011001	$\frac{5}{8}$	$\frac{3}{8}$	0.954

Table 2: Entropy calculated assuming the distribution in the bin string is representative of the actual probability distribution.

This simple model works on a per bin basis, while it is easily observable that there exists patterns in the bin strings. $\{00001111\}$ could be reduced down to $\{01\}$ by introducing a dictionary that maps $\{0000, 1111\} \rightarrow \{0, 1\}$. This illustrates how patterns in the bin string can be exploited with a better model, and how this requires the model incorporate all elements that will occur. There exists a lot of lossless entropy coding schemes. Each with their own strengths and weaknesses. HEVC's Context Adaptive Binary Arithmetic Coder was developed specifically for video encoding.

2.3 Arithmetic Coding

Arithmetic coding is the core of the Context-Adaptive Binary Arithmetic Coder, this entropy coding scheme works by encoding information by representing it as a sub-interval between 0 and 1. This is a different approach compared to Huffman coding, where the input component symbols are first separated and then replaced with a code. Arithmetic coding is what is known as a statistical coding method, meaning that the coding performance is directly related to the preciseness of the statistical model used. Where the statistical model is the probability distribution for each symbol in the symbol string.

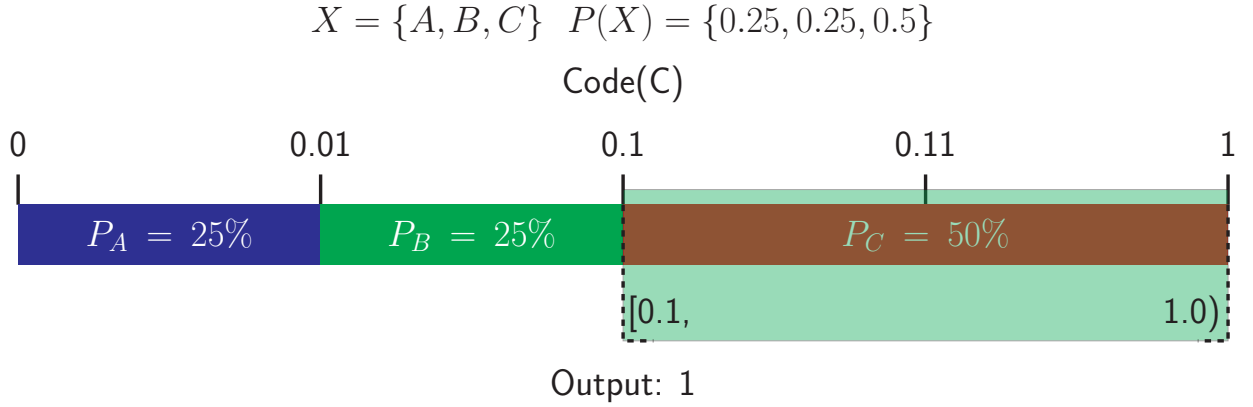


Figure 2: Encoding of a single symbol.

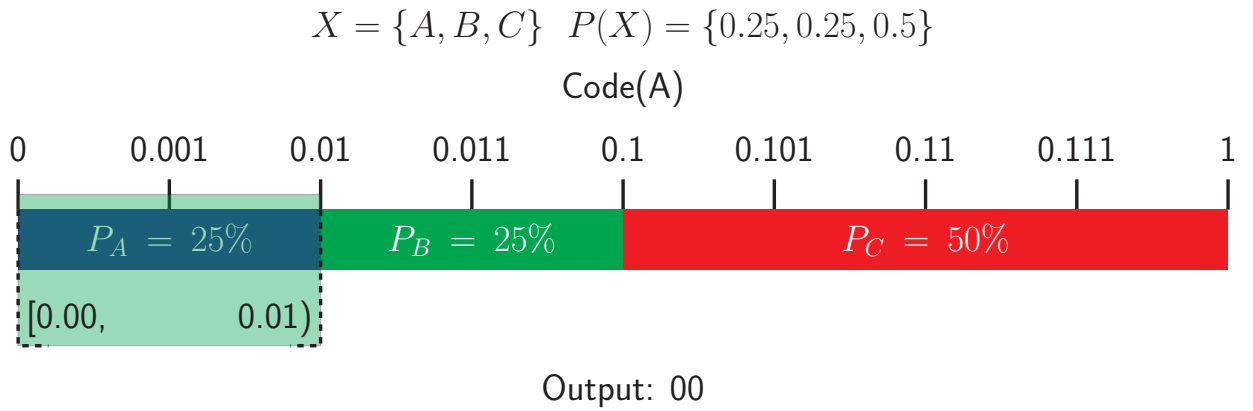


Figure 3: Encoding of a single symbol.

Figures 2 and 3 show how the different symbols are encoded. Where a larger probability percentage directly results in a shorter output code. Note that the output code is equal to the lowest value in the sub interval, and the “0.” symbols can be discarded, as they are inferred present for all arithmetically coded symbol strings.

Coding of multiple sequential symbols is achieved by recursively subdividing into the interval of previously encoded sub intervals.

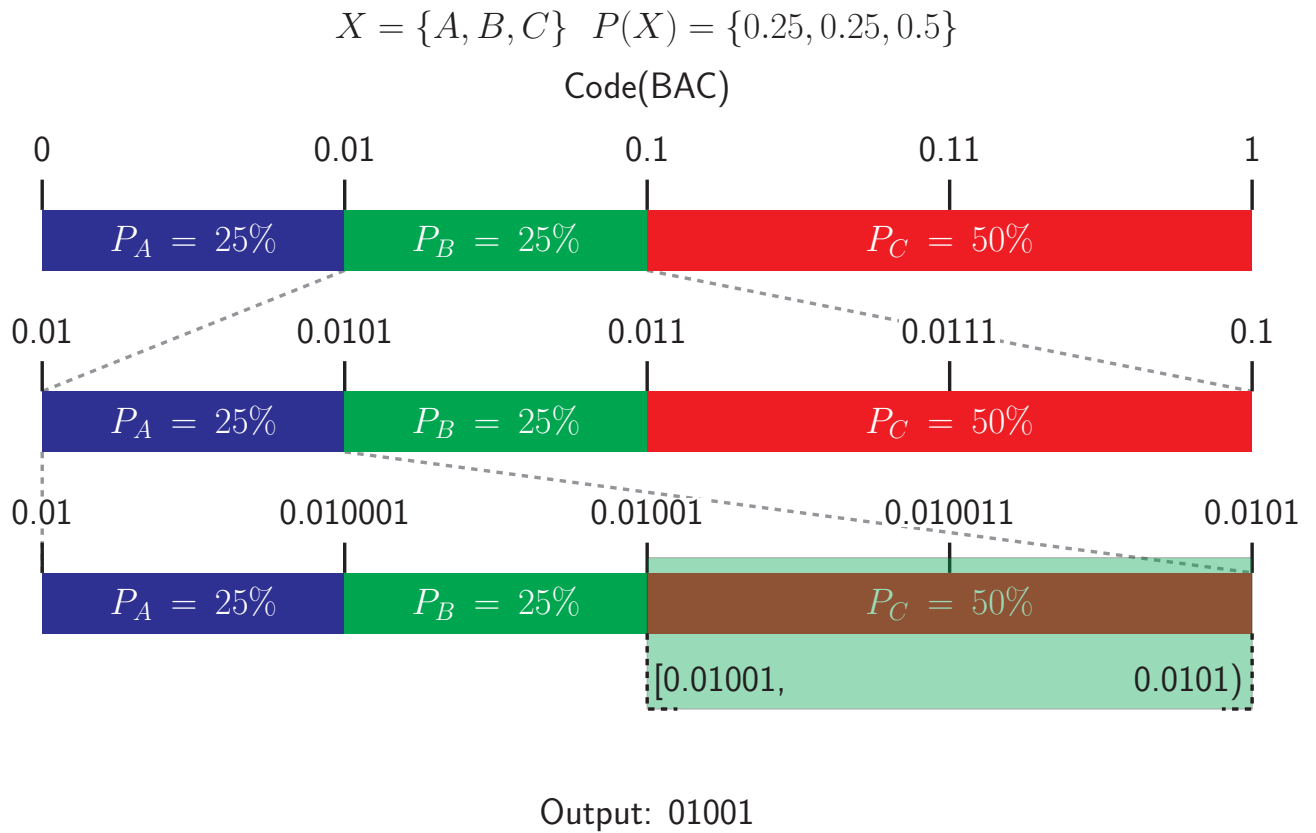


Figure 4: Sequential encoding of three symbols, using a static probability model.

A key element for achieving high compression rates with arithmetic coding is a correct statistical model. HEVC CABAC uses an adaptive context aware probability model. This system works by using a specific probability model(context) for each data element it encodes. Where this probability model is updated for each encoded symbol.

3 HEVC System and Data Structure

One of the bigger changes to HEVC when compared to AVC is the introduction of a more advanced data structure. This change was motivated by the need to efficiently encode higher resolution videos. With this change comes a new set of cryptic acronyms that are added to the standard document vocabulary. While the implemented design covered in this report focuses on the 4×4 Transform Blocks, understanding the parent structures is useful.



Figure 5: HEVC allows larger areas of low complexity to be signaled more efficiently.[4]

Previous digital video coding standards use the Macroblock structure, with a standard of 16×16 samples. For HEVC the macroblocks have been replaced with the Coding tree unit (CTU), allowing for larger block structures (16×16 , 32×32 or 64×64). This innovation is an important part of the coding efficiency improvements HEVC provides. Allowing for large low-complexity areas to be signaled more efficiently. The trade-off being a relative increase in encoding time, but with an added benefit of reduced decoding time.[8]

A naming convention frequently used in HEVC is to use "Unit" suffix when describing the complete pixel data set, i.e. both luma and chroma components, and its accompanying Syntax Elements. The "Block" suffix refers to the distinct luma/chroma components.

Pictures in HEVC are initially divided into CTUs, they are then further divided into Coding Tree Blocks (CTB). One CTB for luma, and two for each chroma component. CTB sizes can be 16×16 , 32×32 or 64×64 . Then these CTBs are further divided into one or more Coding Units (CU). Each division resulting in four smaller regions. This is why the data structure is referred to as a quadtree. CUs are then divided into one or several Transform Units (TU) and prediction units (PU). TUs contain the Transform Blocks of the coefficients for spatial block transforms and quantization which is the focus of this report. TUs and TBs can be 4×4 , 8×8 , 16×16 or 32×32 , but only binarization and coding of TBs of size 4×4 is detailed here.

The data structure of HEVC is well thought out system, that enables many of the performance improvements from the previous standard. The complexity does however make it a challenge to fully comprehend. A thorough understanding of the data structure is required for implementing a complete and correct functional binarizer and context index calculator.

3.1 Syntax Elements

One of the more abstract words heavily utilized in the standard is the Syntax Element. Defined in the standard document as follows:

Syntax element: An element of data represented in the bitstream.

A better definition is not easily construed, but it is possible to look at syntax element as an umbrella term for any property that the data in the bitstream hold. Transform coefficient data for the Luma and Chroma components are known to contribute to the largest amount of data in the video bitstream. This is the main motivation for focusing on these syntax elements.

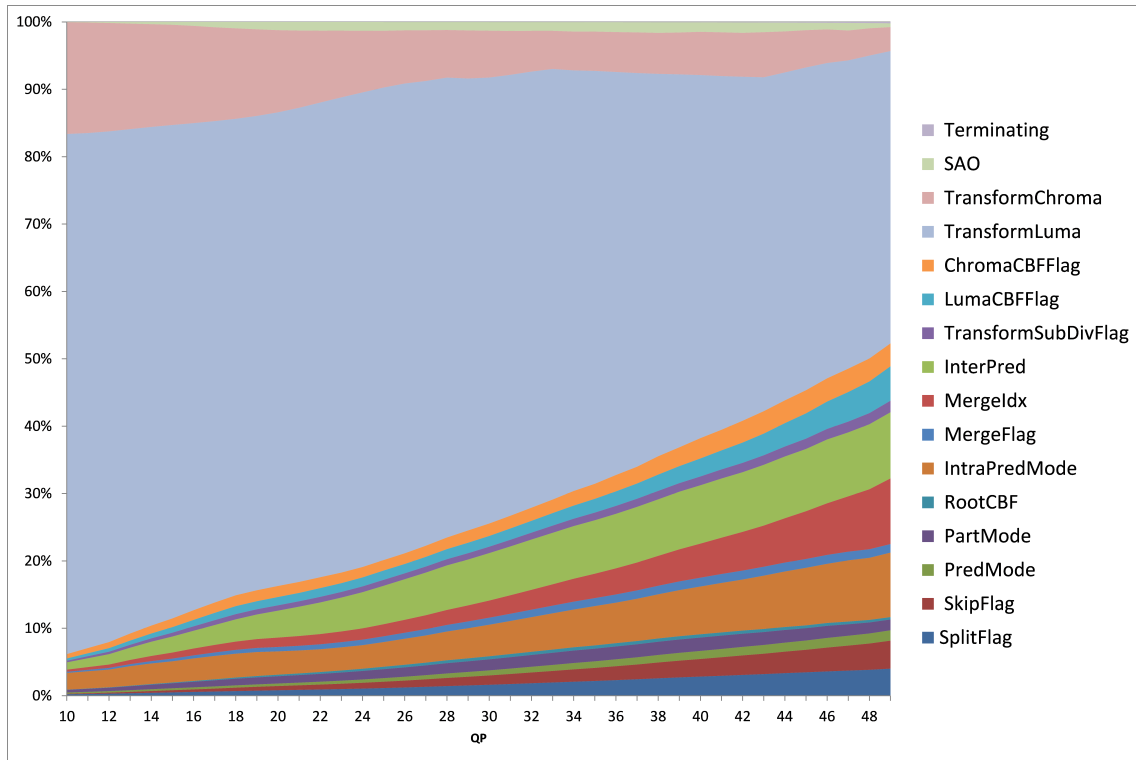


Figure 6: Bitrate distribution of Syntax Elements for varying levels of quantization in HEVC. Where quantization can be viewed as the level of lossy compression. Data shows contribution for encoding of all frame types.[6]

Binarized syntax elements refers to a binarized representation of the properties of the Transform Block data. Where these properties can vary from the location of the last non-zero element in the data set, to the actual binary value of the data.

3.2 CABAC Encoding

Entropy coding in HEVC is based on arithmetic coding, but utilizes a few additional stages designed to improve video coding performance. Input to the HEVC CABAC encoder is syntax elements, and the output is the finished compressed bitstream.

- **Binarization:** The first stage in the encoding is a pre-processing stage that converts the syntax elements into a binary representation more suitable for Binary Arithmetic Coding. Finished binarized syntax elements are commonly referred to as bins.
- **Context Modeling:** Selection of encoding mode for bins is performed. Bypass coding is selected for bins where the distribution is assumed to be uniform, and Regular is selected for bins where this assumption can not be made. Each regular coded bins include an accompanying context index. This context index is calculated based on what syntax element the bins belong to, as well as previously encoded bins. Each context index is a reference to a probability model in a context table.
- **Regular Encoding:** This stage performs Binary Arithmetic Coding of bins using the probability model at the given context index. The probability model at the given context index is updated after encoding of each bin(bit).
- **Bypass Encoding:** Bins with uniformly distributed symbols(equal amounts of ‘1’s and ‘0’s) uses the higher throughput bypass encoding mode. Where the coding algorithm is simplified, due to the exclusion of probability modeling.

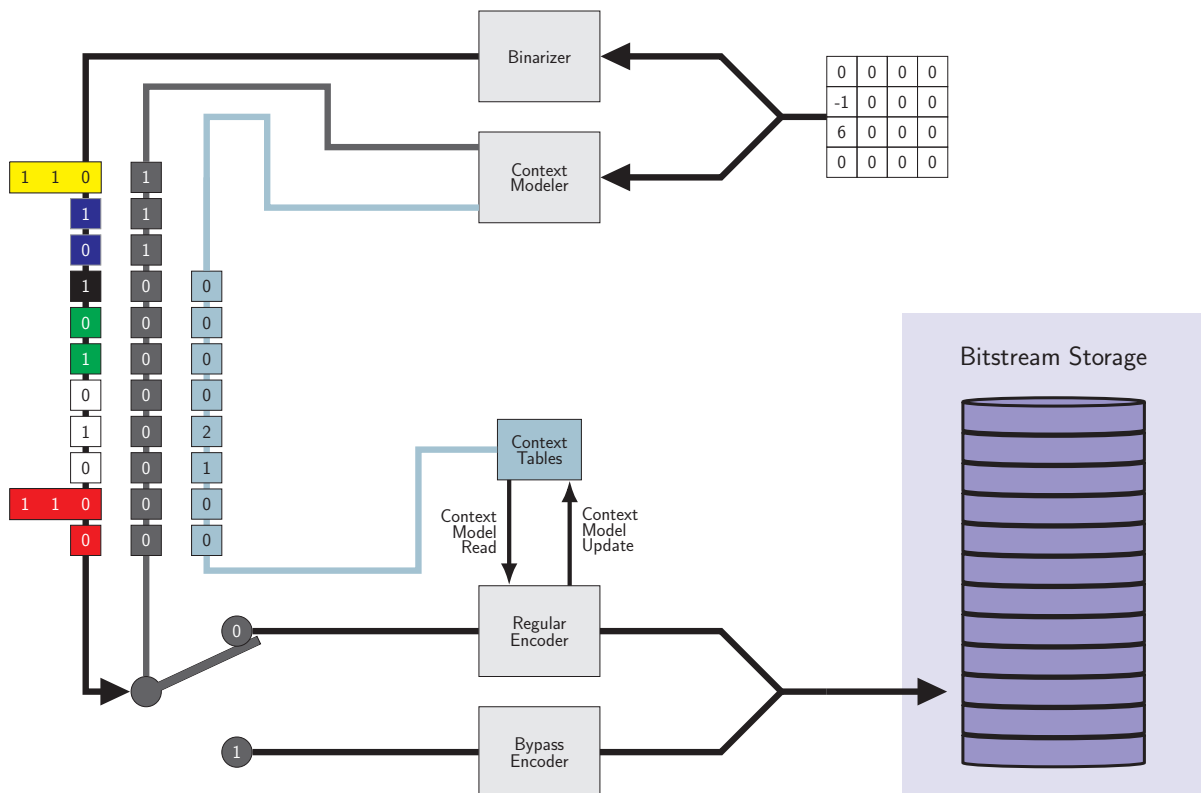


Figure 7: CABAC encoding process for a 4×4 Transform Block.

3.3 CABAC Decoding

The decoding process works by performing the same steps as the encoder, but in reverse order. With the most important difference being that mode selection and context index calculation has to be performed using the previously Decoded and DeBinarized syntax elements. This is enabled by the fact that Binary Arithmetic Coding allows decoding from the front of the bitstream, as well as a binarization scheme designed to facilitate this process. A notable challenge in implementing a CABAC decoder lies in this feedback control system. A simplified software decoder is used for verifying the encoder output, but this report does not focus on covering CABAC decoding in detail.

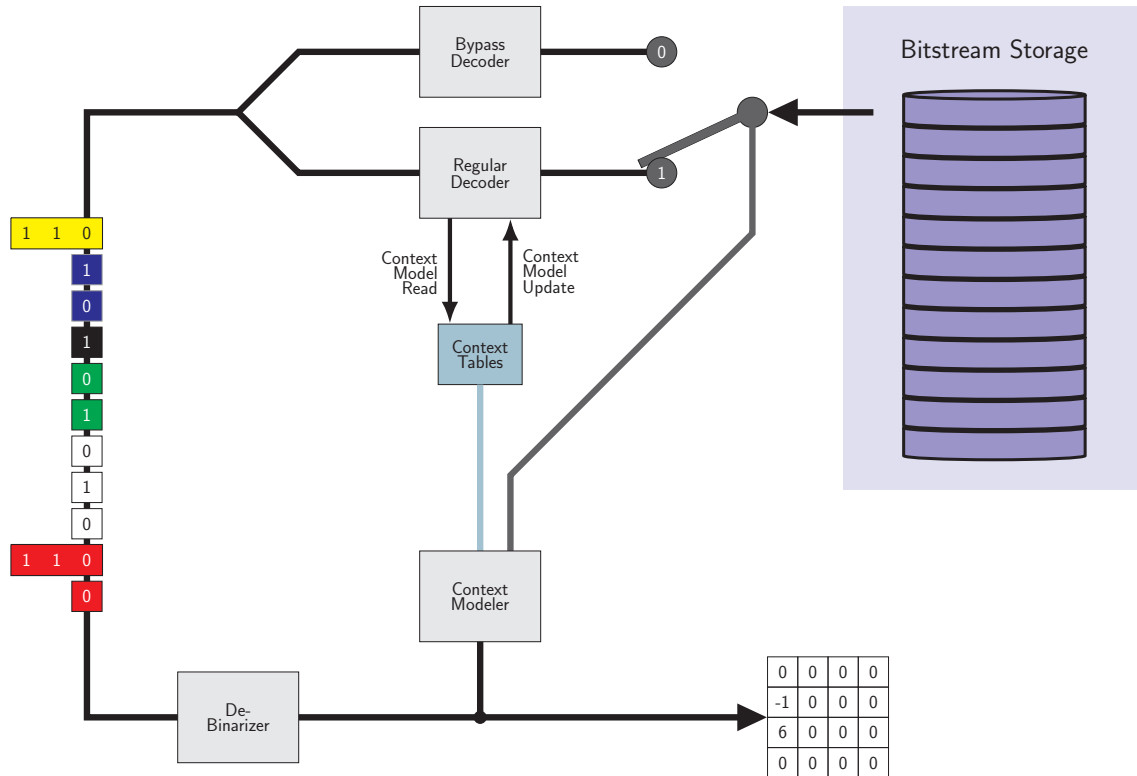


Figure 8: CABAC decoding process for a 4×4 Transform Block.

Note that “CABAC” sometimes refer to only the actual CABAC encoder, and other times to the complete entropy coding system of HEVC, including the Binarizer. Bypass encoding is not really CABAC, since it is not Context-Adaptive, but will often be included in the term CABAC.

4 Binarization

The first stage in the CABAC coder is the binarization of different syntax elements. This Binarizer process aims to truncate the input data, by exploiting certain known properties of the data set. The biggest change in binarization in HEVC compared to the previous standards is the binarization of the syntax elements related to the transform coefficient level values. Some of these binarizations has been made adaptive based on previously binarized transform coefficient level syntax elements. Allowing for more of these syntax elements to be bypass coded. Most notably the `coeff_abs_level_remaining` syntax elements. This section covers the binarization process for a 4×4 Transform Block(TB) transform-coefficient level data, complete with examples.

4.1 Binarization Processes

HEVC uses several different binarization processes. The process chosen is dependent on the syntax element to be binarized, and in some cases the previous binarized syntax element levels. With the goal of choosing the coding method best suited for the properties of the current syntax element to be binarized. The coding methods vary in range, length and growth, as well as the information about the code that needs to be known on both encoder and decoder side. Code-complexity has also been taken into account. Adding too much computational requirements for relative minor coding gains has been avoided. This is especially true for computations where parallelization is difficult. All coding methods used in binarization of the `residual_coding` syntax elements are covered in this chapter.

4.2 Unary, Truncated Unary (TrU) Fixed-Length (FL)

Unary, Truncated Unary and Fixed-length coding are the simplest coding methods used in HEVC. Unary and TrU offers short initial code length with linear growth, with the length inferred from the code. This results in a flexible code suitable for binarizing syntax elements where most values N are small, but which may still function for larger ranges. Fixed-Length requires the length to be known on both the encoder and decoder side. FL is therefore inflexible, but still suitable for certain syntax elements. Syntax elements of length 1(1-bit with value 0 or 1), is said to be FL.

N	Unary(U)	Truncated Unary(TrU)	Fixed-Length(FL)
0	0	0	000
1	10	10	001
2	110	110	010
3	1110	1110	011
4	11110	11110	100
5	111110	111110	101
6	1111110	1111110	110
7	11111110	1111111	111

Table 3: Unary, Truncated Unary and Fixed Length code.

4.3 Truncated Rice (TRk)

HEVC uses the k-th order truncated Rice binarization method. TRk of 0 - 4th order is used in the binarizing process of `coeff_abs_level_remaining`, where the order depends on the previously binarized `coeff_abs_level_remaining`. TRk is similar to Unary and TrU in that the length of the code is inferred from the code itself. The difference is that the code is split into a prefix and a suffix part. The prefix consisting of TrU code, and the suffix is a FL binary representation of the least significant bins. The suffix has the length k, and the prefix is incremented every time the suffix overflows. This allows TRk to dynamically adjust the trade-off between minimum bins length, and range. The largest value in TRk is defined by the `cMax` variable.

N	k				
	0(cMax=3)	1(cMax=7)	2(cMax=15)	3(cMax=31)	4(cMax=63)
0	0	00	000	0000	00000
1	10	01	001	0001	00001
2	110	100	010	0010	00010
3	111	101	011	0011	00011
4	NA	1100	1000	0100	00100
5	NA	1101	1001	0101	00101
6	NA	1110	1010	0110	00110
7	NA	1111	1011	0111	00111

Table 4: Truncated Rice(TRk) code. `cMax` values inferred from the HEVC binarization rules at any given order(k).

4.4 Exp-Golomb

HEVC uses k-th Exp-Golomb coding technique in reverse order. EGk of 1 - 5th order is used in the binarization of `coeff_abs_level_remaining`. This code also uses a unary prefix, as well as a FL suffix that has the length of prefix-length + k. EGk has no inherent range limit, as the length of the suffix is signaled with the prefix. This is an important property as EGk is used as the final coding method of the absolute level.

N	k				
	1	2	3	4	5
0	00	000	0000	00000	000000
1	01	001	0001	00001	000001
2	1000	010	0010	00010	000010
3	1001	011	0011	00011	000011
4	1010	10000	0100	00100	000100
5	1011	10001	0101	00101	000101
6	110000	10010	0110	00110	000110
7	110001	10011	0111	00111	000111
8	110010	10100	100000	01000	001000

Table 5: Reverse order Exp-Golomb(EGk) code.

4.5 Scan Direction

Because of the properties of the residual data in the Transform block, the binarization is affected by the order of which the data is processed. Scan directions chosen are optimal if the coefficient levels are of increasing magnitude, in the order chosen.

Intra Prediction mode	0 to 5	6 to 14	15 to 21	22 to 30	31 to 34
Scan Direction	Diagonal	Vertical	Diagonal	Horizontal	Diagonal

Table 6: Selection of scan direction in 4×4 Transform Blocks.

Scan direction is dependent on the Intra Prediction mode. In most cases the Diagonal scan direction seen in Fig. 19 is used. But in some cases the Vertical and Horizontal scan directions seen in Fig. 10 is employed.

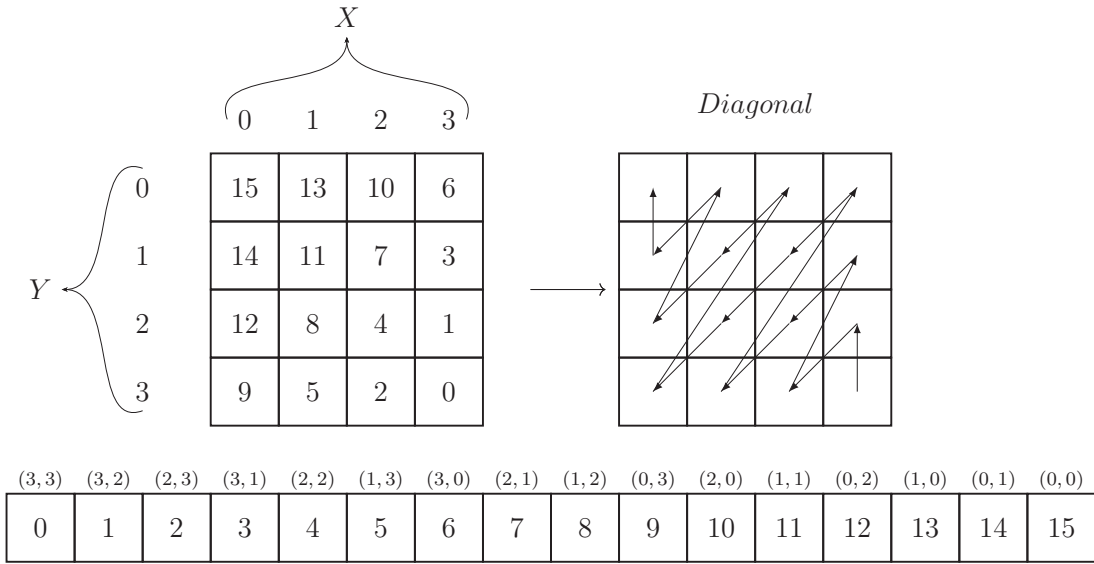


Figure 9: Scan direction for Diagonal scan.

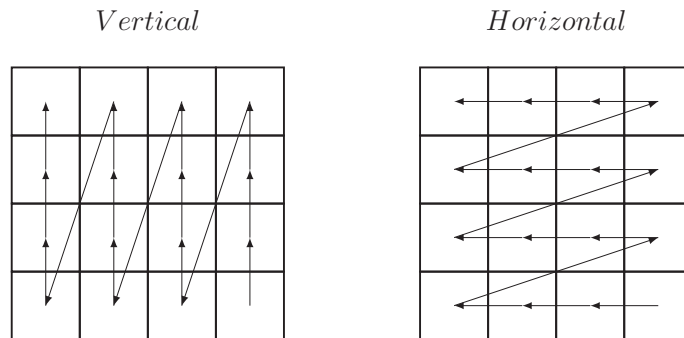


Figure 10: Scan direction for vertical and horizontal scans.

The example binarization covered here uses the Diagonal scan direction. Note that for binarization with any of the other scan directions will result in a different order of the data, and therefore also a different binarization output.

4.6 Transform-coefficient level data

The Syntax elements covered in this report is the Transform-coefficient residual level data. The residual data of a 4×4 TB simply consists of a 4×4 array of signed 16-bit integers. Binary and decimal representation can be seen in Fig. 11, as well as the representation of the data when converted to a one dimensional(16×1) array using the Diagonal Scan Direction. For simplicity, the figures in this chapter utilizes the 16×1 decimal representation for representing the Transform Block, but the finished binarizations are all in binary.

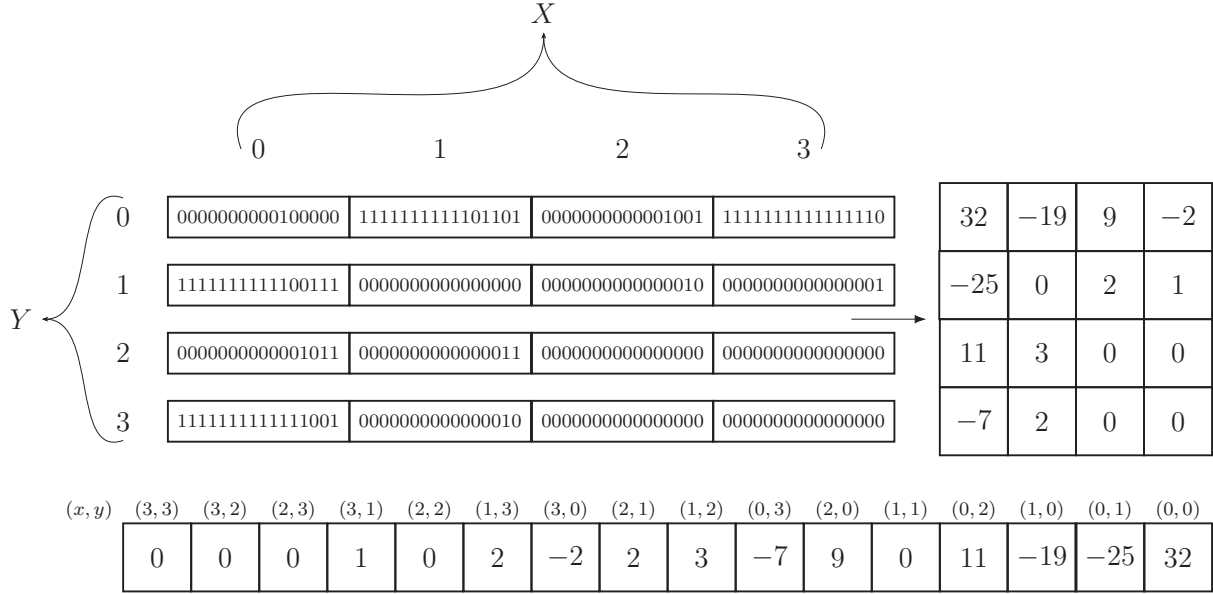


Figure 11: Residual data in the Transform Block to be Binarized

The sample Transform block from Figure 11 is the same for all binarizations covered in this chapter. The Syntax Elements covered in this Section is shown in Table 7

Syntax Element	Abbreviation	Binarization Process	Encoding		
last_sig_coeff_x_prefix	LAST	TR	Regular	Regular	Regular
last_sig_coeff_y_prefix	LAST	TR	Regular	Regular	Regular
sig_coeff_flag	SIG	FL	Regular	Regular	Regular
coeff_abs_level_greater1_flag	ALG1	FL	Regular	Regular	Regular
coeff_abs_level_greater2_flag	ALG2	FL	Regular	Regular	Regular
coeff_abs_level_remaining	ALRem	TrU, TRk and Egk	Bypass	Bypass	Bypass
coeff_sign_flag	SIGN	FL	Bypass	Bypass	Bypass

Table 7: Syntax Elements detailed in this project.

4.7 Coding of First Non Zero Element Coordinate

A major evolution from AVC to HEVC was a change in how the `last_sig_coeff` and `sig_coeff_flag` was binarized. HEVC introduced a scheme where the coordinates of the last non-zero-element(`last_sig_coeff`) in the Transform Block is coded using Truncated Rice. The x- and y-coordinates are split into two distinct syntax elements, `last_sig_coeff_x_prefix` and `last_sig_coeff_y_prefix`. allowing for separate contexts for each coordinate. For TBs larger than 4×4 a Fixed Length suffix is introduced,[9] but this will not be covered further here.

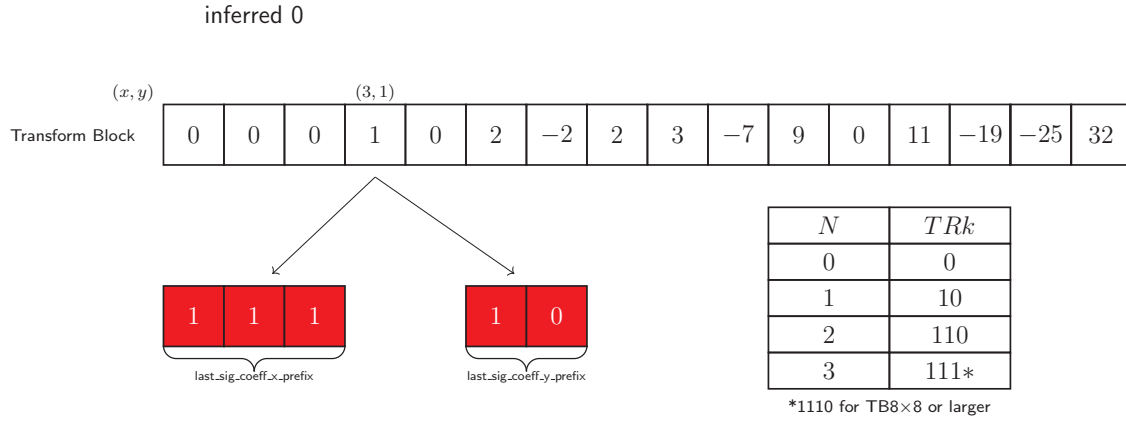


Figure 12: Binarization of `last_sig_coeff_x/y_prefix` in 4×4 Transform Blocks.

Fig. 12 shows an example binarization of `last_sig_coeff_x_prefix` and `last_sig_coeff_y_prefix` using the Diagonal Scan Direction. The notable challenge in implementing this binarization method lies in supporting the three different Scan Directions. For the 4×4 Transform Block, TRk coding is of 0'th order and is therefore identical to Truncated Unary coding.

4.8 Coding of Absolute Level

Absolute levels is defined as the absolute value of the variables contained in the Transform Block. The coding best described as a concatenation of truncated unary(TrK), k -th order truncated Rice(TRk) and $(k+1)$ -th order Exp-Golomb(EGk). Where the TrU coding is implemented using the SIG, ALG1 and ALG2 syntax elements, and the TRk and EGk is implemented using ALRem. Sign data is signaled using the FL coding. Or if the optional sign bit hiding technique is used, signaling of SIGN is potentially skipped.

A useful definition for understanding how absolute level is binarized, is to define this absolute level Z as seen in Equation 2.

$$Z = SIG + ALG1 + ALG2 + ALRem \quad (2)$$

Where SIG, ALG1, and ALG2 all have the value of 0 or 1 when present, or is inferred 0 when not present. There are three thresholding parameters used in the binarization of Z . Two that relates to the coding type, B_0 and B_1 . And one that relates to the TRk and EGk levels, k . B_0 and B_1 is used to separate the three coding methods and k is used to denote the order of the TRk and EGk coding. The binarization process is made adaptive by changing these threshold variables if certain conditions are fulfilled. These conditions are evaluated after coding each level at the current index in the scan. The subblock is then processed by binarizing each Z using the following threshold adaption rules:

Rules:

- Before a subblock is processed, k is set equal to 0 and B_0 is set equal to 2.
- B_1 is defined as $B_1 = 4 \times 2^k + B_0$ and is updated if either k or B_0 is changed.
- B_0 is set equal to 1 after one occurrence of $Z > 1$.
- B_0 is set equal to 0 after eight occurrences of $Z > 0$.
- k is set to $\min(k + 1, 4)$ after each occurrence of $Z > 3 \times 2^k$.

A simplified representation of these rules can be defined as the following:

Simplified Rules:

- Before a subblock is processed, k is set equal to 0.
- ALG2 is only signaled for the first occurrence of $Z > 1$.
- ALG1 is only signaled for the first eight occurrence of $Z > 0$.
- k is set to $\min(k + 1, 4)$ after each occurrence of $Z > 3 \times 2^k$.
- ALG2 is not to be signaled after eight occurrences of ALG1, if all these ALG1s where equal to '0'($Z = 1$).

These simplified rules can be useful for hardware implementation. As it eliminates the somewhat costly update to B_1 .

Figure 13, 14 and 15 shows how the absolute level Z is binarized for a few selected threshold value sets.

	Z	SIG	ALG1	ALG2	ALRem									
TrU $cMax = 3$	0	0												
	1	1	0											
	2	1	1	0										
TRk $k = 0$ $cMax = 4$	3	1	1	1	0									
	4	1	1	1	1	0								
	5	1	1	1	1	1	0							
	6	1	1	1	1	1	1	0						
	7	1	1	1	1	1	1	1	0	0				
	8	1	1	1	1	1	1	1	0	1				
EGk $k + 1 = 1$	9	1	1	1	1	1	1	1	1	0	0	0	0	
	10	1	1	1	1	1	1	1	1	0	0	0	1	
	11	1	1	1	1	1	1	1	1	0	1	0		
	12	1	1	1	1	1	1	1	1	0	1	1		
	13	1	1	1	1	1	1	1	1	1	0	0	0	0
	14	1	1	1	1	1	1	1	1	1	0	0	0	1
	15	1	1	1	1	1	1	1	1	1	0	0	1	0
	16													

Figure 13: Binarization of Z from 0 to 15 using the SIG, ALG1, ALG2 and ALRem syntax elements. ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the initial state for the adaptive binarization scheme.

		Z	SIG	ALG1	ALRem									
TrU	$cMax = 2$	0	0											
		1	1	0										
		2	1	1	0	0								
TRk	$k = 1$ $cMax = 8$	3	1	1	0	1								
		4	1	1	1	0	0							
		5	1	1	1	0	1							
		6	1	1	1	1	0	0						
		7	1	1	1	1	0	1						
		8	1	1	1	1	1	0	0					
		9	1	1	1	1	1	0	1					
		10	1	1	1	1	1	1	0	0	0			
		11	1	1	1	1	1	1	0	0	1			
		12	1	1	1	1	1	1	0	1	0			
EGk	$k + 1 = 2$	13	1	1	1	1	1	1	0	1	1			
		14	1	1	1	1	1	1	1	0	0	0	0	0
		15	1	1	1	1	1	1	1	0	0	0	0	1
		16	1	1	1	1	1	1	1	0	0	0	0	1

Figure 14: Binarization of Z from 0 to 15 using the SIG, ALG1, and ALRem syntax elements (ALG2 absent). ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the state after a first $Z > 1$ has occurred, and before 8 total $Z > 1$ has occurred. $Z > 3$ has also occurred, as shown by the fact that the adaptive variable k now equals 1.

		z	SIG	ALRem									
TrU $cMax = 1$	$B_0 = 1$	0	0										
	$B_0 + 1$	1	1	0	0								
TRk $k = 1$ $cMax = 8$		2	1	0	1								
		3	1	1	0	0							
		4	1	1	0	1							
		5	1	1	1	0	0						
		6	1	1	1	0	1						
		7	1	1	1	1	0	0					
		8	1	1	1	1	0	1					
	$B_1 = 9$												
EGk $k + 1 = 2$	$B_1 + 1$	9	1	1	1	1	1	0	0	0			
		10	1	1	1	1	1	0	0	1			
		11	1	1	1	1	1	0	1	0			
		12	1	1	1	1	1	0	1	1			
		13	1	1	1	1	1	1	0	0	0	0	0
		14	1	1	1	1	1	1	0	0	0	1	
		15	1	1	1	1	1	1	0	0	1	0	
	2^{15}												

Figure 15: Binarization of Z from 0 to 15 using the SIG, ALG1, and ALRem syntax elements (ALG2 inferred 0). ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the state after a total of 8 $Z > 1$ has occurred. $Z > 3$ has also occurred, as shown by the fact that the adaptive variable k now equals 1.

The coding efficiency of the ALRem binarization scheme is noteworthy. Figure 16 shows how it is able to change the effective region of lowest possible code length, by simply changing the adaption variable k . Note that due to the adaption rule ($k = \min(k+1, 4)$), this adaptive binarization scheme is most efficient when processing data of increasing magnitude. This could have been a motivating factor for implementing the multiple scan directions.

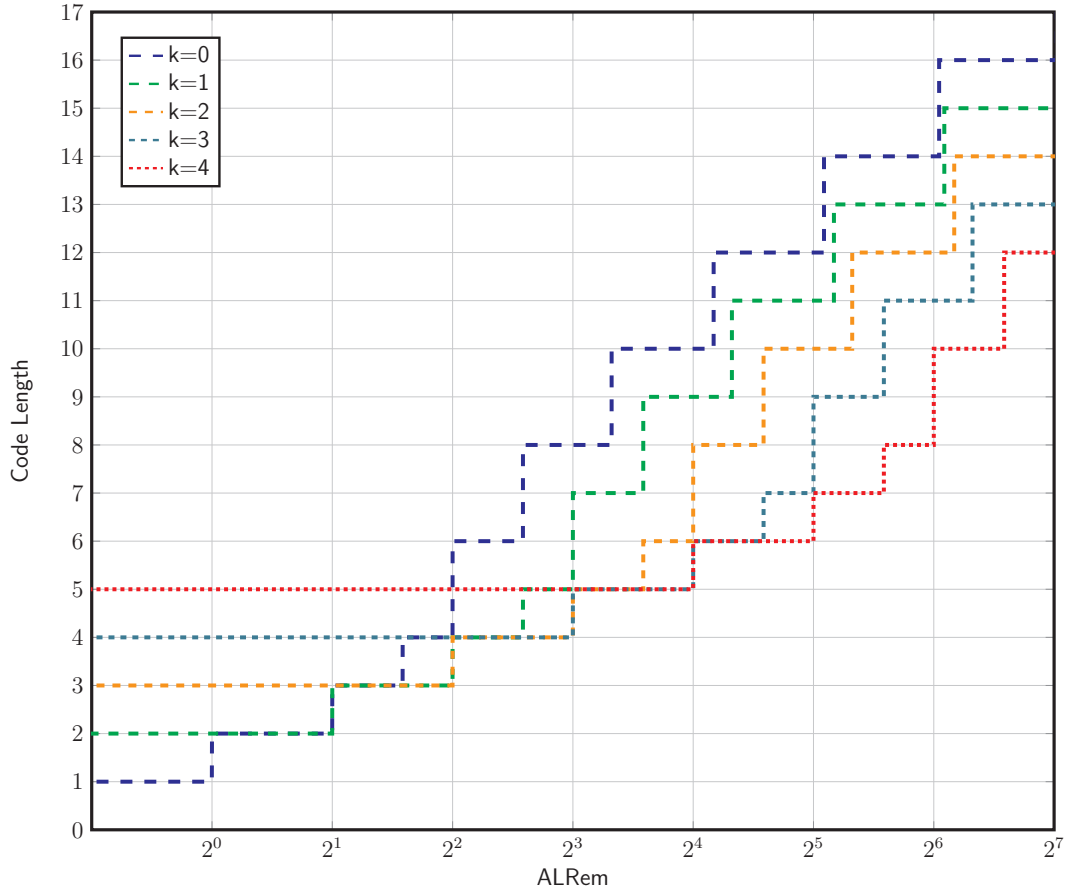


Figure 16: ALRem code length for each value of the adaptive variable k .

4.9 Coding of Sign

Sign is coded using the $\text{SIGN}(\text{coeff_sign_flag})$ that accompanies all the SIG syntax elements when sign bit hiding is not used.

TB	-3	-2	-1	0	1	2	3
SIGN	1	1	1	0	0	0	0

Figure 17: Binarization of SIGN for TB values from -3 to 3. Note that this value is equal to the actual sign bit in the signed integer data type.

Sign bit hiding (SBH) is a technique where the quantizer only signals positive numbers, and instead embeds sign bit into these positive numbers. This is done by using even numbers to represent positive values, and odd numbers to represent negative values. The `sign_data_hiding_flag` indicates if SBH is being used, with the additional condition that there are at least 3 non-zero values in the subblock.

4.10 Grouping of Bins

One advantage in the separation of absolute level Z into SIG, ALG1, ALG2 and ALRem is the improved context modeling accuracy and performance. Another reasoning for splitting these syntax elements is that it allows for grouping of bins based on encoding type. This reduces the amount of switching between regular and bypass coding mode. This is mostly related to high complexity implementations, where frequent switching would diminish the performance gained by using speculative computing. But it should be kept in mind while designing a correctly sequenced binarizer.

4.11 Complete Example

Figure 18 shows how the initial transform coefficient level data from the 4×4 Transform block, totaling $16\text{-bits} \times 4 \times 4 = 256\text{-bits}$, is now reduced down to 81-bits using the HEVC adaptive binarization scheme. The size is now equal to about 32% of the original size, even before CABAC encoding is performed. Notice also how the symbols are biased towards '1' for regular coded bins (SIGN, SIG, ALG1 and ALG2). Counting 21 total '1's and 5 total '0's. This leaves a probability skew of about 80% for the most probable symbol. These bins do of course not share the same probability models (contexts), but this does give some insight into why this binarization scheme is so efficient when paired with a CABAC encoder. The Bypass coded bins (SIGN and ALRem) show a more equally probable distribution of symbols, with a count of 31 total '1's and 24 total '0's, and a probability skew of 56% for the most probable symbol. This is an example of why the higher throughput bypass mode can be utilized with an insignificant compression penalty. A major design goal of HEVC was to reduce the amount of regular coded bins, i.e. increase throughput. The adaptive binarization scheme for the transform coefficient level values was an important factor in achieving this goal.

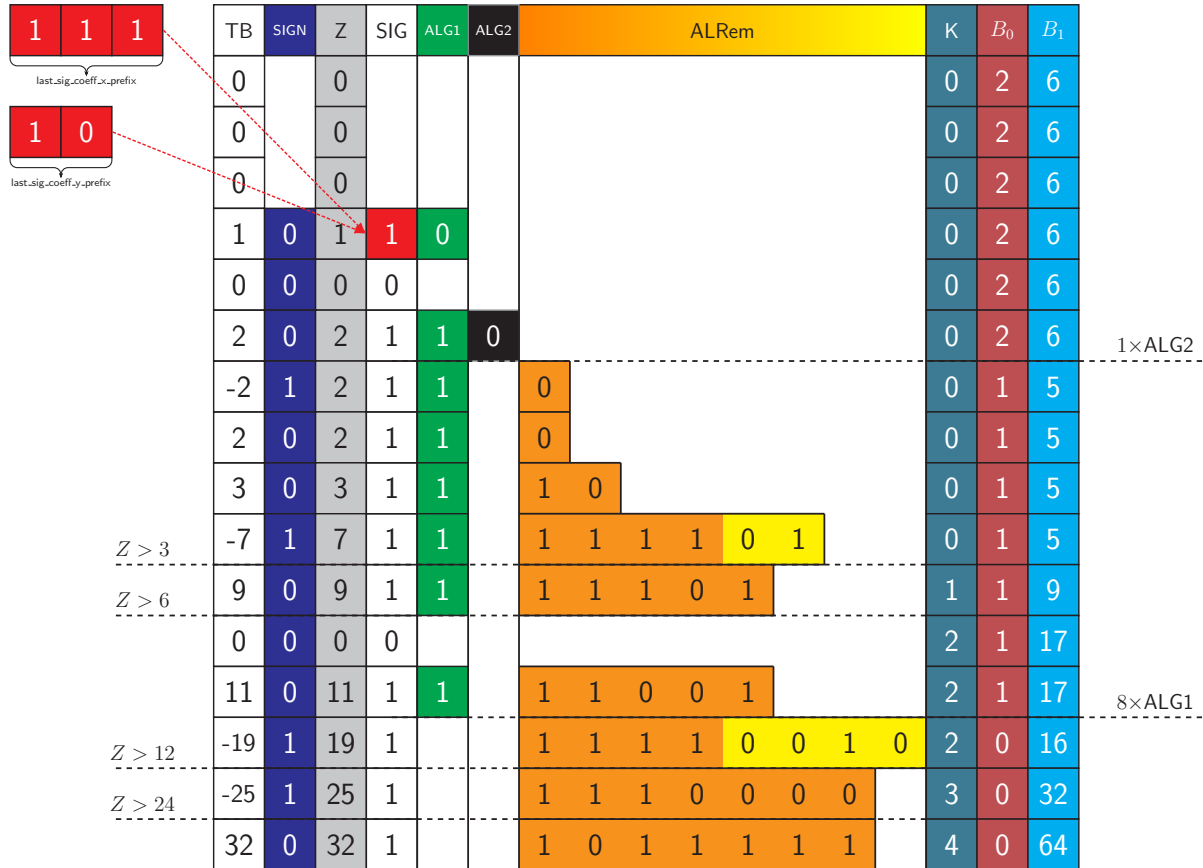


Figure 18: Complete Example, showing how the Transform Block from Figure 11 is binarized using the Diagonal Scan direction. Threshold variable values for each scan, as well as the condition that triggers a transition is also shown. Note that the red square under SIG is not signaled, but inferred directly from the LAST coordinates.

5 Context Modeling

Context modeling of the residual data is restricted to SIG, ALG1 and ALG2, with SIGN and ALRem being bypass coded, and thus not needing a context model. This chapter covers context table structures, context initialization, as well as an short introduction to context selection for SIG, ALG1 and ALG2 in 4×4 Transform Blocks. Due to time constraints, context index calculation was not implemented.

5.1 sig_coeff_flag

SIG in 4×4 TBs uses a position based context selection. The context selection of SIG for larger transform blocks uses a substantially more advanced method, where context index calculation is based on a selection of previously processed bins.

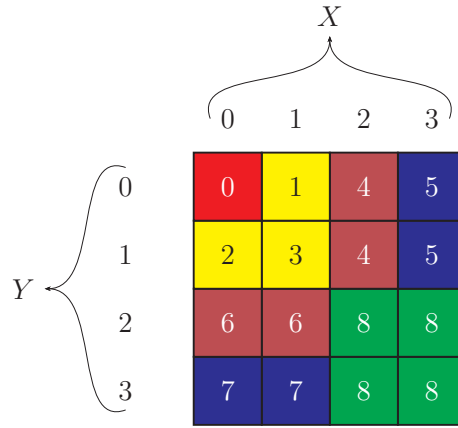


Figure 19: SIG context assignment for 4×4 Transform Blocks.[9]

5.2 sig_coeff_greater1_flag and sig_coeff_greater2_flag

Both the ALG1 and ALG2 uses 6 sets of context models. 4 sets belonging to the luma component, and 2 sets belonging to the chroma component. The details for selection of these sets are covered in section 9.3.4.2.6 and 9.3.4.2.7 in the standard document. Simply put, context sets are calculated based on previously subblock encoding results.

Each context set related to ALG1 contain 4 probability models. Where these models within a set are selected based on the previous values of ALG1 encoded in a subblock.

ALG1 subblock adaption rules:

- At the start of a subblock, the contex index within a set(ctxInc) is set equal to 1.
- For occurrences of absolute values equal to 1, ctxInc is incremented by 1(up to a maximum of 3).
- If any occurrence of an absolute value greater than 1, ctxInc is permanently set to 0, terminating any further adaption.

For ALG2, each related set only contain a single context model. Resulting in a probability model selection equal to the set selection.

5.3 Probability model

Each context table index contain a probability model. It consists of the two variables, valMps and pStateIdx. valMps(Value Most Probable Symbol) is the actual value of the most probable symbol. i.e. '1' or '0'. pStateIdx(Probability State Index) is a reference to a probability estimate. Allowing for probability estimation to be performed using a finite state machine approach. This is a performance optimization that reduces costly multiplications down to a simple table lookup. But requires an additional transition table lookup for each encoded bin. The required precision for the probability model is 1-bit for valMps and 6-bits for the pStateIdx. Totaling 7-bits for each context in the context table.

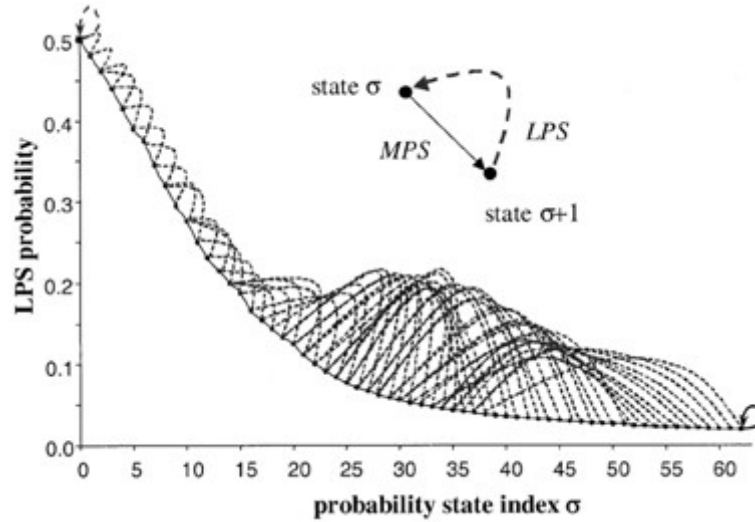


Figure 20: FSM based probability estimation in CABAC. Figure is taken from a presentation for CABAC in h.264.[3] Dotted lines are transitions performed when the bin to be encoded is not equal to valMps, and solid lines are transitions performed when the bin is equal to valMps.

5.4 Initialization

Before encoding, every context is initialized with unique probability model values. This initialization process is covered in detail in section 9.2.1.1 of the standard document. The process is described for finding the initial probability model values of a specific context index. This is done by using tables containing initValues for all context indexes, as well as the context index, initType and SliceQP variables. initType and SliceQP can be seen as simple inputs to the context modeler. With initType ranging from 0-2 and SliceQP ranging from 0-51.

Appendix G shows the C# functions used to generate the context table initial values used in the hardware and software encoders implemented in this project.

6 HEVC CABAC Algorithm

The HEVC CABAC algorithm is documented in the standard document using UML-like flowcharts. This documentation is very thorough, and forms a good basis for implementing a software encoder or decoder. The flowcharts mostly revolve around encoding or decoding a decision. Where a decision is either encoding or decoding a single symbol('1' or '0'). Most of the higher level CABAC parsing process description is limited to decoding only. These higher level processes are largely related to ordering of the syntax elements to be binarized and encoded, and therefore is not vital to the implementation of the CABAC encoder module itself. Thus it is possible perform correct CABAC encoding of the transform coefficient level values of 4×4 Transform Blocks, given that the syntax elements are binarized in proper order, and that accompanying context indexes are correct. Even if the higher level processes are not implemented.

6.1 Overview

The algorithm is based on arithmetic coding, and is described in the standard as being based on the principle of recursive interval subdivision. Where the encoded output is represented as a sub-interval between 0.0 and 1.0. Instead of first finding the final sub-interval, and then outputting its lower bound as the finished encoded bitstream, the algorithm is designed to output uniquely decodable sub-interval at each recursive step as it climbs down towards the final sub-interval. This is achieved by representing intervals with finite precision range variables, and checking if these ranges fall below a certain threshold at each recursion step. A consequence of these range variables having finite precision, is that they require rescaling when they fall below these thresholds. This rescaling is simply implemented by a logical left shift(doubling) of the range variables. Note that some ivlLow rescaling is dependent on which region(lower, upper or middle) it resides in, as to prevent overflowing.

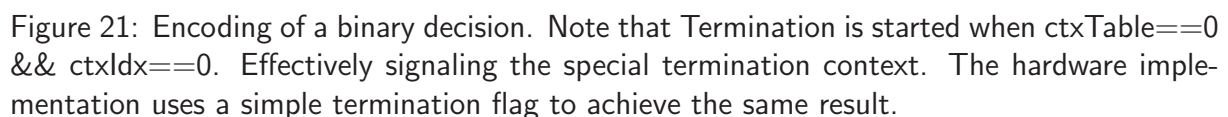
6.2 Variable Initialization

Updates towards variables inside the flowcharts are global. Only during initialization/reset of the encoder or decoder should these variables be set to their initial values.

Variable	Initial Value	
	Decoder	Encoder
codlRange	510	510
codlLow	0	0
qCodlRangeidx	0	0
CodlrangeLPS	0	0
codlOffset	0	read_bits(9)
codlLow	0	0
BitsOutstanding	NA	0
firstBitFlag	NA	1

Table 8: Initial values for encoding and decoding. See Section 6.7 for read_bits function description.

Input to EncodeDecision is the context index, BypassFlag, as well as the binary symbol('1' or '0') to be encoded. Output is 0 or more finished encoded binary symbols. This is called for each bin in the binarized syntax element, using the same context index and BypassFlag. The actual software model uses a functions more closely resembling the standard document flowcharts.



6.4 Renormalization

Renormalization procedure is responsible for renormalizing the range variables such that sufficient precision is available, as well as actually outputting finished encoded symbols. Efficient hardware implementation of renormalization is challenging due to the nested while loop structure. With the second while loop being performed in the PutBit procedure. Note that outputting a symbol for a sub-interval in the middle region is deferred until an interval in the upper or lower region is found. This is needed as it is not possible to know for certain what symbol should be output when rescaling in the middle region.[1] This is achieved using a bitsOutstanding variable to count consecutive symbols found in the middle region.

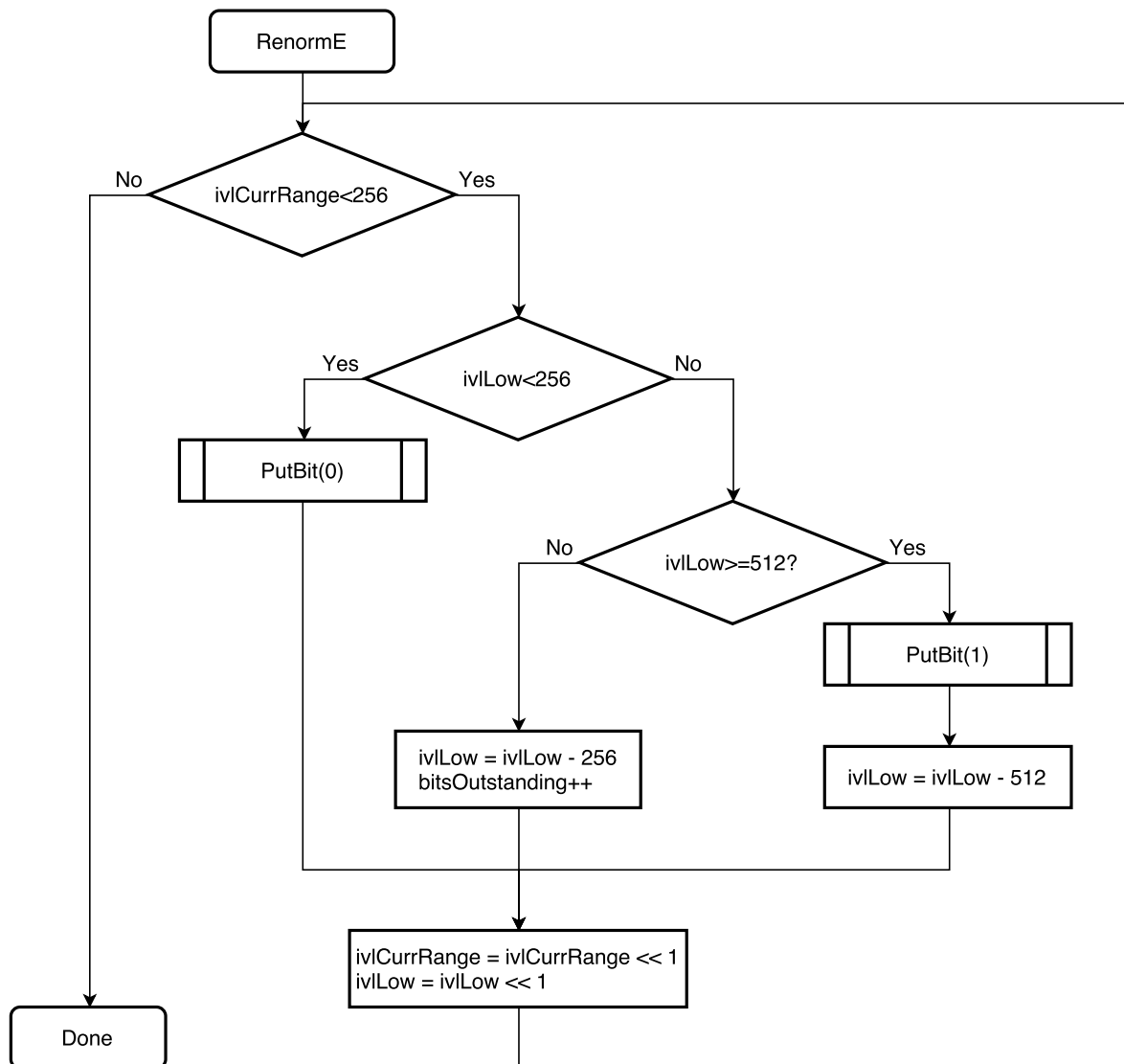


Figure 22: Renormalization during encoding of a binary decision.

6.5 Writing to Bitstream

Writing to bistream is done using the PutBit procedure. Which is also responsible for skipping the first encoded symbol, as well as outputting the bitsOutstanding. The exact reasoning for skipping the first bit is not included in the standard document.

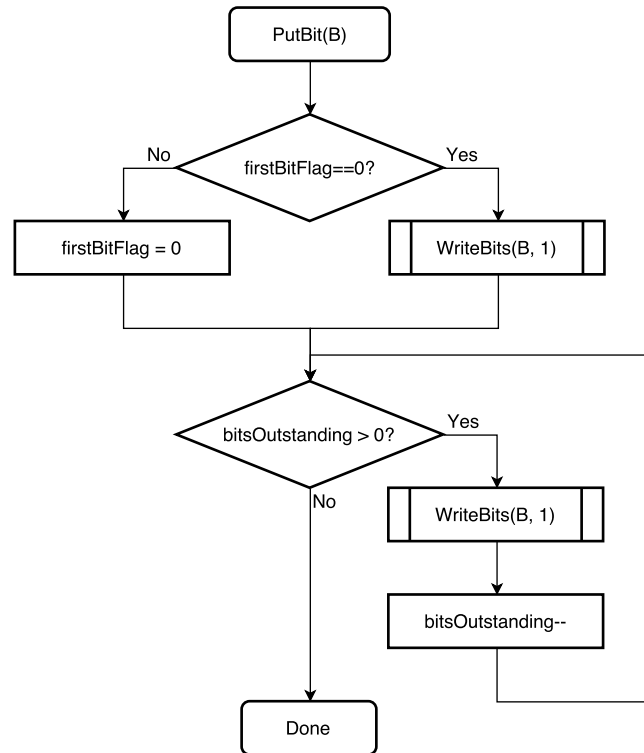


Figure 23: PutBit function. See Section 6.6 WriteBits function description.

6.6 WriteBits

The WriteBits(int B, int N) function is an abstraction for adding bits to the bitstream. It is specified to write N-bits with the value B to the bitstream, and then advance a bitstream-pointer by N. The bitstream-pointer is not vital to the encoding or decoding of the bitstream, but is related to proper alignment of the data structure.

6.7 read_bits

The read_bits(int N) return the N first bits from the bitstream, starting with at the current bitstream-pointer location. Then the bitstream pointer is advanced by N.

6.8 Decoding a Decision

Input to DecodeDecicion is the context index and BypassFlag. The function will return a single decoded symbol binVal every time it is called. The bitstream is read internally using the read_bits function. When the range variables falls below a certain threshold, renormalization is performed by reading the next symbol in the bitstream. Where this symbol essentially contain information about the next sub-interval.

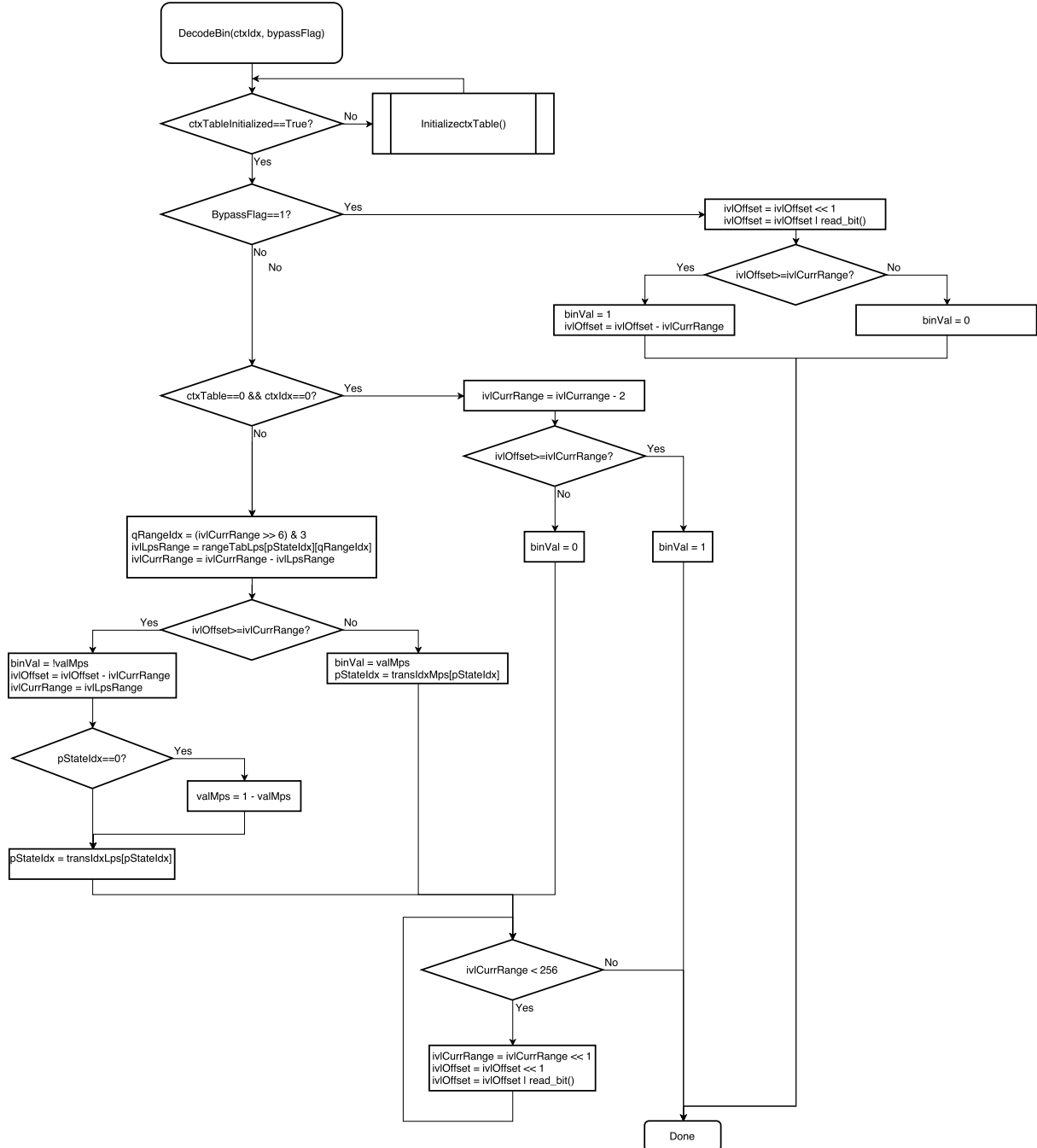


Figure 24: Decoding of a binary decision.

7 Software Model

For the purpose of understanding and verifying the functionality of the HEVC CABAC entropy coding scheme, a independent software model was developed. This section covers the implemented software model. Development is based on Recommendation ITU-T H.265(version 12/2016)[10]. Naming convention is inherited from this document.

The documentation[10] for HEVC is based on C-like pseudocode and UML-diagrams. C# was chosen for its C-like syntax, as well as a shorter development time frame compared to C/C++. The only significant downsides to choosing C# over C++, is a potential performance loss. Which was not deemed important. The software model uses a Windows Forms based interface. This limits the software to windows based computers. The submodules is developed independent from the interface module, facilitating porting.

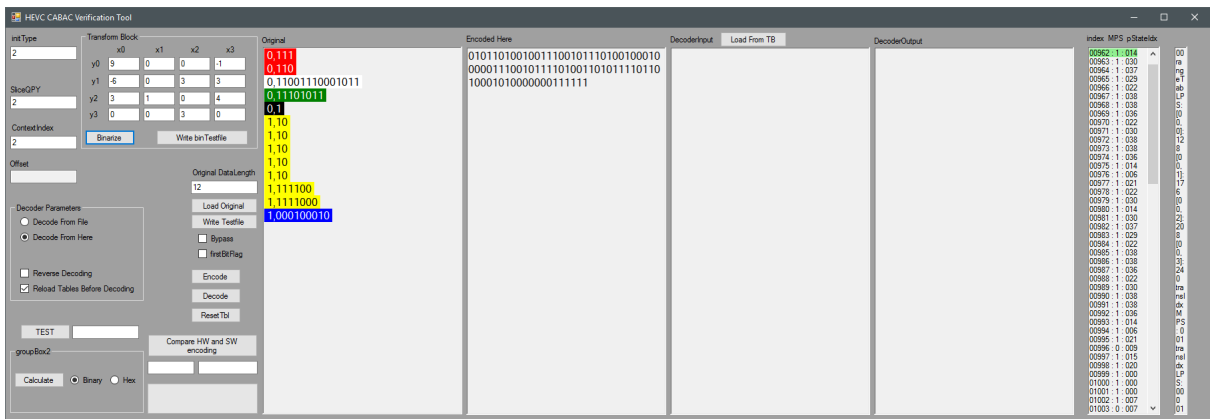


Figure 25: Software model user interface.

The main requirement for the software was to be able to generate binary test and verification strings to be used for hardware verification. For this reason, the data structure is simplified compared to the bitstream structure described in the standard. Designing a HEVC compliant data structure is well outside the scope of the CABAC entropy coding. In addition to the encoder module, the decoder was also implemented in software. This allowed for functional verification of both the software and hardware encoder.

7.1 Binarization and Context Index Calculation

While the documentation for the encoder and decoder is very thorough, covering all steps in great detail. Grasping the inner workings of the implemented binarizer and Context Index Calculator is a bit more challenging. The reason for this is that while the encoder/decoder interfacing is reliant on understanding the outputs of the Binarizer and Context Index Calculator. Implementing a correct Binarizer and Context Index Calculator requires understanding the structure of all the data types used in HEVC. This restriction, along with the reduced amount of supported syntax elements, has led to a crude implementation of binarization and context modeling. The alternative approach of using the official HEVC Test Model(HM) was deemed too time consuming.

The current Software binarizer does not binarize the Transform Block correctly. The plan was to address this issue with the implementation of the context index calculator, but this was not completed due to time constraints.

7.2 Encoder and Decoder

With the C# language supporting every statement in the documentation flowchart, implementing a software encoder and decoder was relatively easy. The encoder source code can be found in **Appendix D**, and the decoder source code can be found in **Appendix E**. Instead of working with bit-files, the encoder and decoder uses ASCII based '1's and '0's as the symbols to be encoded and decoded. This allowed for simple interfacing between the hardware testbenches. The context table is implemented using the same initial table as the hardware implementation. With context index calculation being unfinished, the current system uses a static values for SliceQP, initType and ctxIdx.

The initial plan was to design a robust and user friendly system that would allow for proper verification of the hardware encoder. This would involve the implementation of a debinarizer and context index calculator. But with so many parts of the system left incomplete, the current software model is better described as a C# sandbox used to verify the Hardware CABAC encoder. **Appendix I** contains a guide for using a simplified version of the software encoder and decoder.

7.3 Interfacing With TestBenches

The software models main purpose was to be able to verify the correctness of the Hardware module outputs. To make this process more efficient, both the testbenches and software model should read and write to the same files. This concept was planned for both the Binarizer/Context Modeller and the Hardware Encoder, but it was only completed for the Hardware Encoder. The asynchronous fifo was only verified using the accompanying testbench.

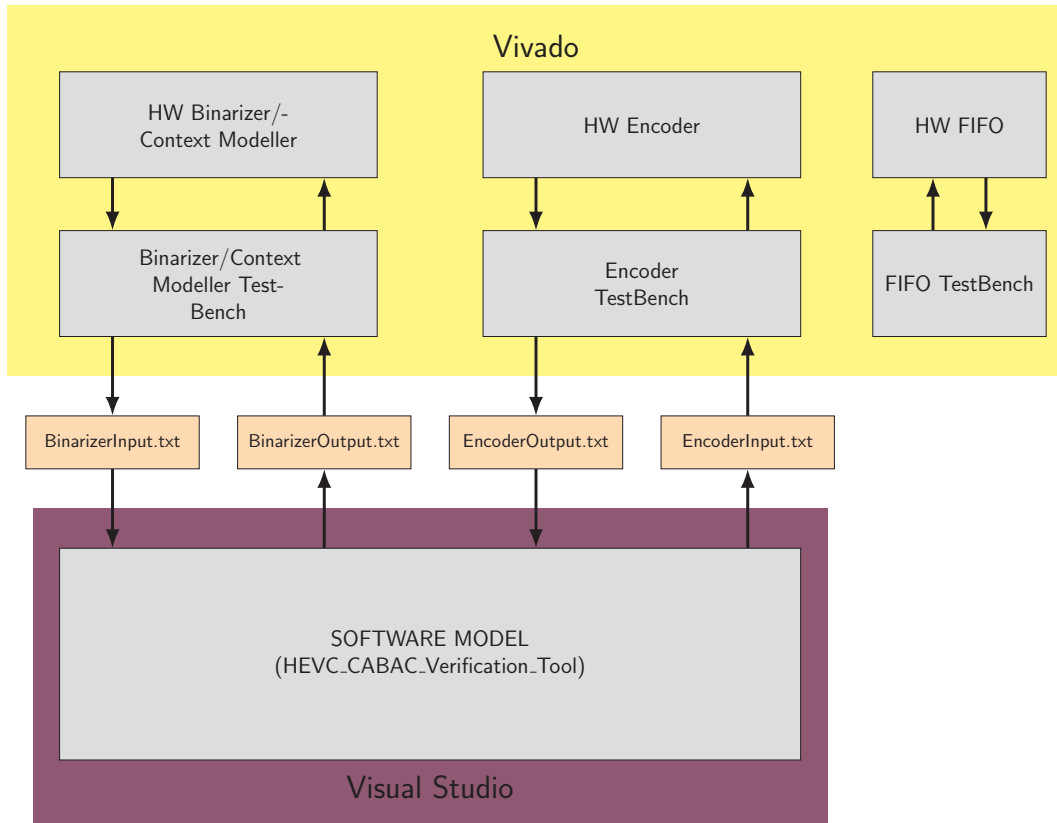


Figure 26: System structure for interfacing with between hardware and software. Including development platforms. Note that the Binarizer/Context Modeller was not completed. **Appendix G** covers setup for the completed system, as well as a tutorial on how Hardware/- Software Encoding can be compared.

8 Hardware Implementation

The naming convention of variables, signals and types is largely inherited from ITU-T H.265 v4 chapter 9. Some changes are made to account for the differences in software vs hardware design. The target during development is the ZEDBOARD. This board uses the Zynq-7000(7z020clg484-1) all programamable SoC.

8.1 Modules

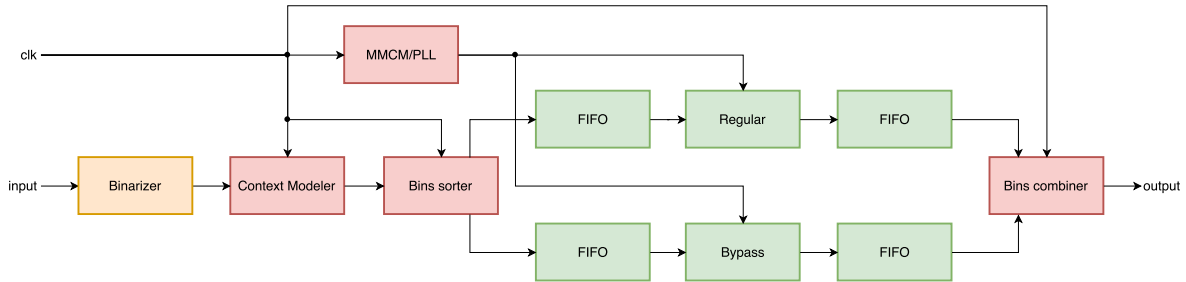


Figure 27: The planed modules for the completed CABAC encoder system. Implemented modules in green, partially implemented modules in orange, and incompleted modules in red. Note that the Regular and Bypass encoder is combined in the current hardware design.

The hardware implementation is split into different modules to facilitate greater abstraction levels, as well as simplify development towards a specific timing constraint. Because of the sequential nature regular and bypass coding, Any CABAC implementation could benefit from supporting different clock domains for the different modules. This is one of the reasons that the fifo is designed to be asynchronous.

8.2 Parameters

Every Module except the Asynchronous fifo modules comes paired with a parameter settings file. Due to the different abstraction in verilog, parameters for the fifo is changed in the source file. These parameters effect the functionality and properties of the modules. Such as I/O width and For loop depths. The main goal of the implemented code is to be able to optimize towards hardware targets by manipulating the parameter files.

8.3 Byte Packing and Alignment

Data structure alignment of the finished encoded bitstream is important, because of the variable length output of the implemented encoder. The plan was to first output finished encoded symbols to a fifo, and have a barrel shifter combine completed sections of the data. Due to time constraints, this was not completed.

8.4 Binarizer Implementation

The implemented binarizer was designed to efficiently binarize 4×4 Transform Blocks. Several methods for binarization of the relevant syntax elements were explored, with the main focus of finding an efficient implementation of coding the ALRem syntax element. This implementation was designed without a predefined interface from the rest of the encoder. Therefore this work should only serve as inspiration for designing a compliant binarizer. Further work should also try to incorporate both binarization and context index calculation into a single module. The examples here use the same transform block as the one covered in section 4. Where the logical rules used are more closely related to the simplified rules than the original. These rules are covered in 4.8.

The Binarizer code is provided in **Appendix A**. Much of the complexity for implementing a hardware binarizer for the residual coding syntax elements lies in efficient implementation of the different scan directions, as well as proper handling of the adaption rules. Both of these challenges are solved in this current design. Efficient binarization of the `coeff_abs_level_remaining` is also achieved.

8.4.1 Syntax Frames

For the purpose of greater abstraction levels, as well as more efficient hardware implementation. A sort of preprocessing procedure is applied before certain finished syntax elements are output. This allows parallel processing of the residual data, such as OR-reductions and sign-bit checking, while still obeying the strict rules of the finished syntax elements. These preprocessed vectors are referred to as frames.

8.4.2 `last_sig_coeff`

Binarization of `last_sig_coeff` prefixes for all the different scan directions were implemented by using constant arrays of integers. This allowed indexing of the input Transform blocks to be performed using a for-loop structure. Where each element is checked to see if it is non-zero. This was implemented by indexing the `sig_coeff_frame` seen in Figure 28.

8.4.3 sig_coeff_flag

The sig_coeff_flags is a Fixed Length Binarization that represents every non-zero element in the TB. The number of syntax elements is derived from the last_sig_coeff syntax elements, varying from 0 to 15. The reason for the amount being 15 instead of 16, is because the first non-zero element is inferred directly from the coordinates provided by last_sig_coeff.

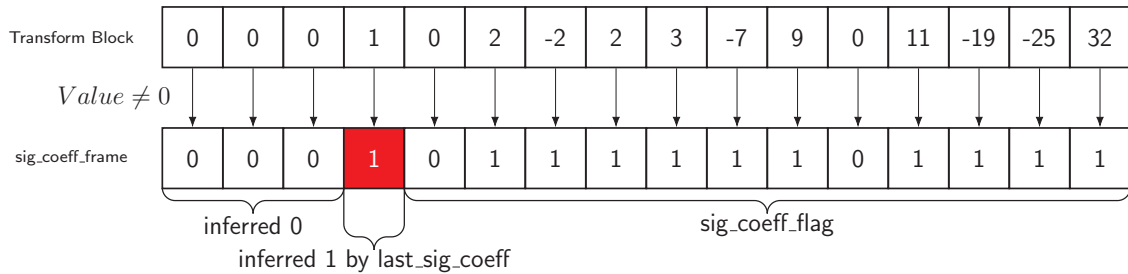


Figure 28: Binarization of sig_coeff_flag

Figure 28 shows how the first non-zero element index is derived from last_sig_coeff. The subsequent elements in the TB is then checked for non-zero elements. With '0' resulting in '0' and '1' resulting in '1' in the finished binarized sig_coeff_flag Syntax Elements. Notice how sig_coeff_flag can be implemented by outputting the remainder of the full sig_coeff_frame starting with the index after the one inferred by last_sig_coeff. Overall a pretty simple binarization procedure. The hiding of the first non-zero element does however introduce some complications when binarizing elements with dependencies on sig_coeff_flag. One simple solution to this is to keep this non-zero element stored in the full sig_coeff_frame. Fig. 28 shows this non-zero element in red.

8.4.4 coeff_abs_level_greater1_flag

The `coeff_abs_level_greater1_flag` is a Fixed Length Binarization that indicates if a non-zero element ('1's in `sig_coeff_flag`) has an absolute value of greater than 1. The amount of syntax elements is derived from the amount of '1's in `sig_coeff_flag` (pluss the inferred first '1' seen in red in Fig. 31), but has a maximum of 8 per subblock.

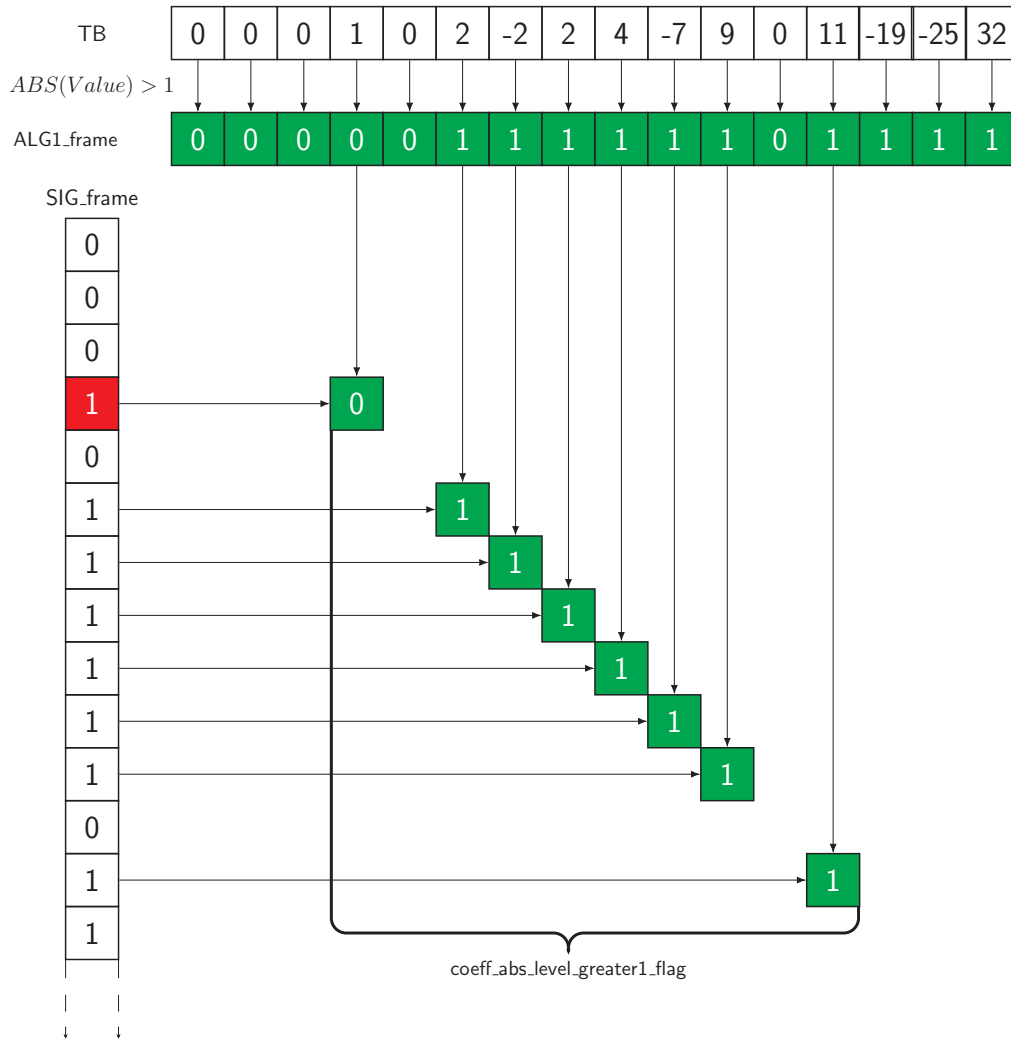


Figure 29: Binarization of `coeff_abs_level_greater1_flag`

Fig. 31 shows how the binarization procedure can be implemented using the full `coeff_abs_level_greater1_frame` and `sig_coeff` frame vectors.

8.4.5 coeff_abs_level_greater2_flag

The `coeff_abs_level_greater2_flag` is a Fixed Length Binarization that indicates if a non-zero element ('1's in `sig_coeff_flag`) has an absolute value of greater than 2.

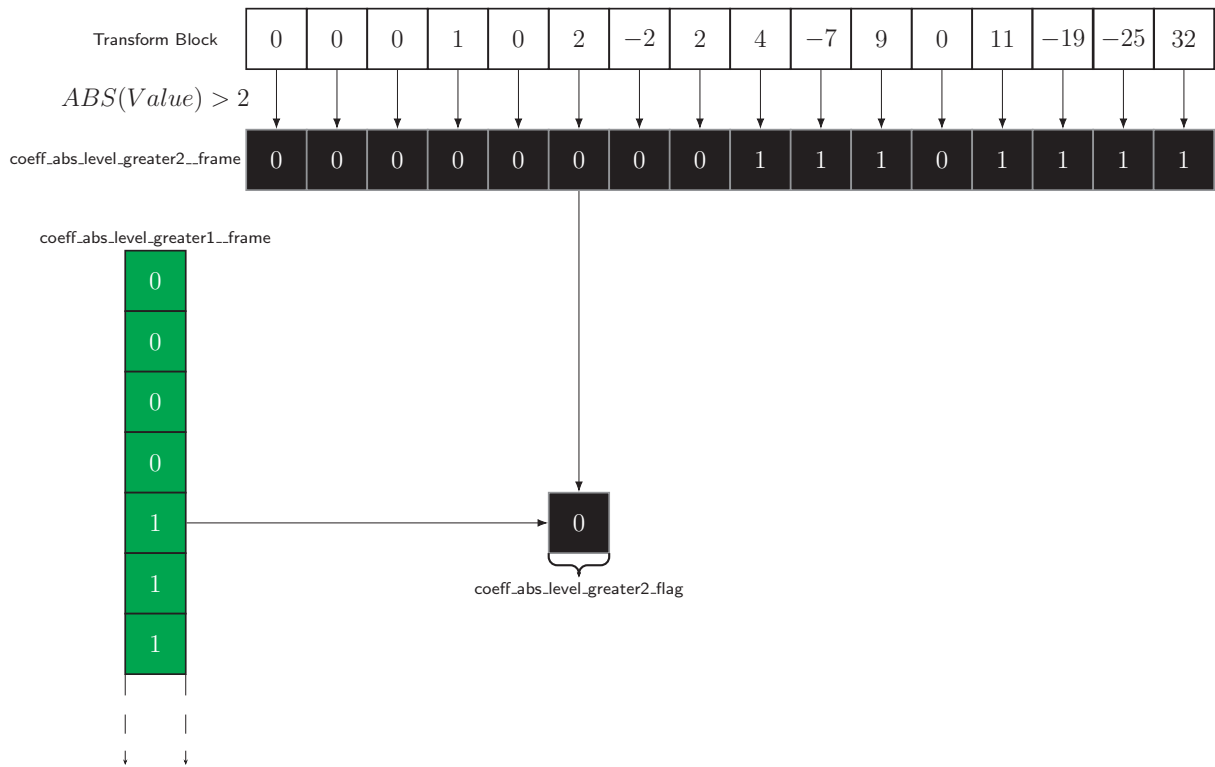


Figure 30: Binarization of `coeff_abs_level_greater2_flag`

Very similar to `coeff_abs_level_greater1_flag`, but this syntax element is limited to a length of 1 per subblock. There exists a special case for when 8 consecutive occurrences of ALG1 with the value '0' appear before a non zero element with an absolute value of greater than 2. Here the ALG2 should not be output, but skipped entirely. With the simplified rules it is easy to sometime output ALG2 here. Special care should be put into handling this possible error. This error is the likely culprit of the wrong binarization by the software model.

8.4.6 coeff_abs_level_remaining

The adaptive binarization scheme for ALRem is a substantially more advanced process than the previous syntax elements covered here. The standard describes binarization of `coeff_abs_level_remaining` as a concatenation of TRk and EGk. Both of these coding methods may seem complex enough by themselves, and when truncated even more so. A substantial amount of effort was spent trying to find an efficient way of computing these codes, with very impractical or low performance results. The MIT Open Access Article on Entropy Coding in HEVC[11] does however introduce an alternative representation using a concatenation of a unary prefix and a fixed length suffix.

TabIdx	Z_{min}	Z_{max}	Prefix bins	Suffix bins	Prefix Length	Suffix Length	Max k
0	0	$k^2 - 1$	0	C	1	k	4
1	1×2^k	$2 \times 2^k - 1$	10	C	2	k	4
2	2×2^k	$3 \times 2^k - 1$	110	C	3	k	4
3	$2^k \times (2^0 + 2)$	$2^k \times (2^1 + 2) - 1$	1110	C	4	k	4
4	$2^k \times (2^1 + 2)$	$2^k \times (2^2 + 2) - 1$	11110	xC	5	1 + k	4
5	$2^k \times (2^2 + 2)$	$2^k \times (2^3 + 2) - 1$	111110	xxC	6	2 + k	4
6	$2^k \times (2^3 + 2)$	$2^k \times (2^4 + 2) - 1$	1111110	xxxC	7	3 + k	4
7	$2^k \times (2^4 + 2)$	$2^k \times (2^5 + 2) - 1$	11111110	xxxxC	8	4 + k	4
8	$2^k \times (2^5 + 2)$	$2^k \times (2^6 + 2) - 1$	111111110	xxxxxC	9	5 + k	4
9	$2^k \times (2^6 + 2)$	$2^k \times (2^7 + 2) - 1$	1111111110	xxxxxxC	10	6 + k	4
10	$2^k \times (2^7 + 2)$	$2^k \times (2^8 + 2) - 1$	11111111110	xxxxxxxC	11	7 + k	4
11	$2^k \times (2^8 + 2)$	$2^k \times (2^9 + 2) - 1$	111111111110	xxxxxxxxxC	12	8 + k	4
12	$2^k \times (2^9 + 2)$	$2^k \times (2^{10} + 2) - 1$	1111111111110	xxxxxxxxxC	13	9 + k	4
13	$2^k \times (2^{10} + 2)$	$2^k \times (2^{11} + 2) - 1$	11111111111110	xxxxxxxxxC	14	10 + k	4
14	$2^k \times (2^{11} + 2)$	$2^k \times (2^{12} + 2) - 1$	111111111111110	xxxxxxxxxC	15	11 + k	3
15	$2^k \times (2^{12} + 2)$	$2^k \times (2^{13} + 2) - 1$	1111111111111110	xxxxxxxxxC	16	12 + k	2
16	$2^k \times (2^{13} + 2)$	$2^k \times (2^{14} + 2) - 1$	11111111111111110	xxxxxxxxxC	17	13 + k	1
17	$2^k \times (2^{14} + 2)$	$2^k \times (2^{15} + 2) - 1$	111111111111111110	xxxxxxxxxC	18	14 + k	0

Table 9: Alternative representation of ALRem using TrU and FL coding[11]- The suffix bins are shown as x and C, where x represents a bin, and C represents a fixed length bin string of length k.

This alternative representation was key in finding an efficient method for calculating ALRem. The prefix bins is found by simply checking where Z resides. The suffix bins for any value Z within Z_{min} and Z_{max} at a given TabIdx, is found by calculating $Z - Z_{min}$. Note that this will result in an fixed length representation of the binarized suffix, where the length is equal to the suffix length at the given TabIdx. Table 10 shows the generalized version of this table. **Appendix H** documents an interactive model of the table.

Tabldx	Z_{min}	Z_{max}	Prefix bins	Suffix bins	Prefix Length	Suffix Length	Max k
0	0	$k^2 - 1$	0	$Z - Z_{min}$	1	k	4
1	1×2^k	$2 \times 2^k - 1$	10	$Z - Z_{min}$	2	k	4
2	2×2^k	$3 \times 2^k - 1$	110	$Z - Z_{min}$	3	k	4
3	$2^k \times (2^0 + 2)$	$2^k \times (2^1 + 2) - 1$	1110	$Z - Z_{min}$	4	k	4
4	$2^k \times (2^1 + 2)$	$2^k \times (2^2 + 2) - 1$	11110	$Z - Z_{min}$	5	1 + k	4
5	$2^k \times (2^2 + 2)$	$2^k \times (2^3 + 2) - 1$	111110	$Z - Z_{min}$	6	2 + k	4
6	$2^k \times (2^3 + 2)$	$2^k \times (2^4 + 2) - 1$	1111110	$Z - Z_{min}$	7	3 + k	4
7	$2^k \times (2^4 + 2)$	$2^k \times (2^5 + 2) - 1$	11111110	$Z - Z_{min}$	8	4 + k	4
8	$2^k \times (2^5 + 2)$	$2^k \times (2^6 + 2) - 1$	111111110	$Z - Z_{min}$	9	5 + k	4
9	$2^k \times (2^6 + 2)$	$2^k \times (2^7 + 2) - 1$	1111111110	$Z - Z_{min}$	10	6 + k	4
10	$2^k \times (2^7 + 2)$	$2^k \times (2^8 + 2) - 1$	11111111110	$Z - Z_{min}$	11	7 + k	4
11	$2^k \times (2^8 + 2)$	$2^k \times (2^9 + 2) - 1$	111111111110	$Z - Z_{min}$	12	8 + k	4
12	$2^k \times (2^9 + 2)$	$2^k \times (2^{10} + 2) - 1$	1111111111110	$Z - Z_{min}$	13	9 + k	4
13	$2^k \times (2^{10} + 2)$	$2^k \times (2^{11} + 2) - 1$	11111111111110	$Z - Z_{min}$	14	10 + k	4
14	$2^k \times (2^{11} + 2)$	$2^k \times (2^{12} + 2) - 1$	111111111111110	$Z - Z_{min}$	15	11 + k	3
15	$2^k \times (2^{12} + 2)$	$2^k \times (2^{13} + 2) - 1$	1111111111111110	$Z - Z_{min}$	16	12 + k	2
16	$2^k \times (2^{13} + 2)$	$2^k \times (2^{14} + 2) - 1$	11111111111111110	$Z - Z_{min}$	17	13 + k	1
17	$2^k \times (2^{14} + 2)$	$2^k \times (2^{15} + 2) - 1$	111111111111111110	$Z - Z_{min}$	18	14 + k	0

Table 10: Alternative representation of ALRem where $Z - Z_{min}$ at any Tabldx is a binary number with length indicated by the Suffix Length at that specific Tabldx

Tabldx	Z_{min}	Z_{max}	Prefix bins	Suffix bins	Prefix Length	Suffix Length	Max k
0	0	0	0		1	k	4
1	1	1	10		2	k	4
2	2	2	110		3	k	4
3	3	3	1110		4	k	4
4	4	5	11110		5	1 + k	4
5	6	9	111110	10	6	2 + k	4
6	10	17	1111110		7	3 + k	4
7	18	33	11111110		8	4 + k	4

Table 11: Example Binarization of ALRem for $Z = 8$ and $k = 0$ using the method proposed in Table 10

Tabldx	Z_{min}	Z_{max}	Prefix bins	Suffix bins	Prefix Length	Suffix Length	Max k
0	0	3	0		1	k	4
1	4	7	10		2	k	4
2	8	11	110	00	3	k	4
3	12	15	1110		4	k	4
4	16	23	11110		5	1 + k	4

Table 12: Example Binarization of ALRem for $Z = 8$ and $k = 2$ using the method proposed in Table 10

This alternative approach was implemented both using a table based approach, and using the switch case statement. The current table based approach showed very low performance, but this could be due to using a single table for all values of k . The case based approach, seen in **Appendix A**, did however show decent performance. Further work of designing a binarizer should incorporate one of these methods to binarize ALRem. The current designs does not signal the actual unary prefix code, as this coding can simply be inferred when the length is known. Therefore, the prefix is simply signaled using the `prefixLength` output. This would require extra logic in the bypass coder. But with this extra logic being relatively simple, the reduction in signaling could be very beneficial.

8.4.7 coeff_abs_level_sign_flag

The `coeff_abs_level_sign_flag` is a Fixed Length Binarization that indicates if a non-zero element('1's in `sig_coeff_flag`) has an value of less than 0. This is a very simple binarization that only require checking of the sign bit. The only real complexity lies in supporting the optional sign bit hiding technique. Which in practice involves skipping this binarization under certain conditions.

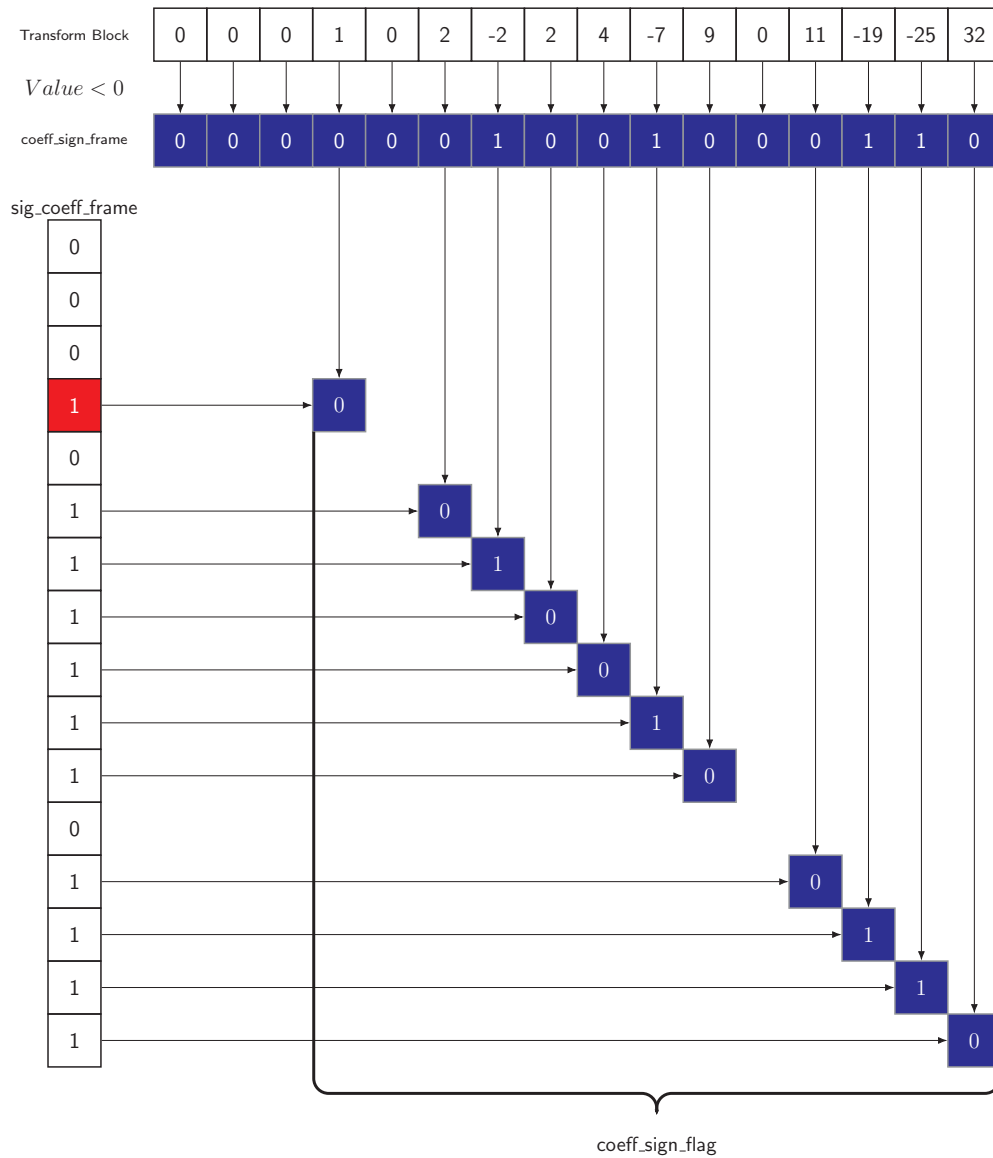


Figure 31: Binarization of `coeff_abs_level_sign_flag`

8.5 Context Index Calculator

The context index calculator module was not implemented. Although this module was first planned to be designed separately, integration with the binarizer module could be very beneficial. This is due to the large amount of data dependencies shared between them. Designing a Binarizer/context Index Calculator without first knowing the completed interface of the rest of the HEVC encoder modules is a very inefficient approach. As the amount of different data and signals(syntax elements) it needs to be able to support is rather comprehensive.

8.6 CABAC Encoder

The Hardware Encoder was developed in VHDL alongside the C# HEVC CABAC Verification Tool. This implementation performs Context-Adaptive Binary Arithmetic encoding as specified in the Recommendation ITU-T H.265. The only caveat being that the context table is limited to the residual coding syntax elements. Code is provided in Appendix B.

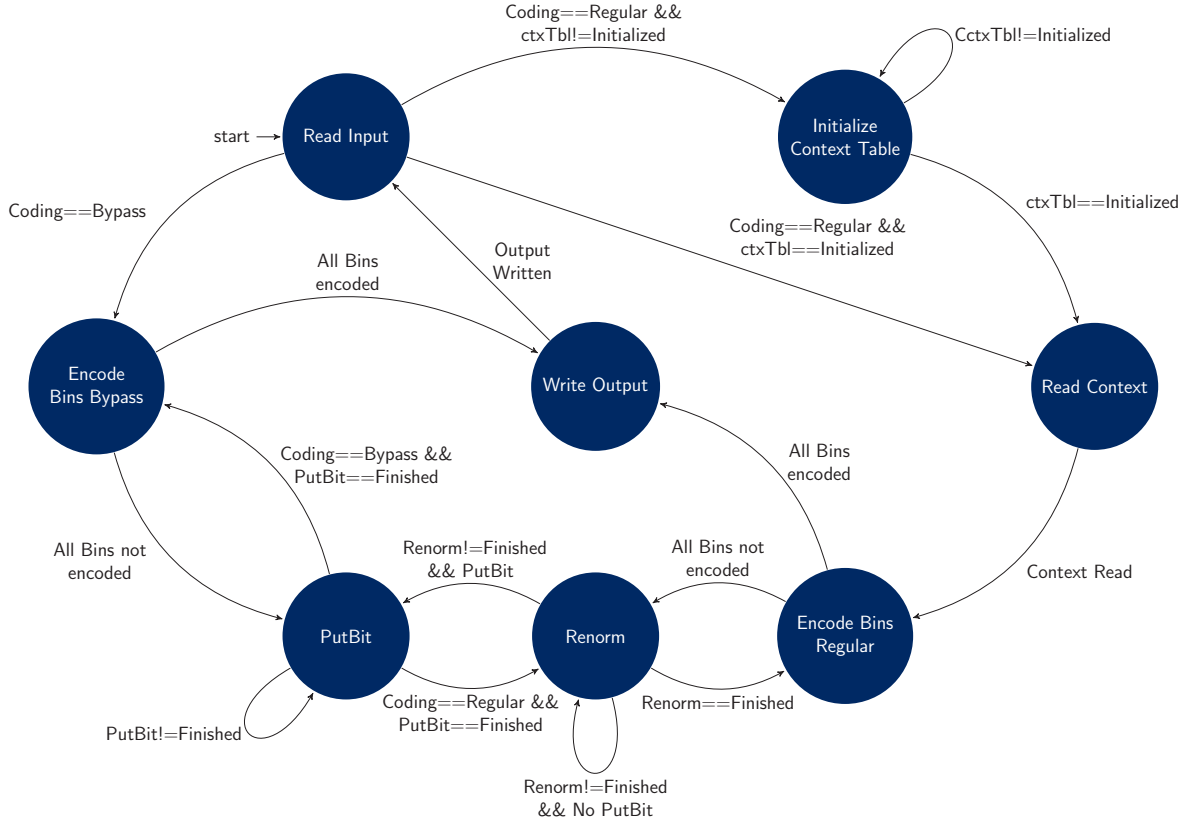


Figure 32: Simplified state machine diagram for the hardware encoder. Context update is written during the Write Output state.

Encoding of termination is implemented, but not included in this diagram. The initial plan was to decouple the Bypass and Regular encoders into two distinct modules. This would allow for implementation of a dispatcher to split the workload whenever a regular and a bypass coded bin are encoded in order. This did however introduce a few complications to the update of the range variables. Bypass encoding was instead incorporated into the regular coder, as a simple state. This allowed for both the Bypass and Regular parts of the encoder to use the same PutBit state, as well as sharing range variables. The main challenge in implementing the hardware encoder lies in correctness of the nested while loops located in the RenormE and PutBit part of the algorithm. These while loops contributed the largest amount of non-trivial bugs during development, and was the main motivator for developing the software model. Another major challenge is optimizing throughput using efficient pipelining, but due to the time constraints and the complexity of such designs, this was not explored further.

The focus of the current implementation was achieving correct output of the encoder. As can be seen by the code, the resemblance to the specification flowcharts is clear. A consequence of this is that a large amount of possible redundant clock cycles, where the whole cycle is spent checking a simple conditional statement. This is a result of the while-loops in the specification. During **enc_bin_r** a conditional check to see if the output is ready to be written or if a renormalization is required. This conditional check could possibly be moved to the end of the renormalization, effectively skipping this cycle. Further the **RenormE** state includes a conditional check that could possibly be moved as a condition for entering this state. The *bitsOutstanding* loop located in the **PutBit** also introduces complications.

S.E	Type	State												
1	Regular	r_input	r_ctx	enc_bin_r	RenormE	PutBit	RenormE	enc_bin_r	w_output					
2	Bypass									r_input	enc_bin_b	PutBit	enc_bin_b	w_output
Clk		1	2	3	4	5	6	7	8	9	10	11	12	13

Table 13: Sample state sequence for encoding of regular and then Bypass coded Bins for the implemented hardware encoder. Assuming that the context table is already initialized, and that **RenormE** and **PutBit** is limited to single cycle iterations.

Inefficiency of the current state machine is quite apparent. Conditional checks that are close to trivial in software, results in possible excessive cycles for the hardware design. Where it is possible for multiple sequences of **RenormE** and **PutBit** to occur. Degrading performance even further. This demonstrates some of the challenges in implementing a pipelining scheme.

Syntax Element Nr.	Type	State												
1	Regular	r_input	r_ctx	enc_bin_r	RenormE	RenormE	RenormE	PutBit	PutBit	RenormE	enc_bin_r	w_output		
2	Bypass												r_input	enc_bin_b
Clk		1	2	3	4	5	6	7	8	9	10	11	12	13

Table 14: Sample state sequence where multiple **RenormE** and **PutBit** cycles are required for encoding.

Syntax Element Nr.	Type	Pipeline Stage												
1	Regular	r_input	r_ctx	enc_bin_r	RenormE	PutBit	RenormE	enc_bin_r	w_output					
2	Regular		r_input	r_ctx	enc_bin_r	RenormE	PutBit	RenormE	enc_bin_r	w_output				
3	Regular			r_input	r_ctx	enc_bin_r	RenormE	PutBit	RenormE	enc_bin_r	w_output			
4	Regular				r_input	r_ctx	enc_bin_r	RenormE	PutBit	RenormE	enc_bin_r	w_output		
5	Bypass					r_input	enc_bin_b	STALL	STALL	PutBit	enc_bin_b	STALL	w_output	
6	Bypass						r_input	enc_bin_b	STALL	STALL	PutBit	enc_bin_b	STALL	w_output
Clk		1	2	3	4	5	6	7	8	9	10	11	12	13

Table 15: Sample state sequence for encoding of regular and then Bypass coded Bins using a potential pipelined implementation. Assuming that **RenormE** and **PutBit** is completed in a single cycle, which may not be feasible. A proper high performance pipelined implementation would most likely look different, incorporating speculative execution principles.

8.6.1 Interface

Interfacing the encoder is relatively simple. Input data is written along with the accompanying Input data length. When the Start signal is asserted, the encoder will perform Regular encoding if BypassI is low and Bypass encoding if BypassI is high. When encoding is completed, the finished data is written to the Output, along with the Output data length. Encoding is finished when the Finished signal is high. Termination will be encoded if TermI is asserted at the start of encoding. The encoder testbench serves as a reference example of interfacing the encoder.

Port	Direction	Type	Description
Clk	in	std_logic	Clock.
Input	in	std_logic_vector	Input data.
InputLen	in	std_logic_vector	Input data length.
ctxIdx	in	std_logic_vector	Context index.
SliceQPY	in	std_logic_vector	SliceQPY.
initType	in	std_logic_vector	initType.
Resetn	in	std_logic	Active low reset.
Start	in	std_logic	Start encoding signal.
Output	out	std_logic_vector	Output data.
OutputLen	out	std_logic_vector	Output data length.
BypassI	in	std_logic	Input BypassFlag.
BypassO	out	std_logic	Output BypassFlag.
TermI	in	std_logic	Input TerminationFlag.
TermO	out	std_logic	Output TerminationFlag.
Finished	out	std_logic	Encoding finished signal.

Table 16: Encoder ports.

Some redundant ports are present. Offsets from SliceQPY and initType could possible be calculated in the binarizer/context modeler. The termination flags are unnecessary, but requires a rework of the termination logic. Both of these issues should be solved when there is a better picture of all the other modules required in a HEVC encoder.

8.6.2 Parameters

ctxIdxRange should be equal to the total amount of syntax element contexts that are supported. Maximum width of the coeff_abs_level_remaining syntax element is 34. This sets the lower bound of the input width that needs to be supported. Alternatively it is possible to first encode the unary prefix bins before encoding the suffix bins. Resulting in an reduction of minimum input width down to 18. PutBitLoopLen was introduced as a parameter because of its large impact on performance. Changing PutBitLoopLen affects how many bits are processed in the BitsOutstanding loop during each clock cycle.

CABAC_EncParameters	Type	Description
ctxIdxRange	Constant Integer	Total number of different Syntax Elements Context Implemented. Update this number if additional contexts are added.
InputW	Constant Integer	Input Width
OutputW	Constant Integer	Output Width
PutBitLoopLen	Constant Integer	Defines the maximum number of bitsOutstanding that are output in the PutBit state.

Table 17: Hardware Encoder Parameters.

8.6.3 Transition Tables

HEVC CABAC uses precalculated transition tables to perform many of the computationally demanding operations in the algorithm. These tables could possibly introduce some complications for the performance of the design, but none were observed during development. The table arrays consists of 256 elements for the rangeTabLPS, and 64 elements each for both the transIdxLPS and transIdxMPS. The values for these arrays can be copied directly from the standard document. Tables, including the Context tables initials, are stored as text files along with the source code.

8.6.4 Context Table

The context index table for the residual coding syntax elements is initialized with 120(121 counting termination) different initial values dependent on sliceQPY and initType. SliceQPY ranging from 0-51 and initType ranging from 0-2. Resulting in a total of $120 \times 52 \times 3 = 18720$ different initial values. Calculation of these values are covered in Section 5.4 on page 36. It is possible to perform these calculation during the start of each slice, but the current implementation uses a pre-calculated Table for storing the initial values. The table is implemented as 7 bits with (5 downto 0) representing pStateIdx and (6) representing valMPS. The standard document does a very good job of documenting the procedures for initializing all context tables, but does so by spreading them across 38 different tables. **Appendix G** shows the distribution of contexts for the syntax elements in the current context table. Note that the architecture handles the offsets from SliceQPY and initType directly.

8.6.5 Context Handling

Syntax Element	Binarization Process	Context Table Index		
		initType = 0	initType = 1	initType = 2
last_sig_coeff_x_prefix	TR	0 - 17	120 - 137	240 - 257
last_sig_coeff_y_prefix	TR	18 - 35	138 - 155	258 - 175
sig_coeff_flag	FL	40 - 83	160 - 203	280 - 323
coeff_abs_level_greater1_flag	FL	84 - 107	204 - 227	324 - 347
coeff_abs_level_greater2_flag	FL	108 - 113	228 - 133	348 - 353
coeff_abs_level_remaining	TrU, TRk and EGk	Bypass	Bypass	Bypass
coeff_sign_flag	FL	Bypass	Bypass	Bypass

Table 18: Syntax Elements supported in the current context table.

Contexts handling is done by separating them across three levels. This is needed to avoid overwriting the initial values, as well as reducing the amount of read/write accesses to the working context table.

	Elements	Description
ctxIdxTableInitials	18720	Initialized as read-only memory. Contains the proper initialization variables for all values of SliceQP and initType.
ctxIdxTable	120	Initialized from reading ctxIdxTableInitials for a given value of SliceQP and initType. Used as the working context table during regular encoding. Indexed using ctxIdx.
currCtx	1	Initialized by reading ctxIdxTable at a given ctxIdx before regular encoding is performed. Contains valMps(currCtx(6)) and pStateldx(currCtx(5 downto 0)) that is used in regular encoding. After encoding is completed the ctxIdxTable is updated with the value of currCtx at the same ctxIdx it was read from.

Table 19: Hardware Encoder Context Table Structure.

It was originally planned to introduce a way of checking if a context in the ctxIdxTable was initialized, and only load a value from the ctxIdxTableInitials if needed. Implementing this was more challenging than anticipated. Resulting in the current design, where the whole context table is initialized at the beginning of a slice. To better facilitate performance of coding of grouped bins, it could be beneficial to check if the current context index is the same as the next to be encoded. This could save cycles that would be spent on writing and reading to the context table. There are virtually endless ways of implementing context handling, and is something that could be challenging to optimize.

8.6.6 BitsOutstanding Loop

The algorithm requires the ability to output *bitsOutstanding* equal to the largest possible number of finished encoded bins in a slice. This is somewhat trivial for a software standpoint, only requiring the *bitsOutstanding* register to have a sufficiently large precision. Hardware does however require that output is written when the output registers is about to overflow. Even when this overflow limit is reduced by many orders of magnitude, the occurrence of when *bitsOutstanding* overflows the output width should be relatively rare. The current implementation does not address this issue, but it could be beneficial to hold off on fixing this until the design is completely optimized.

8.6.7 Termination

The current termination logic uses termination flag instead of the correct special case *ctxIdx* related to termination. Changing this would only require introduction of the special termination context in the context table, in addition to a simple logical check change. The standard document(NOTE 2 - page 206) [10] hints at a special case of normal decoding that can be used for termination. If this method can be used, it could severely reduce the logic required to perform termination.

8.6.8 Register precision

The number of minimum bits required for each variable in the algorithm is defined as minimum precision in the reference document. They differ from each coding method. Bypass coding requires an additional bit for the **ivlLow** variable. This is due to the built in renormalization in Bypass coding. While most of the range variables are shared, Regular coding uses some extra variables for probability modeling and indexing of transition tables.

Range Variable	Required Precision	
	Bypass	Regular
ivlLow	11	10
ivlCurrRange	9	9
ivlLpsRange	8	8
qRangeIdx	NA	2
pStatIdx	NA	6
valMps	NA	1

Table 20: Range variable precision requirements.

There is no issue related to using a higher precision register than required. So for an encoder implementation for where the range variables are shared, **ivlLow** is simply implemented with 11-bit precision. All tables used are instantiated using the minimum required register precision.

8.6.9 Utilization

The synthesis results shows a relatively low utilization by the CABAC encoder. This should leave room for the rest of the HEVC encoder modules. The design does not infer any latches.

Resource	Utilization	Available	Utilization %
LUT	2930	53200	5,51
LUTRAM	20	17400	0,11
FF	150	106400	0,14

Table 21: Synthesis results using Vivado(2016.4). Device used is the Zedboards 7z020clg484-1. Parameters: InputW = 34, OutputW = 40, PutBitLoopLen = 10.

Resource	Utilization	Available	Utilization %
LUT	2892	53200	5,44
LUTRAM	20	17400	0,11
FF	96	106400	0,09

Table 22: Synthesis results using Vivado(2016.4). Device used is the Zedboards 7z020clg484-1. Parameters: InputW = 18, OutputW = 10, PutBitLoopLen = 5.

Most of the utilization comes from implementing the different tables. Reducing the parameter values, effectively reducing the required mapping of the PutBitVal to the output seems to have negligible effects on utilization. This is something that is likely to change drastically when the FPGA is populated by the rest of the HEVC encoding modules. As the place and route restrictions will be much more prominent. Differences in utilization between synthesis and implementation are practically non-existent.

8.6.10 Frequency

While utilization between synthesis and implementation are virtually identical, maximum frequency vary substantially. The performance is calculated using no input/output wire delays. This should be kept in mind when implementing into a complete system. Critical path of the current design varies depending on what the parameters are set to.

Output Width	Input Width	PutBitLoopLen	Type	Max Frequency
40	34	10	Synthesis	113,430127
40	34	10	Implementation	108,8139282
40	34	5	Synthesis	118,4413123
40	34	5	Implementation	103,4554107
40	18	10	Synthesis	123,3501912
40	18	10	Implementation	106,7919692
40	18	5	Synthesis	122,8501229
40	18	5	Implementation	105,8873359
20	34	10	Synthesis	121,2709192
20	34	10	Implementation	103,6591687
20	34	5	Synthesis	119,9760048
20	34	5	Implementation	105,2299274
20	18	10	Synthesis	123,3806292
20	18	10	Implementation	102,396068
20	18	5	Synthesis	123,3501912
20	18	5	Implementation	101,1838511

Table 23: CABAC Encoder maximum frequency for a few select parameters.

8.6.11 Performance

Quantifying the actual CABAC Encoder performance requires real world test data. Where this test data is the output provided by the binarizer and context modeler in a working HEVC encoder system. The reason for this is that the test data structure has a large impact on the actual run time of the algorithm. Either by the distribution of the regular vs bypass coded bins, or simply by the actual entropy of the data to be encoded. Even the preciseness of the probability models have a large impact on the throughput of regular encoding.

The current testbench uses arbitrary test data either generated using the software model, or simply written to the test text file. In any case, this data is not representative of actual real world binarized syntax elements. It is only useful in verifying correctness of the design when comparing output to the software encoder.

It is possible to generate an estimate of bypass encoder performance. Since this encoding is to be performed for uniformly distributed bins, using randomly generated data might actually be representative.

Syntax Element Length	Encoding Time ns	cycles	bins/cycle	Mb/s
10	2301000	272533,4589	0,366927424	43,4593655
5	1301300	154127,6793	0,324406364	38,4231154
1	501000	59339,09731	0,168522955	19,9600798

Table 24: Performance for bypass encoding. Calculated using $F_{max} = 118.44$ MHz, over 10000 iterations. Syntax element data are randomly generated for varying lengths.

The same can not be said for regular encoding. Where it is not possible to find useful information using the same approach.

Syntax Element Length	Encoding Time ns	cycles	bins/cycle	Mb/s
10	3901000	462039,5581	0,216431685	25.6344527
5	1429500	169311,8555	0,295313047	34,9772648
1	607000	71893,87638	0,139093905	16,4744646

Table 25: Performance for regular encoding. Calculated using $F_{max} = 118.44$ MHz, over 10000 iterations. Syntax element data are randomly generated for varying lengths.

8.7 Fifo Buffer

Because of the irregular throughput of the modules in a CABAC circuit, it is beneficial to introduce a fifo buffer to connect them. For this reason a general purpose fifo was developed. With the Zedboards artix-7 FPGA natively supporting clock manipulation with its Mixed-Mode Clock Manager(MMCM) module, there existed motivation for making this fifo asynchronous. This could possibly allow for the modules to run at different frequencies. Due to the incompleteness of the Binarizer and Context Index Calculator modules, the asynchronous fifo was never utilized.

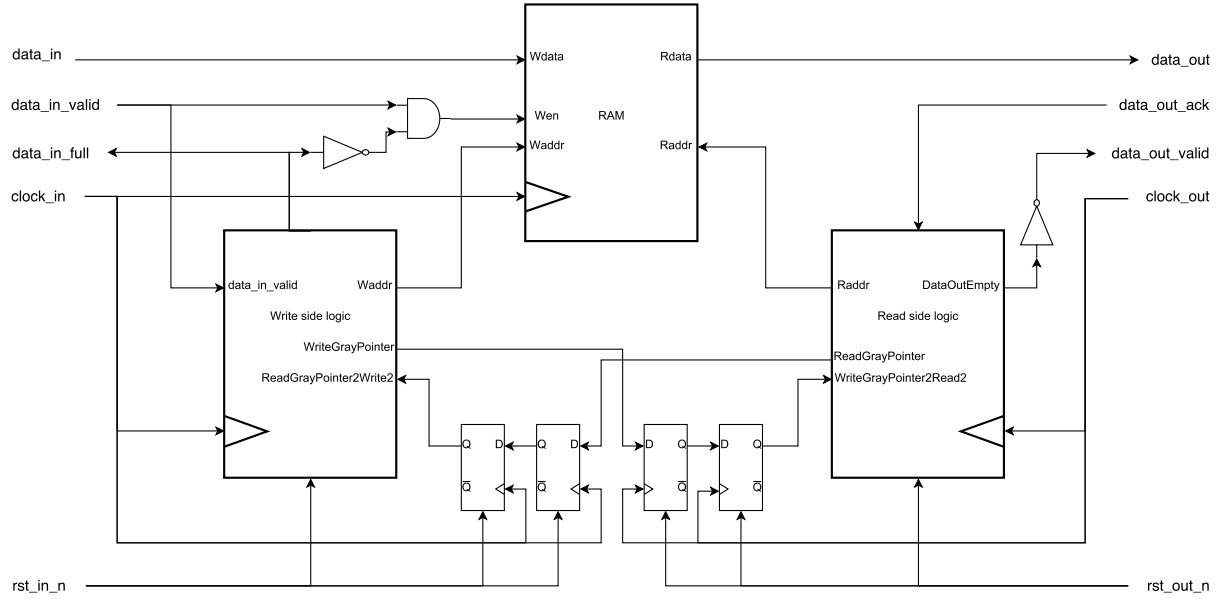


Figure 33: Asynchronous fifo.

The fifo design achieves glitch free asynchronous operation by using a proven method of passing address pointers using gray code.[2] Source code is provided in **Appendix C**. The buffer size as well as data width is implemented as fully customizable parameters. This would allow for writing the context index, BypassFlag, Bins data and Bins length to the same address in the buffer. The buffer is instantiated as block ram as specified in the Xilinx dual port block ram example.

9 Results and Discussion

9.1 Binarizer and Context Index Calculator

Far and away the most challenging part of the binarizer implementation is the coding of the ALRem syntax element. A disproportionate amount of effort was spent on the fruitless endeavor of finding an efficient way of computing it. In the end, the proposed FSM approach was able to achieve efficient binarization. Had this method been known at the start of the project, it would have freed up a substantial amount of time. Time that could possibly be spent on integrating a context index calculator with the binarizer. The analysis of the subject performed in this thesis should provide a good foundation for future work.

9.2 CABAC Hardware Encoder

The current architecture of the hardware encoder is a relatively low performance implementation. But with the framework it provides, it allows for the exploration of higher performance architectures. There exists a lot of research into hardware optimizations that could more easily be understood with the presence of a working design. Furthermore, there still remains work to be done on the design. Expansion of the context table is required. In addition to the many possible improvements that could be made to the handling of these contexts. Pipelining could also be implemented, but should be postponed until more research into the higher performance architectures has been performed.

9.3 Achieving Correctness

A lot of effort was put into achieving correctness of the design, resulting in the development of the software model. But any serious attempt at making any HEVC compliant hardware should utilize the HEVC HM TEST Model software. This would allow for tracing and in depth analysis of the data flow. Having this tool would be invaluable compared to only working with the standard document. Anything less will introduce some uncertainty about the actual correctness of the designs.

9.4 Future Work

While the actual CABAC coding algorithm may not leave much room left for improvements. The binarization schemes used are bound to see modifications with succeeding revisions of HEVC, or even complete reworks with the introduction of new standards. Leaving practical implementations aside, there exists many possibilities of studying these binarization methods. Being able to quantify the actual binarizer performance would however require a framework, such as the HEVC HM Test Model.

References

- [1] Joe Bertolami. Context adaptive binary arithmetic coding. <http://bertolami.com/index.php?engine=blog&content=posts&detail=arithmetic-coding>, January 2015.
- [2] Clifford E. Cummings. Synthesis and scripting techniques for designing multiasynchronous clock designs. <http://www.deepchip.com/downloads/cliffsnug01.pdf>, 2001.
- [3] Heiko Schwarz Detlev Marpe and Thomas Wiegand. Cabac context-based adaptive binary arithmetic coding in the h.264/avc video compression ieee csvt. <http://slideplayer.com/slide/5674258/>, July 2003.
- [4] <http://z3technology.com>. Hevc- what's next in video compression. <http://z3technology.com/News/HEVC-What-is-next-in-video-compression/26.html>.
- [5] in cites.com. An interview with dr. thomas wiegand. <http://www.in-cites.com/scientists/ThomasWiegand.html>, 2007.
- [6] Tomasz Grajek Krzysztof Wegner Jakub Siast Krzysztof Klimaszewski Olgierd Stankiewicz Marek Domaski Jakub Stankowski, Damian Karwowski. Bitrate distribution of syntax elements in the hevc encoded video. <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6948726>, 2014.
- [7] Anush Moorthy Anne Aaron Jan De Cock, Aditya Mavlankar. A large-scale comparison of x264, x265, and libvpxa sneak peek. <https://medium.com/netflix-techblog/a-large-scale-comparison-of-x264-x265-and-libvpx-a-sneak-peek-2e81e88f8b0f>, 2016.
- [8] Heiko Schwarz Thiow Keng Tan Jens-Rainer Ohm, Gary J. Sullivan and Thomas Wiegand. Comparison of the coding efficiency of video coding standards including high efficiency video coding (hevc). http://iphone.hhi.de/wiegand/assets/pdfs/2012_12_IEEE-HEVC-Performance.pdf, 2012.
- [9] Nguyen Nguyen Tianying Ji Marta Karczewicz Gordon Clare Felix Henry Joel Sole, Rajan Joshi and Alberto Duenas. Transform coefficient coding in hevc. <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6324418>, 2012.
- [10] TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU SG16. Rec. itu-t h.265 v4. <https://www.itu.int/rec/T-REC-H.265-201612-I/en>, 2017.
- [11] Vivienne Sze and Detlev Marpe. Entropy coding in hevc. high efficiency video coding (hevc): 209-274. <https://dspace.mit.edu/handle/1721.1/100315>, 2014.
- [12] Madhukar Budagavi Vivienne Sze. High throughput cabac entropy coding in hevc. <http://ieeexplore.ieee.org/document/6317157/?arnumber=6317157&tag=1>, 2012.
- [13] Wikipedia. High efficiency video coding implementations and products. https://en.wikipedia.org/wiki/High_Efficiency_Video_Coding_implementations_and_products, 2017.

- [14] [www.openhub.net](https://www.openhub.net/p/hevc/analyses/latest/languages_summary). Hecv test model(hm) analysis. https://www.openhub.net/p/hevc/analyses/latest/languages_summary, 2017.

Appendix A

```
1  -- *****
2  -- Binarizer for HEVC H.265
3  -- 01/03/2017
4  -- Norwegian University of Science and Technology
5  -- Lars Erik Songe Paulsen
6  -- *****
7
8  -- *****
9  -- TODO LIST:
10 -- *****
11 --
12 -- *****
13
14 --pragma synthesis_off
15 -- your simulation-only code
16 --pragma synthesis_on
17
18 library ieee;
19 library std;
20 use ieee.std_logic_1164.all;
21 use ieee.numeric_std.all;
22 use ieee.std_logic_unsigned.all;
23 use ieee.std_logic_misc.all; --
24   ↪ http://www1.pldworld.com/@wilina/html/technote/tool/manual/15i\_doc/fndtn/vhd/vhd10\_3.htm
25
26 library work;
27 use work.BinarizerParameters.all;
28
29 entity Binarizer is
30     port
31     (
32         Clk                : in std_logic;
33         DataIn              : in std_logic_vector((16*CoeffWidth)-1 downto 0);
34         ScanDir             : in std_logic_vector(1 downto 0);
35         Resetn              : in std_logic;
36         StartBinarizer      : in std_logic;
37         DataOut             : out std_logic_vector(OutputWidth-1 downto 0);
38         DataLength          : out std_logic_vector(OutputWidthLength-1 downto 0);
39         PrefixLength        : out std_logic_vector(OutputWidthLength-1 downto 0);
40         Finished            : out std_logic
41     );
42
43 end Binarizer;
44
45 architecture struct of Binarizer is
46
47     -- -----
48     -- Signal declarations
49     -- -----
50
51     type BinarizeStateType is(
52         read_input,
53         write_last_sig_coeff_x_prefix,
54         write_last_sig_coeff_y_prefix,
55         write_sig_coeff_flag,
56         write_coeff_abs_level_greater1,
57         write_coeff_abs_level_greater2,
58         sync,
59         write_coeff_abs_level_remaining,
60         write_coeff_sign_flag,
61         write_finished
62     );
63
64     type transform_block is array(15 downto 0) of integer range -(2**(coeffwidth-1)) to
65         ↪ (2**(coeffwidth-1));
66
67     signal BinarizeState : BinarizeStateType;
68
69
```

```

70     signal coefficients : Transform_Block;
71     signal coeff_abs_level_greater1_flag : std_logic_vector(0 to 15);
72     signal coeff_abs_level_greater2_flag : std_logic_vector(0 to 15);
73     signal sig_coeff_flag : std_logic_vector(0 to 15);
74     signal ABS_index : integer range 0 to 15;
75     signal ABS_started : std_logic;
76     signal ABS_done : std_logic;
77     signal ABS_level_writeout : integer range 0 to 32767;
78     signal k : integer range 0 to 4;
79
80 begin
81
82     Next_ABS : process(Clk, DataIn, Resetn, StartBinarizer)
83         variable ABS_index_cycler : integer range 0 to 15;
84
85     begin
86         if Resetn = '0' then
87             ABS_index_cycler := 0;
88             ABS_index <= 0;
89             ABS_done <= '0';
90         elsif rising_edge(Clk) then
91             case BinarizeState is
92                 when sync | write_coeff_abs_level_remaining =>
93                     find_next_ALG2_loop : for ABS_index_cycler in 0 to 15 loop
94                         if (ABS_index_cycler <= ABS_index) then --<<<<<<<< Here be issues
95                             --spin TODO FIX end condition
96                         else
97                             if(coeff_abs_level_greater1_flag(ABS_index_cycler) = '1') then
98                                 report "ALG found at ABS_index[" & integer'image(ABS_index_cycler) & "];"
99                                 ABS_index <= ABS_index_cycler;
100                                ABS_level_writeout <= coefficients(ABS_index_cycler);
101                                if(ABS_index_cycler = 15) then
102                                    ABS_done <= '1';
103                                else
104                                    exit find_next_ALG2_loop;
105                                end if;
106                            end if;
107                        end if;
108                    end loop;
109                when others =>
110                    ABS_index <= 0;
111                    ABS_done <= '0';
112            end case;
113        end if;
114    end process;
115
116    Binarize : process(Clk, DataIn, Resetn, StartBinarizer)
117
118        -- -----
119        -- Variable declarations
120        -- -----
121
122    variable sig_coeff_flag_found : std_logic;
123    variable sig_coeff_flag_index : integer range OutputWidth-1 downto OutputWidth-14;
124    variable coeff_sign_flag : std_logic_vector(0 to 15);
125    variable ALG1_index : integer range 0 to 8;
126    variable SIGN_index : integer range 0 to 15;
127    variable Scan_Direction : Scan_Directions := DiagonalScan; -- TODO: Add all scan directions
128
129    begin
130        if Resetn = '0' then
131            sig_coeff_flag <= (others => '0');
132            sig_coeff_flag_found := '0';
133            sig_coeff_flag_index := OutputWidth-1;
134            ALG1_index := 0;
135            SIGN_index := 0;
136            DataOut <= (others => '0');
137            DataLength <= (others => '0');
138            BinarizeState <= read_input;
139            Finished <= '1';
140            coeff_abs_level_greater1_flag <= (others => '0');
141            coeff_abs_level_greater2_flag <= (others => '0');
142            k <= 0;

```

```

143     PrefixLength <= (others => '0');
144
145     elsif rising_edge(Clk) then
146         case BinarizeState is
147             when read_input =>
148                 BinarizeState <= write_last_sig_coeff_x_prefix;
149                 Finished <= '0';
150
151                 for i in 0 to 15 loop
152                     -----
153                     -- Read absolute values.
154                     -----
155                     coefficients(Scan_Direction(15-i)) <=
156                         ↪ to_integer(abs(signed(DataIn(((CoeffWidth*(i+1))-1) downto
157                         ↪ (CoeffWidth*i)))));
158
159                     -----
160                     -- Read sign.
161                     -----
162                     coeff_sign_flag(Scan_Direction(15-i)) := DataIn((CoeffWidth*(i+1))-1);
163
164                     -----
165                     -- Find non-zero data.
166                     -----
167                     sig_coeff_flag(Scan_Direction(15-i)) <= or_reduce(DataIn(((CoeffWidth*(i+1))-2)
168                         ↪ downto (CoeffWidth*i)));
169
170                     -----
171                     -- Find >1 data and >2 data. Procedure dependent on sign bit.
172                     -----
173                     case DataIn((CoeffWidth*(i+1))-1) is
174                         when '0' =>
175                             coeff_abs_level_greater1_flag(Scan_Direction(15-i)) <=
176                                 ↪ or_reduce(DataIn(((CoeffWidth*(i+1))-2) downto
177                                 ↪ ((CoeffWidth*i)+1)));
178                             coeff_abs_level_greater2_flag(Scan_Direction(15-i)) <=
179                                 ↪ (DataIn(CoeffWidth*i) and DataIn((CoeffWidth*i)+1))
180                                 or
181                                 ↪ or_reduce(DataIn(((CoeffWidth*(i+1))-2)
182                                 ↪ downto ((CoeffWidth*i)+2)));
183
184                         when '1' =>
185                             coeff_abs_level_greater1_flag(Scan_Direction(15-i)) <=
186                                 ↪ nand_reduce(DataIn(((CoeffWidth*(i+1))-2) downto (CoeffWidth*i)));
187                             coeff_abs_level_greater2_flag(Scan_Direction(15-i)) <=
188                                 ↪ (DataIn(CoeffWidth*i) and (not(DataIn((CoeffWidth*i)+1))))
189                                 or
190                                 ↪ nand_reduce(DataIn(((CoeffWidth*(i+1))-2)
191                                 ↪ downto ((CoeffWidth*i)+2)));
192
193                     when others =>
194                         end case;
195                 end loop;
196
197             when write_last_sig_coeff_x_prefix =>
198                 for i in 0 to 15 loop
199                     report "coefficients(" & integer'image(i) & "): " &
200                         ↪ integer'image(coefficients(i));
201                 end loop;
202
203                 report "sig_coeff_flag: " & integer'image(conv_integer(sig_coeff_flag));
204                 report "coeff_abs_level_greater1_flag: " &
205                     ↪ integer'image(conv_integer(coeff_abs_level_greater1_flag));
206                 report "coeff_abs_level_greater2_flag: " &
207                     ↪ integer'image(conv_integer(coeff_abs_level_greater2_flag));
208                 report "coeff_sign_flag: " & integer'image(conv_integer(coeff_sign_flag));
209
210                 BinarizeState <= write_last_sig_coeff_y_prefix;
211                 write_last_sig_coeff_x_prefix_loop : for index in 0 to 15 loop
212                     if(sig_coeff_flag(index) = '1') then
213                         case index is
214                             when 0 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
215                             DataLength <=
216                                 ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));

```

```

200         when 1 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
201         DataLength <=
202             ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
203     when 2 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
204     DataLength <=
205         ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
206     when 3 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
207     DataLength <=
208         ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
209     when 4 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
210     DataLength <=
211         ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
212     when 5 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
213     DataLength <=
214         ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
215     when 6 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
216     DataLength <=
217         ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
218     when 7 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
219     DataLength <=
220         ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
221     when 8 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
222     DataLength <=
223         ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
224     when 9 => DataOut(OutputWidth-1) <= '0';
225     DataLength <=
226         ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
227     when 10 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
228     DataLength <=
229         ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
230     when 11 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
231     DataLength <=
232         ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
233     when 12 => DataOut(OutputWidth-1) <= '0';
234     DataLength <=
235         ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
236     when 13 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
237     DataLength <=
238         ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
239     when 14 => DataOut(OutputWidth-1) <= '0';
240     DataLength <=
241         ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
242     when 15 => DataOut(OutputWidth-1) <= '0';
243     DataLength <=
244         ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
245     end case;
246     exit write_last_sig_coeff_x_prefix_loop;
247 end if;
248 end loop;
249 when write_last_sig_coeff_y_prefix =>
250     BinarizeState <= write_sig_coeff_flag;
251     write_last_sig_coeff_y_prefix_loop : for index in 0 to 15 loop
252         if(sig_coeff_flag(index) = '1') then
253             case index is
254                 when 0 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
255                 DataLength <=
256                     ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
257                 when 1 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
258                 DataLength <=
259                     ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
260                 when 2 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
261                 DataLength <=
262                     ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
263                 when 3 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
264                 DataLength <=
265                     ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
266                 when 4 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
267                 DataLength <=
268                     ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
269                 when 5 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
270                 DataLength <=
271                     ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
272                 when 6 => DataOut(OutputWidth-1) <= '0';

```

```

252         DataLength <=
253             ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
254     when 7 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
255         DataLength <=
256             ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
257     when 8 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
258         DataLength <=
259             ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
260     when 9 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
261         DataLength <=
262             ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
263     when 10 => DataOut(OutputWidth-1) <= '0';
264         DataLength <=
265             ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
266     when 11 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
267         DataLength <=
268             ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
269     when 12 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
270         DataLength <=
271             ↪ std_logic_vector(to_unsigned(3,OutputWidthLength));
272     when 13 => DataOut(OutputWidth-1) <= '0';
273         DataLength <=
274             ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
275     when 14 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
276         DataLength <=
277             ↪ std_logic_vector(to_unsigned(2,OutputWidthLength));
278     when 15 => DataOut(OutputWidth-1) <= '0';
279         DataLength <=
280             ↪ std_logic_vector(to_unsigned(1,OutputWidthLength));
281     end case;
282     exit write_last_sig_coeff_y_prefix_loop;
283 end if;
284 end loop;
285 when write_sig_coeff_flag =>
286     BinarizeState <= write_coeff_abs_level_greater1;
287     write_sig_coeff_flag_loop : for index in 0 to 15 loop
288         if sig_coeff_flag_found = '1' then
289             DataOut(sig_coeff_flag_index) <= sig_coeff_flag(index);
290             sig_coeff_flag_index := sig_coeff_flag_index - 1;
291             if(sig_coeff_flag(index) = '1') then
292                 coefficients(index) <= coefficients(index) - 1;
293             end if;
294         elsif(sig_coeff_flag(index) = '1') then
295             sig_coeff_flag_found := '1';
296             DataLength <= std_logic_vector(to_unsigned(15-index,OutputWidthLength));
297             coefficients(index) <= coefficients(index) - 1;
298         end if;
299     end loop;
300 when write_coeff_abs_level_greater1 =>
301     BinarizeState <= write_coeff_abs_level_greater2;
302     write_coeff_abs_level_greater1_loop : for index in 0 to 15 loop
303         if sig_coeff_flag(index) = '1' and ALG1_index < 8 then
304             if( coeff_abs_level_greater1_flag(index) = '1') then
305                 DataOut(OutputWidth-1-ALG1_index) <= '1';
306                 report "ALG1:coefficients(" & integer'image(index) & "): " &
307                     ↪ integer'image(coefficients(index));
308                 coefficients(index) <= coefficients(index) - 1;
309                 report "ALG1:coefficients(" & integer'image(index) & "): " &
310                     ↪ integer'image(coefficients(index));
311             else
312                 DataOut(OutputWidth-1-ALG1_index) <= '0';
313             end if;
314             ALG1_index := ALG1_index + 1;
315         end if;
316     end loop;
317     DataLength <= std_logic_vector(to_unsigned(ALG1_index,OutputWidthLength));
318 when write_coeff_abs_level_greater2 =>
319     BinarizeState <= sync;
320     write_coeff_abs_level_greater2_loop : for index in 0 to 15 loop
321         if sig_coeff_flag(index) = '1' then

```



```

313         if (coeff_abs_level_greater2_flag(index) = '1') then
314             DataOut(OutputWidth-1) <= '1';
315             report "ALG2:coefficients(" & integer'image(index) & "): " &
                 ↳ integer'image(coefficients(index));
316             coefficients(index) <= coefficients(index) - 1;
317             report "ALG2:coefficients(" & integer'image(index) & "): " &
                 ↳ integer'image(coefficients(index));
318         else
319             DataOut(OutputWidth-1) <= '0';
320         end if;
321         DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
322         exit write_coeff_abs_level_greater_2_loop;
323     end if;
324 end loop;
325
326 when sync =>
327     BinarizeState <= write_coeff_abs_level_remaining;
328
329     for i in 0 to 15 loop
330         report "coefficients(" & integer'image(i) & "): " &
                 ↳ integer'image(coefficients(i));
331     end loop;
332
333 when write_coeff_abs_level_remaining =>
334     report "write_coeff_abs_level_remaining: @ k: " & integer'image(k) & "
                 ↳ ABS_level_writeout: " & integer'image(ABS_level_writeout);
335     if(ABS_done = '1') then
336         BinarizeState <= write_coeff_sign_flag;
337     end if;
338     case k is
339     when 0 =>
340         case (ABS_level_writeout) is
341         when 0 =>
342             DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
343             PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
344         when 1 =>
345             DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
346             PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
347         when 2 =>
348             DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
349             PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
350         when 3 =>
351             DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
352             PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
353         when 4 to 5 =>
354             DataOut(OutputWidth-1 downto OutputWidth-1) <=
                 ↳ std_logic_vector(to_unsigned(ABS_level_writeout-4,1));
355             DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
356             PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
357             k <= 1;
358         when 6 to 9 =>
359             DataOut(OutputWidth-1 downto OutputWidth-2) <=
                 ↳ std_logic_vector(to_unsigned(ABS_level_writeout-6,2));
360             DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
361             PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
362         when 10 to 17 =>
363             DataOut(OutputWidth-1 downto OutputWidth-3) <=
                 ↳ std_logic_vector(to_unsigned(ABS_level_writeout-10,3));
364             DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
365             PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
366         when 18 to 33 =>
367             DataOut(OutputWidth-1 downto OutputWidth-4) <=
                 ↳ std_logic_vector(to_unsigned(ABS_level_writeout-18,4));
368             DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
369             PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
370         when 34 to 65 =>
371             DataOut(OutputWidth-1 downto OutputWidth-5) <=
                 ↳ std_logic_vector(to_unsigned(ABS_level_writeout-34,5));
372             DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
373             PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
374         when 66 to 129 =>
375             DataOut(OutputWidth-1 downto OutputWidth-6) <=
                 ↳ std_logic_vector(to_unsigned(ABS_level_writeout-66,6));

```

```

376         DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
377         PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
378     when 130 to 257 =>
379         DataOut(OutputWidth-1 downto OutputWidth-7) <=
380             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-130,7));
381         DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
382         PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
383     when 258 to 513 =>
384         DataOut(OutputWidth-1 downto OutputWidth-8) <=
385             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-258,8));
386         DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
387         PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
388     when 514 to 1025 =>
389         DataOut(OutputWidth-1 downto OutputWidth-9) <=
390             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-514,9));
391         DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
392         PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
393     when 1026 to 2049 =>
394         DataOut(OutputWidth-1 downto OutputWidth-10) <=
395             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-1026,10));
396         DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
397         PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
398     when 2050 to 4097 =>
399         DataOut(OutputWidth-1 downto OutputWidth-11) <=
400             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-2050,11));
401         DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
402         PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
403     when 4098 to 8193 =>
404         DataOut(OutputWidth-1 downto OutputWidth-12) <=
405             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4098,12));
406         DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
407         PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
408     when 8194 to 16385 =>
409         DataOut(OutputWidth-1 downto OutputWidth-13) <=
410             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8194,13));
411         DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
412         PrefixLength <= std_logic_vector(to_unsigned(17,OutputWidthLength));
413     when 16386 to 32767 =>
414         DataOut(OutputWidth-1 downto OutputWidth-14) <=
415             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16386,14));
416         DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
417         PrefixLength <= std_logic_vector(to_unsigned(18,OutputWidthLength));
418     when others =>
419     end case;
420 when 1 =>
421     case (ABS_level_writeout) is
422     when 0 to 1 =>
423         DataOut(OutputWidth-1 downto OutputWidth-1) <=
424             ↪ std_logic_vector(to_unsigned(ABS_level_writeout,1));
425         DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
426         PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
427     when 2 to 3 =>
428         DataOut(OutputWidth-1 downto OutputWidth-1) <=
429             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-2,1));
430         DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
431         PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
432     when 4 to 5 =>
433         DataOut(OutputWidth-1 downto OutputWidth-1) <=
434             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4,1));
435         DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
436         PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
437     when 6 to 7 =>
438         DataOut(OutputWidth-1 downto OutputWidth-1) <=
439             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-6,1));
440         DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
441         PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
442     when 8 to 11 =>
443         DataOut(OutputWidth-1 downto OutputWidth-2) <=
444             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8,2));
445         DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
446         PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
447     when 12 to 19 =>
448         DataOut(OutputWidth-1 downto OutputWidth-3) <=
449             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-12,3));

```

```

436         DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
437         PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
438     when 20 to 35 =>
439         DataOut(OutputWidth-1 downto OutputWidth-4) <=
440             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-20,4));
441         DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
442         PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
443     when 36 to 67 =>
444         DataOut(OutputWidth-1 downto OutputWidth-5) <=
445             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-36,5));
446         DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
447         PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
448     when 68 to 131 =>
449         DataOut(OutputWidth-1 downto OutputWidth-6) <=
450             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-68,6));
451         DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
452         PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
453     when 132 to 259 =>
454         DataOut(OutputWidth-1 downto OutputWidth-7) <=
455             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-132,7));
456         DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
457         PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
458     when 260 to 515 =>
459         DataOut(OutputWidth-1 downto OutputWidth-8) <=
460             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-260,8));
461         DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
462         PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
463     when 516 to 1027 =>
464         DataOut(OutputWidth-1 downto OutputWidth-9) <=
465             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-516,9));
466         DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
467         PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
468     when 1028 to 2051 =>
469         DataOut(OutputWidth-1 downto OutputWidth-10) <=
470             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-1028,10));
471         DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
472         PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
473     when 2052 to 4099 =>
474         DataOut(OutputWidth-1 downto OutputWidth-11) <=
475             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-2052,11));
476         DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
477         PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
478     when 4100 to 8195 =>
479         DataOut(OutputWidth-1 downto OutputWidth-12) <=
480             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4100,12));
481         DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
482         PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
483     when 8196 to 16387 =>
484         DataOut(OutputWidth-1 downto OutputWidth-13) <=
485             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8196,13));
486         DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
487         PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
488     when 16388 to 32767 =>
489         DataOut(OutputWidth-1 downto OutputWidth-14) <=
490             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16388,14));
491         DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
492         PrefixLength <= std_logic_vector(to_unsigned(17,OutputWidthLength));
493     when others =>
494     end case;
495 when 2 =>
496     case (ABS_level_writeout) is
497     when 0 to 3 =>
498         DataOut(OutputWidth-1 downto OutputWidth-2) <=
499             ↪ std_logic_vector(to_unsigned(ABS_level_writeout,2));
500         DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
501         PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
502     when 4 to 7 =>
503         DataOut(OutputWidth-1 downto OutputWidth-2) <=
504             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4,2));
505         DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
506         PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
507     when 8 to 11 =>
508         DataOut(OutputWidth-1 downto OutputWidth-2) <=
509             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8,2));

```

```

496         DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
497         PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
498     when 12 to 15 =>
499         DataOut(OutputWidth-1 downto OutputWidth-2) <=
500             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-12,2));
501         DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
502         PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
503     when 16 to 23 =>
504         DataOut(OutputWidth-1 downto OutputWidth-3) <=
505             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16,3));
506         DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
507         PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
508     when 24 to 39 =>
509         DataOut(OutputWidth-1 downto OutputWidth-4) <=
510             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-24,4));
511         DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
512         PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
513     when 40 to 71 =>
514         DataOut(OutputWidth-1 downto OutputWidth-5) <=
515             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-40,5));
516         DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
517         PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
518     when 72 to 135 =>
519         DataOut(OutputWidth-1 downto OutputWidth-6) <=
520             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-72,6));
521         DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
522         PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
523     when 136 to 263 =>
524         DataOut(OutputWidth-1 downto OutputWidth-7) <=
525             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-136,7));
526         DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
527         PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
528     when 264 to 519 =>
529         DataOut(OutputWidth-1 downto OutputWidth-8) <=
530             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-264,8));
531         DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
532         PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
533     when 520 to 1031 =>
534         DataOut(OutputWidth-1 downto OutputWidth-9) <=
535             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-520,9));
536         DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
537         PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
538     when 1032 to 2055 =>
539         DataOut(OutputWidth-1 downto OutputWidth-10) <=
540             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-1032,10));
541         DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
542         PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
543     when 2056 to 4103 =>
544         DataOut(OutputWidth-1 downto OutputWidth-11) <=
545             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-2056,11));
546         DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
547         PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
548     when 4104 to 8199 =>
549         DataOut(OutputWidth-1 downto OutputWidth-12) <=
550             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4104,12));
551         DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
552         PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
553     when 8200 to 16391 =>
554         DataOut(OutputWidth-1 downto OutputWidth-13) <=
555             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8200,13));
556         DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
557         PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
558     when 16392 to 32767 =>
559         DataOut(OutputWidth-1 downto OutputWidth-14) <=
560             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16392,14));
561         DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
562         PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
563     when others =>
564     end case;
565 when 3 =>
566     case (ABS_level_writeout) is
567     when 0 to 7 =>
568         DataOut(OutputWidth-1 downto OutputWidth-3) <=
569             ↪ std_logic_vector(to_unsigned(ABS_level_writeout,3));

```

```

556         DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
557         PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
558     when 8 to 15 =>
559         DataOut(OutputWidth-1 downto OutputWidth-3) <=
560             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8,3));
561         DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
562         PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
563     when 16 to 23 =>
564         DataOut(OutputWidth-1 downto OutputWidth-3) <=
565             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16,3));
566         DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
567         PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
568     when 24 to 31 =>
569         DataOut(OutputWidth-1 downto OutputWidth-3) <=
570             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-24,3));
571         DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
572         PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
573     when 32 to 47 =>
574         DataOut(OutputWidth-1 downto OutputWidth-4) <=
575             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-32,4));
576         DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
577         PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
578     when 48 to 79 =>
579         DataOut(OutputWidth-1 downto OutputWidth-5) <=
580             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-48,5));
581         DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
582         PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
583     when 80 to 143 =>
584         DataOut(OutputWidth-1 downto OutputWidth-6) <=
585             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-80,6));
586         DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
587         PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
588     when 144 to 271 =>
589         DataOut(OutputWidth-1 downto OutputWidth-7) <=
590             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-144,7));
591         DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
592         PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
593     when 272 to 527 =>
594         DataOut(OutputWidth-1 downto OutputWidth-8) <=
595             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-272,8));
596         DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
597         PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
598     when 528 to 1039 =>
599         DataOut(OutputWidth-1 downto OutputWidth-9) <=
600             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-528,9));
601         DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
602         PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
603     when 1040 to 2063 =>
604         DataOut(OutputWidth-1 downto OutputWidth-10) <=
605             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-1040,10));
606         DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
607         PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
608     when 2064 to 4111 =>
609         DataOut(OutputWidth-1 downto OutputWidth-11) <=
610             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-2064,11));
611         DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
612         PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
613     when 4112 to 8207 =>
614         DataOut(OutputWidth-1 downto OutputWidth-12) <=
615             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4112,12));
616         DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
617         PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
618     when 8208 to 16399 =>
619         DataOut(OutputWidth-1 downto OutputWidth-13) <=
620             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8208,13));
621         DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
622         PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
623     when 16400 to 32767 =>
624         DataOut(OutputWidth-1 downto OutputWidth-14) <=
625             ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16400,14));
626         DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
627         PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
628     when others =>

```



```

615         end case;
616     when 4 =>
617         case (ABS_level_writeout) is
618             when 0 to 15 =>
619                 DataOut(OutputWidth-1 downto OutputWidth-4) <=
620                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout,4));
621                 DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
622                 PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
623             when 16 to 31 =>
624                 DataOut(OutputWidth-1 downto OutputWidth-4) <=
625                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16,4));
626                 DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
627                 PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
628             when 32 to 47 =>
629                 DataOut(OutputWidth-1 downto OutputWidth-4) <=
630                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-32,4));
631                 DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
632                 PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
633             when 48 to 63 =>
634                 DataOut(OutputWidth-1 downto OutputWidth-4) <=
635                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-48,4));
636                 DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
637                 PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
638             when 64 to 95 =>
639                 DataOut(OutputWidth-1 downto OutputWidth-5) <=
640                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-64,5));
641                 DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
642                 PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
643             when 96 to 159 =>
644                 DataOut(OutputWidth-1 downto OutputWidth-6) <=
645                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-96,6));
646                 DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
647                 PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
648             when 160 to 287 =>
649                 DataOut(OutputWidth-1 downto OutputWidth-7) <=
650                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-160,7));
651                 DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
652                 PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
653             when 288 to 543 =>
654                 DataOut(OutputWidth-1 downto OutputWidth-8) <=
655                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-288,8));
656                 DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
657                 PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
658             when 544 to 1055 =>
659                 DataOut(OutputWidth-1 downto OutputWidth-9) <=
660                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-544,9));
661                 DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
662                 PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
663             when 1056 to 2079 =>
664                 DataOut(OutputWidth-1 downto OutputWidth-10) <=
665                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-1056,10));
666                 DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
667                 PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
668             when 2080 to 4127 =>
669                 DataOut(OutputWidth-1 downto OutputWidth-11) <=
670                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-2080,11));
671                 DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
672                 PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
673             when 4128 to 8223 =>
674                 DataOut(OutputWidth-1 downto OutputWidth-12) <=
675                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-4128,12));
676                 DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
677                 PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
678             when 8224 to 16415 =>
679                 DataOut(OutputWidth-1 downto OutputWidth-13) <=
680                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-8224,13));
681                 DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
682                 PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
683             when 16416 to 32767 =>
684                 DataOut(OutputWidth-1 downto OutputWidth-14) <=
685                     ↪ std_logic_vector(to_unsigned(ABS_level_writeout-16416,14));
686                 DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
687                 PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));

```

```

674         when others      =>
675             end case;
676     end case;
677
678     if(k<4 and (ABS_level_writeout) > (3*(2**k))) then
679         k <= k+1;
680     end if;
681
682     when write_coeff_sign_flag =>
683         BinarizeState <= write_finished;
684         write_coeff_sign_flag_loop : for index in 0 to 15 loop
685             if sig_coeff_flag(index) = '1' then
686                 if coeff_sign_flag(index) = '1' then
687                     DataOut(OutputWidth-1 - SIGN_index) <= '1';
688                 else
689                     DataOut(OutputWidth-1 - SIGN_index) <= '0';
690                 end if;
691                 SIGN_index := SIGN_index + 1;
692             end if;
693             DataLength <= std_logic_vector(to_unsigned(SIGN_index,OutputWidthLength));
694         end loop;
695     when others =>
696         BinarizeState <= write_finished;
697         Finished <= '1';
698         DataOut      <= (others => '0');
699         DataLength    <= (others => '0');
700     end case;
701 end if;
702 end process;
703 end struct;

```

Appendix B

```
1  -- *****
2  -- CABAC_Enc coder for HEVC h.265
3  -- 01/03/2017
4  -- Norwegian University of Science and Technology
5  -- Lars Erik Songe Paulsen
6  -- *****
7
8  -- *****
9  -- TODO LIST:
10 -- *****
11 -- BinCountInNALUnits
12 -- bitsOutstanding overflow
13 -- Replace ctxIdxTableInitials with calculation
14 -- Proper memory interfacing for tables
15 -- *****
16
17 library ieee;
18 use      ieee.std_logic_1164.all;
19 use      ieee.std_logic_arith.ALL;
20 use      ieee.numeric_std.all;
21 use      ieee.std_logic_unsigned.all;
22 use      ieee.std_logic_misc.all;
23
24 library std;
25 use      std.textio.all;
26
27 library work;
28 use      work.CABAC_EncParameters.all;
29
30 entity CABAC_Enc is
31
32     port
33     (
34         Clk          : in  std_logic;
35         Input         : in  std_logic_vector(InputW-1 downto 0);
36         InputLen      : in  std_logic_vector(InputWLen-1 downto 0);
37         ctxIdx        : in  std_logic_vector(6 downto 0);
38         SliceQPY      : in  std_logic_vector(5 downto 0);
39         initType      : in  std_logic_vector(1 downto 0);
40         Resetn        : in  std_logic;
41         Start         : in  std_logic;
42         Output        : out std_logic_vector(OutputW-1 downto 0);
43         OutputLen     : out std_logic_vector(OutputWLen-1 downto 0);
44         BypassI       : in  std_logic;
45         Bypass0       : out std_logic;
46         TermI         : in  std_logic;
47         Term0         : out std_logic;
48         Finished      : out std_logic
49     );
50
51 end CABAC_Enc;
52
53 architecture struct of CABAC_Enc is
54
55     -- -----
56     -- Type declarations
57     -- -----
58
59     -- States
60     type CABAC_EncStateType is(
61         r_Input,
62         init_ctxTbl,
63         r_ctx,
64         enc_bin_r,
65         enc_bin_b,
66         RenormE,
67         PutBit,
68         w_ctx,
69         enc_Term,
70         w_finished
71     );
```



```

72
73 -- Table types
74 type transIdx_t      is array(0 to 63)                of std_logic_vector(5 downto 0);
75 type qRange_t        is array(0 to 3)                of std_logic_vector(7 downto 0);
76 type rangeTabLps_t   is array(0 to 63)                of qRange_t;
77 type ctxTblInit_t     is array(0 to (52*3*ctxIdxRange)-1) of std_logic_vector(6 downto 0);
78 type ctxTbl_t         is array(0 to ctxIdxRange-1)      of std_logic_vector(6 downto 0);
79
80 -----
81 -- Functions
82 -----
83 function string_to_binary(inp: string) return std_logic_vector is
84     variable temp: std_logic_vector(inp'length-1 downto 0) := (others => 'X');
85 begin
86     for i in inp'range loop
87         case inp(i) is
88             when '0' => temp(i-1) := '0';
89             when '1' => temp(i-1) := '1';
90             when others => temp(i-1) := 'X';
91         end case;
92     end loop;
93     return temp;
94 end function string_to_binary;
95
96 impure function InitctxTbl (RomFileName : in string) return ctxTblInit_t is
97     FILE romfile : text is in RomFileName;
98     variable RomFileLine : line;
99     variable rom : ctxTblInit_t;
100     variable TestString : string(7 downto 1);
101 begin
102     for i in ctxTblInit_t'range loop
103         readline(romfile, RomFileLine);
104         read(RomFileLine, TestString);
105         rom(i) := string_to_binary(TestString)(6 downto 0);
106     end loop;
107     return rom;
108 end function;
109
110 impure function InitttransIdx (RomFileName : in string) return transIdx_t is
111     FILE romfile : text is in RomFileName;
112     variable RomFileLine : line;
113     variable rom : transIdx_t;
114     variable TestString : string(8 downto 1);
115 begin
116     for i in 0 to 63 loop
117         readline(romfile, RomFileLine);
118         read(RomFileLine, TestString);
119         rom(i) := string_to_binary(TestString)(5 downto 0);
120     end loop;
121     return rom;
122 end function;
123
124 impure function InitrangTabLps (RomFileName : in string) return rangeTabLps_t is
125     FILE romfile : text is in RomFileName;
126     variable RomFileLine : line;
127     variable rom : rangeTabLps_t;
128     variable TestString : string(8 downto 1);
129 begin
130     for i in 0 to 3 loop
131         for j in 0 to 63 loop
132             readline(romfile, RomFileLine);
133             read(RomFileLine, TestString);
134             rom(j)(i) := string_to_binary(TestString)(7 downto 0);
135         end loop;
136     end loop;
137     return rom;
138 end function;
139
140 -----
141 -- Signal declarations
142 -----
143
144 -----

```

```

145 -- Signal declarations
146 -----
147 signal CABAC_EncState          : CABAC_EncStateType;
148 signal currCtx                 : std_logic_vector(6 downto 0);
149 signal initTbl                 : integer range 0 to ctxIdxRange-1;
150
151 -- Tables
152 signal rangeTabLPS             : rangeTabLps_t := InitrangeTabLps("CABAC_Enc_Tables\rangeTabLPS.txt");
153 signal transIdxLPS             : transIdx_t := InittransIdx ("CABAC_Enc_Tables\transIdxLPS.txt");
154 signal transIdxMPS             : transIdx_t := InittransIdx ("CABAC_Enc_Tables\transIdxMPS.txt");
155 signal ctxIdxTableInitials     : ctxTblInit_t := InitctxTbl
    ↪ ("CABAC_Enc_Tables\ctxIdxTableInitials.txt");
156 signal ctxIdxTable             : ctxTbl_t;
157
158 begin
159
160 -----
161 ---- Debugger process
162 -----
163 --Debugger : process(Resetn)
164 --begin
165 --if rising_edge(Resetn) then
166 --    for i in 0 to 63 loop
167 --        report "index: " & integer'image(i) &
168 --        " rangeTabLPS: " & integer'image(conv_integer(rangeTabLPS(i)(0))) &
169 --        " " & integer'image(conv_integer(rangeTabLPS(i)(1))) &
170 --        " " & integer'image(conv_integer(rangeTabLPS(i)(2))) &
171 --        " " & integer'image(conv_integer(rangeTabLPS(i)(3))) &
172 --        " transIdxLPS: " & integer'image(conv_integer(transIdxLPS(i))) &
173 --        " transIdxMPS: " & integer'image(conv_integer(transIdxMPS(i)));
174 --    end loop;
175 --    for i in 0 to 17939 loop
176 --        report "index: " & integer'image(i) &
177 --        " ctxIdxTableInitials: " & integer'image(conv_integer(ctxIdxTableInitials(i)));
178 --    end loop;
179 --end if;
180 --end process;
181
182 -----
183 -- ctxIdxTable interfacing process
184 -----
185 ctxIdxTableLookups : process(Clk)
186 begin
187     if rising_edge(Clk) then
188         case CABAC_EncState is
189             when init_ctxTbl => -- TODO verify full table is loaded
190                 -- Read context table initial values with the correct offset
191                 ctxIdxTable(initTbl)
192                     <= ctxIdxTableInitials((conv_integer(SliceQPY)*(ctxIdxRange*3))
193                     +(conv_integer(initType)*ctxIdxRange)+initTbl);
194             when r_ctx =>
195                 -- Store current context in working register
196                 -- TODO: Figure out why this needs to in CABAC_Enc Process.
197             when w_ctx =>
198                 -- Update context from working register
199                 ctxIdxTable(conv_integer(ctxIdx)) <= currCtx;
200             when others =>
201                 end case;
202         end if;
203     end process;
204
205 -----
206 -- CABAC_Enc coding main process
207 -----
208 CABAC_Enc : process(Clk, Input, initType, SliceQPY, BypassI, Resetn, Start, ctxIdx, TermI)
209
210 -----
211 -- Variable declarations
212 -----
213
214 -- Encoding vals
215 variable ivlLow : std_logic_vector(10 downto 0);--unsigned(10 downto 0);
216 variable ivlCurrRange : std_logic_vector(8 downto 0);--unsigned(8 downto 0);

```

```

217 variable ivlLpsRange : std_logic_vector(7 downto 0);
218 variable qRangeIdx : std_logic_VECTOR(0 to 1);
219
220 -- binVals
221 variable bins : std_logic_vector(InputW-1 downto 0);
222 variable binValI : integer range 0 to InputW-1;
223 variable binsLen : std_logic_vector(InputWLen-1 downto 0);
224
225 -- PutBit variables
226 variable PutBitVal : std_logic;
227 variable PutBitI : integer range OutputW-1 downto 0;
228 variable bitsOutstanding : integer range 0 to OutputW-1;
229 variable firstBitFlag : std_logic;
230
231 variable Flushed : std_logic_vector(1 downto 0);
232
233 variable InitFlag : std_logic;
234
235 begin
236   if Resetn = '0' then
237     currCtx          <= (others => '0');
238     Output           <= (others => '0');
239     OutputLen        <= (others => '0');
240     CABAC_EncState   <= r_Input;
241     Finished         <= '1';
242     Term0            <= '0';
243     Bypass0         <= '0';
244     InitFlag        := '0';
245     Flushed         := "00";
246     initTbl         <= 0;
247     ivlLow          := (others => '0'); --0
248     ivlCurrRange    := "111111110"; --510
249     firstBitFlag    := '1';
250     bitsOutstanding := 0;
251     qRangeIdx       := (others => '0');
252     PutBitI         := 0;
253   elsif rising_edge(Clk) then
254     case CABAC_EncState is
255       when r_Input =>
256         if (Start = '1') then
257           if (InitFlag = '0') then
258             CABAC_EncState <= init_ctxTbl;
259           elsif (TermI = '1') then
260             CABAC_EncState <= enc_Term;
261           else
262             if (BypassI = '1') then
263               CABAC_EncState <= enc_bin_b;
264             else
265               CABAC_EncState <= r_ctx;
266             end if;
267           end if;
268           Bypass0 <= BypassI;
269           bins    := Input;
270           binsLen := InputLen;
271           Finished <= '0';
272           binValI  := InputW-1;
273         end if;
274       when init_ctxTbl =>
275         if (initTbl = (ctxIdxRange-1)) then
276           if (BypassI = '1') then
277             CABAC_EncState <= enc_bin_b;
278           else
279             CABAC_EncState <= r_ctx;
280           end if;
281           InitFlag := '1';
282         else
283           initTbl <= initTbl + 1;
284         end if;
285       when r_ctx =>
286         CABAC_EncState <= enc_bin_r;
287         currCtx <= ctxIdxTable(conv_integer(ctxIdx));
288       when enc_bin_b =>
289         if (binValI >= (InputW-conv_integer(binsLen))) then

```

```

290     ivlLow := ivlLow(9 downto 0) & "0";
291     if (bins(binValI) /= '0') then
292         ivlLow := ivlLow + ivlCurrRange;
293     end if;
294     if (ivlLow >= 1024) then
295         PutBitVal := '1';
296         CABAC_EncState <= PutBit;
297         ivlLow := ivlLow - 1024;
298     elsif (ivlLow < 512) then
299         PutBitVal := '0';
300         CABAC_EncState <= PutBit;
301     else
302         ivlLow := ivlLow - 512;
303         bitsOutstanding := bitsOutstanding + 1;
304     end if;
305     binValI := binValI - 1;
306 else
307     OutputLen <= std_logic_vector(to_unsigned(PutBitI, OutputWLen));
308     CABAC_EncState <= w_ctx;
309 end if;
310 when enc_bin_r =>
311     if (binValI >= (InputW - conv_integer(binsLen))) then
312         qRangeIdx := ivlCurrRange(7 downto 6);
313         ivlLpsRange := rangeTabLPS(conv_integer(currCtx(5 downto
314             ↪ 0)))(conv_integer(qRangeIdx));
315         ivlCurrRange := ivlCurrRange - ivlLpsRange;
316         if (bins(binValI) /= currCtx(6)) then
317             ivlLow := ivlLow + ivlCurrRange;
318             ivlCurrRange := "0" & ivlLpsRange;
319             if (currCtx(5 downto 0) = "000000") then
320                 currCtx(6) <= not currCtx(6);
321             end if;
322             currCtx(5 downto 0) <= transIdxLPS(conv_integer(currCtx(5 downto 0)));
323         else
324             currCtx(5 downto 0) <= transIdxMPS(conv_integer(currCtx(5 downto 0)));
325         end if;
326         binValI := binValI - 1;
327         CABAC_EncState <= RenormE;
328     else
329         OutputLen <= std_logic_vector(to_unsigned(PutBitI, OutputWLen));
330         CABAC_EncState <= w_ctx;
331     end if;
332 when RenormE =>
333     if (ivlCurrRange < 256) then
334         if (ivlLow < 256) then
335             PutBitVal := '0';
336             CABAC_EncState <= PutBit;
337         elsif (ivlLow >= 512) then
338             ivlLow := ivlLow - 512;
339             PutBitVal := '1';
340             CABAC_EncState <= PutBit;
341         else
342             ivlLow := ivlLow - 256;
343             bitsOutstanding := bitsOutstanding + 1;
344             ivlCurrRange := ivlCurrRange(7 downto 0) & "0";
345             ivlLow := ivlLow(9 downto 0) & "0";
346         end if;
347     else
348         if (Flushed = "11") then
349             OutputLen <= std_logic_vector(to_unsigned(PutBitI, OutputWLen));
350             CABAC_EncState <= w_finished;
351         elsif (Flushed = "01") then
352             Flushed := "10";
353             PutBitVal := ivlLow(9);
354             CABAC_EncState <= PutBit;
355         else
356             CABAC_EncState <= enc_bin_r;
357         end if;
358     end if;
359 when PutBit =>
360     if (firstBitFlag /= '0') then
361         firstBitFlag := '0';
362     else

```

```

362         Output((OutputW-1)-PutBitI) <= PutBitVal;
363         PutBitI := PutBitI + 1;
364     end if;
365     PutBit_loop : for i in 0 to PutBitLoopLen-1 loop -- TODO: Potential overflow here if
        ↪ bitsOutstanding > PutBitLoopLen
366     if (bitsOutstanding > 0) then
367         Output((OutputW-1)-PutBitI) <= not PutBitVal;
368         bitsOutstanding := bitsOutstanding - 1;
369         PutBitI := PutBitI + 1;
370     else
371         if(Flushed = "10") then
372             Output((OutputW-1)-PutBitI downto (OutputW-1)-PutBitI-1) <= ivlLow(8) &
                ↪ '1';
373             OutputLen <= std_logic_vector(to_unsigned(PutBitI+2,OutputWLen));
374             CABAC_EncState <= w_finished;
375         elsif(BypassI = '1') then
376             CABAC_EncState <= enc_bin_b;
377         else
378             ivlCurrRange := ivlCurrRange(7 downto 0) & "0";
379             ivlLow := ivlLow(9 downto 0) & "0";
380             CABAC_EncState <= RenormE;
381         end if;
382         exit PutBit_loop;
383     end if;
384 end loop;
385 when w_ctx =>
386     CABAC_EncState <= r_Input;
387     PutBitI := 0;
388     Finished <= '1';
389 when enc_Term =>
390     ivlCurrRange := ivlCurrRange - 2;
391     if (bins(binValI) /= '0') then
392         ivlLow := ivlLow + ivlCurrRange;
393         ivlCurrRange := "000000010"; --2
394         Flushed := "01";
395         CABAC_EncState <= RenormE;
396     else
397         Flushed := "11";
398         CABAC_EncState <= RenormE;
399     end if;
400 when others =>
401     CABAC_EncState <= w_finished;
402     Finished <= '1';
403     Term0 <= '1';
404 end case;
405 end if;
406 end process;
407 end struct;

```

Appendix C

```
1  // *****
2  // Asynchronous fifo
3  // 08.05.17
4  // Norwegian University of Science and Technology
5  // Lars Erik Songe Paulsen
6  // *****
7
8  // *****
9  // TODO LIST:
10 // *****
11 // No "almost full" or "almost empty" signaling logic implemented
12 // *****
13 `timescale 1ns/1ps
14
15 module fifo #(parameter
16     BUFFER_SIZE = 16,
17     DATA_WIDTH = 32,
18     ADDRESS_WIDTH = clogb2(BUFFER_SIZE) - 1
19 )
20 (
21 // -----
22 // Data in interface
23 // -----
24 input wire rst_in_n,
25 input wire clock_in,
26 input wire [DATA_WIDTH-1:0] data_in,
27 input wire data_in_valid,
28 output reg data_in_full,
29
30 // -----
31 // Data out interface
32 // -----
33 input wire rst_out_n,
34 input wire clock_out,
35 output wire [DATA_WIDTH-1:0] data_out,
36 output reg data_out_valid,
37 input wire data_out_ack
38 );
39
40 // -----
41 // Functions
42 // -----
43 // ceil log_2
44 function integer clogb2;
45     input integer depth;
46     for (clogb2=0; depth>0; clogb2=clogb2+1)
47         depth = depth >> 1;
48 endfunction
49
50 // -----
51 // Memory interface and logic
52 // Low latency version(no output register)
53 // See XilinxSimpleDualPort1ClockBlockRamExample.v for detailed documentation
54 // -----
55 reg [DATA_WIDTH-1:0] Buffer[BUFFER_SIZE-1:0];
56
57 // Initialize memory values to all zeros
58 generate
59     integer ram_index;
60     initial
61         for(ram_index = 0; ram_index < BUFFER_SIZE; ram_index = ram_index + 1)
62             Buffer[ram_index] = {DATA_WIDTH{1'b0}};
63 endgenerate
64
65
66
67 // Conditional sampling of data_in
68 always @(posedge clock_in) begin
69     if (data_in_valid && !data_in_full)
70         Buffer[BufferWriteAddress] <= data_in;
71 end
```

```

72
73 // data_out must only be sampled when data_out_valid is asserted
74 assign data_out = Buffer[BufferReadAddress];
75
76 // MSB used for checking fifo full condition
77 // Remainder is actual Buffer address
78 reg [ADDRESS_WIDTH:0] ExtendedBufferWriteAddress, ExtendedBufferReadAddress;
79
80 // Used for addressing memory
81 wire [ADDRESS_WIDTH-1:0] BufferWriteAddress, BufferReadAddress;
82
83 // Binary coded (ADDRESS_WIDTH) bit memory next address
84 wire [ADDRESS_WIDTH:0] WriteNextAddress, ReadNextAddress;
85
86 // Gray coded Pointers for generating full/empty signals
87 reg [ADDRESS_WIDTH:0] WriteGrayPointer, ReadGrayPointer;
88
89 // Gray coded Next Pointers for synchronizing across clock domains
90 wire [ADDRESS_WIDTH:0] WriteGrayNextPointer, ReadGrayNextPointer;
91
92 // Gray coded pointers for synchronizing across clock domains
93 // 2 registers used to avoid metastability
94 reg [ADDRESS_WIDTH:0] WriteGrayPointer2Read1, ReadGrayPointer2Write1;
95 reg [ADDRESS_WIDTH:0] WriteGrayPointer2Read2, ReadGrayPointer2Write2;
96
97 // Wires to signal fifo status
98 wire DataInFull, DataOutEmpty;
99
100 // -----
101 // Write side logic
102 // -----
103 // Check full condition
104 assign DataInFull = (WriteGrayNextPointer ==
105                     {~ReadGrayPointer2Write2[ADDRESS_WIDTH:ADDRESS_WIDTH-1],
106                      ReadGrayPointer2Write2[ADDRESS_WIDTH-2:0]});
107 // Remove MSB before memory indexing
108 assign BufferWriteAddress = ExtendedBufferWriteAddress[ADDRESS_WIDTH-1:0];
109 // Increase Write address if conditions are met
110 assign WriteNextAddress = ExtendedBufferWriteAddress +
111                           (data_in_valid & ~data_in_full);
112 // Binary to Gray code conversion
113 assign WriteGrayNextPointer = (WriteNextAddress>>1) ^ WriteNextAddress;
114
115 always @(posedge clock_in or negedge rst_in_n) begin
116     if (!rst_in_n) begin
117         data_in_full <= 0;
118         ExtendedBufferWriteAddress <= 0;
119         WriteGrayPointer <= 0;
120         WriteGrayPointer2Read1 <= 0;
121         WriteGrayPointer2Read2 <= 0;
122     end
123     else begin
124         // Update data in full register
125         data_in_full <= DataInFull;
126         // Update Write address register
127         ExtendedBufferWriteAddress <= WriteNextAddress;
128         // Update current Gray code writepointer
129         WriteGrayPointer <= WriteGrayNextPointer;
130         // Send previous Gray code writepointer to Read side logic
131         WriteGrayPointer2Read1 <= WriteGrayPointer;
132         WriteGrayPointer2Read2 <= WriteGrayPointer2Read1;
133     end
134 end
135
136 // -----
137 // Read side logic
138 // -----
139 // Check empty condition
140 assign DataOutEmpty = (ReadGrayNextPointer==WriteGrayPointer2Read2);
141 // Remove MSB before memory indexing
142 assign BufferReadAddress = ExtendedBufferReadAddress[ADDRESS_WIDTH-1:0];
143 // Increase Read address if conditions are met
144 assign ReadNextAddress = ExtendedBufferReadAddress + (data_out_ack & data_out_valid);

```

```

145 // Binary to Gray code conversion
146 assign ReadGrayNextPointer = (ReadNextAddress>>1) ^ ReadNextAddress;
147
148 always @(posedge clock_out or negedge rst_out_n) begin
149     if (!rst_out_n) begin
150         data_out_valid          <= 0;
151         ExtendedBufferReadAddress <= 0;
152         ReadGrayPointer          <= 0;
153         ReadGrayPointer2Write1   <= 0;
154         ReadGrayPointer2Write2   <= 0;
155     end
156     else begin
157         // Update data out valid register
158         data_out_valid          <= !DataOutEmpty;
159         // Update Read adress register
160         ExtendedBufferReadAddress <= ReadNextAddress;
161         // Update current Gray code readpointer
162         ReadGrayPointer          <= ReadGrayNextPointer;
163         // Send previous Gray code readpointer to Write side logic
164         ReadGrayPointer2Write1   <= ReadGrayPointer;
165         ReadGrayPointer2Write2   <= ReadGrayPointer2Write1;
166     end
167 end
168 endmodule

```



```

1  `timescale 1ns/1ps
2
3  module fifo_tb;
4  parameter BUFFER_SIZE = 128;//128;
5  parameter DATA_WIDTH = 32;
6
7      // Data in interface
8      reg rst_in_n;
9      reg clock_in;
10     reg [DATA_WIDTH-1:0] data_in;
11     reg data_in_valid;
12     wire data_in_full;
13
14     // Data out interface
15     reg rst_out_n;
16     reg clock_out;
17     wire [DATA_WIDTH-1:0] data_out;
18     wire data_out_valid;
19     reg data_out_ack;
20
21     fifo dut(.rst_in_n(rst_in_n),
22             .clock_in(clock_in),
23             .data_in(data_in),
24             .data_in_valid(data_in_valid),
25             .data_in_full(data_in_full),
26             .rst_out_n(rst_out_n),
27             .clock_out(clock_out),
28             .data_out(data_out),
29             .data_out_valid(data_out_valid),
30             .data_out_ack(data_out_ack)
31             );
32
33     reg [31:0] COMPARE[0:65536];
34     reg [7:0] in_index, out_index;
35
36     initial begin // data in
37         in_index = 8'b00000000;
38         clock_in = 1'b0;
39         rst_in_n = 1'b0;
40         data_in_valid = 1'b1;
41         data_in = 32'h00000001;
42         #5;
43         clock_in = 1'b1;
44         rst_in_n = 1'b1;
45         //data_in_valid = 1'b1;
46         #5;
47         repeat (20) begin
48             if(!data_in_full) begin
49                 if(clock_in) begin // change data on negedge
50                     if(data_in != 0) begin
51                         data_in = (data_in) + 1;
52                     end
53                     else begin
54                         data_in <= 1'b1;
55                     end
56                 end
57                 else begin
58                     end
59             end
60             clock_in = ~clock_in;
61             #50;
62         end
63
64
65
66
67         repeat (200) begin
68             if(!data_in_full) begin
69                 if(clock_in) begin // change data on negedge
70                     if(data_in != 0) begin
71                         data_in = (data_in) + 1;
72                     end
73                     else begin

```

```

74         data_in <= 1'b1;
75     end
76 end
77 else begin
78     end
79 end
80 clock_in = ~clock_in;
81 #2;
82 end
83
84 repeat (200) begin
85     if(!data_in_full) begin
86         if(clock_in) begin // change data on negedge
87             if(data_in != 0) begin
88                 data_in = (data_in) + 1;
89             end
90             else begin
91                 data_in <= 1'b1;
92             end
93         end
94         else begin
95             end
96         end
97         clock_in = ~clock_in;
98         #50;
99     end
100 end
101
102 // Sample data to compare with output
103 always @(posedge clock_in or negedge rst_in_n) begin
104     if (!rst_in_n) begin
105
106     end
107     else if (data_in_valid && !data_in_full) begin
108         COMPARE[in_index] <= (data_in);
109         in_index <= in_index + 1;
110     end
111 end
112
113 initial begin // data out
114     out_index = 8'b00000000;
115     clock_out = 1'b0;
116     rst_out_n = 1'b0;
117     data_out_ack = 1'b0;
118     #5;
119     rst_out_n = 1'b1;
120     data_out_ack = 1'b1;
121     #220;
122     clock_out = 1'b1;
123
124
125
126
127
128
129
130
131
132
133 repeat (200) begin
134     #1;
135     if (data_out_valid) begin
136         if(clock_out) begin
137             if (data_out == COMPARE[out_index]) begin
138                 $monitor("data match");
139             end
140             else begin
141                 $monitor("data mismatch: out_index: %d: %d != %d",out_index,
142                     COMPARE[out_index], data_out);
143             end
144         end
145         else begin
146             out_index = out_index + 1;

```

```

147         end
148     end
149     clock_out = ~clock_out;
150 end
151
152 repeat (120) begin
153     #25;
154     if (data_out_valid) begin
155         if (clock_out) begin
156             if (data_out == COMPARE[out_index]) begin
157                 $monitor("data match");
158             end
159             else begin
160                 $monitor("data mismatch: out_index: %d: %d != %d",out_index,
161                     COMPARE[out_index], data_out);
162             end
163         end
164         else begin
165             out_index = out_index + 1;
166         end
167     end
168     clock_out = ~clock_out;
169 end
170
171 repeat (360) begin
172     #1;
173     if (data_out_valid) begin
174         if (clock_out) begin
175             if (data_out == COMPARE[out_index]) begin
176                 $monitor("data match");
177             end
178             else begin
179                 $monitor("data mismatch: out_index: %d: %d != %d",out_index,
180                     COMPARE[out_index], data_out);
181             end
182         end
183         else begin
184             out_index = out_index + 1;
185         end
186     end
187     clock_out = ~clock_out;
188 end
189 end
190 endmodule

```

Appendix D

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows.Forms;
7
8  namespace HEVC_CABAC_Verification_Tool
9  {
10     class CABAC_encoder
11     {
12         private List<Syntax_element> S_E_ = new List<Syntax_element>();
13         private int S_E_index;
14         private List<bin> Encoded;
15
16         public uint offset;
17
18         static uint qCodIRangeidx, CodIRangeLPS, codIRange, codIOffset, codILow;
19         static uint bitsOutstanding;
20         public bool firstBitFlag;
21
22
23         static uint ctxIdxTable_depth = 120;
24
25         public uint[,] rangeTabLPS = new uint[64, 4];
26         public uint[] transIdxMPS = new uint[64];
27         public uint[] transIdxLPS = new uint[64];
28         public uint[] pStateIdxTable = new uint[ctxIdxTable_depth * 3 * 52];
29         public uint[] MPSIdxTable = new uint[ctxIdxTable_depth * 3 * 52];
30
31         public List<bin> Encode(List<Syntax_element> S_E)
32         {
33             S_E_ = S_E;
34             Encoded = new List<bin>();
35             bitsOutstanding = 0;
36             S_E_index = 0;
37
38             uint bin;
39             bool bypass;
40
41             ResetCodeVals();
42
43             try
44             {
45                 while (S_E_index < S_E.Count)
46                 {
47                     bin = read_bit(out bypass);
48                     if (bypass)
49                     {
50                         EncodeBypass(bin);
51                     }
52                     else
53                     {
54                         EncodeDecision(bin);
55                     }
56                 }
57                 EncodeTerminate(1);
58
59                 return Encoded;
60             }
61             catch (Exception ex)
62             {
63                 MessageBox.Show(ex.ToString());
64             }
65
66             return new List<bin>();
67         }
68
69
70
71
```

```

72
73
74
75
76
77
78
79 public uint read_bit(out bool bypass)
80 {
81     try
82     {
83         while (S_E_index < S_E_.Count)
84         {
85             if (S_E_[S_E_index].currPos < S_E_[S_E_index].Bins.Count)
86             {
87                 bypass = S_E_[S_E_index].bypass;
88                 return (S_E_[S_E_index].Bins[S_E_[S_E_index].currPos++] == '1') ? (uint)1 : 0;
89             }
90             else
91             {
92                 S_E_index++;
93             }
94         }
95     }
96     catch (Exception ex)
97     {
98         MessageBox.Show(ex.ToString());
99     }
100     //MessageBox.Show("read bit called when finished");
101     bypass = false;
102     return 0;
103 }
104
105 public void ResetCodeVals()
106 {
107     //The status of the arithmetic decoding engine is represented by the variables codIRange and
108     //↳ codIOffset.
109     //In the initialization procedure of the arithmetic decoding process,
110     //codIRange is set equal to 510 and codIOffset is set equal to the value returned from
111     //↳ read_bits(9)
112     //interpreted as a 9 bit binary representation of an unsigned integer with most significant bit
113     //↳ written first.
114     codIRange = 510;
115     codILow = 0;
116     qCodIRangeidx = 0;
117     CodIrangeLPS = 0;
118     codIOffset = 0;
119     codILow = 0;
120 }
121
122 public void EncodeDecision(uint bin)
123 {
124     qCodIRangeidx = (codIRange >> 6) & 3;
125     CodIrangeLPS = rangeTabLPS[pStateIdxTable[offset], qCodIRangeidx];
126     codIRange = codIRange - CodIrangeLPS;
127     if (bin != MPSIdxTable[offset])
128     {
129         codILow = codILow + codIRange;
130         codIRange = CodIrangeLPS;
131         if (pStateIdxTable[offset] == 0)
132         {
133             MPSIdxTable[offset] = 1 - MPSIdxTable[offset];
134         }
135         pStateIdxTable[offset] = transIdxLPS[pStateIdxTable[offset]];
136     }
137     else
138     {
139         pStateIdxTable[offset] = transIdxMPS[pStateIdxTable[offset]];
140     }
141     RenormE();
142 }

```

```

142 public void EncodeBypass(uint bin)
143 {
144     try
145     {
146         codILow = codILow << 1;
147         if (bin != 0)
148         {
149             codILow = codILow + codIRange;
150         }
151         if (codILow >= 1024)
152         {
153             PutBit(1);
154             codILow = codILow - 1024;
155         }
156         else if (codILow < 512)
157         {
158             PutBit(0);
159         }
160         else
161         {
162             codILow = codILow - 512;
163             bitsOutstanding++;
164         }
165     }
166     catch (Exception ex)
167     {
168         MessageBox.Show(ex.ToString());
169     }
170 }
171
172 public void RenormE()
173 {
174     while (codIRange < 256)
175     {
176         if (codILow < 256)
177         {
178             PutBit(0);
179         }
180         else if (codILow >= 512)
181         {
182             codILow = codILow - 512;
183             PutBit(1);
184         }
185         else
186         {
187             codILow = codILow - 256;
188             bitsOutstanding = bitsOutstanding + 1;
189         }
190         codIRange = codIRange << 1;
191         codILow = codILow << 1;
192     }
193 }
194
195 //string teststring;
196
197 public void PutBit(uint B)
198 {
199     if (B != 0 && B != 1) { MessageBox.Show("ERROR: PutBit called with argument: " + B.ToString() +
200 ↵      "\n Only 0 or 1 is valid arguments"); }
201
202     if (firstBitFlag)
203     {
204         firstBitFlag = false;
205     }
206     else
207     {
208         bin tempBin = new bin();
209         tempBin.val = (B == 1) ? '1' : '0';
210         Encoded.Add(tempBin);
211     }
212     while (bitsOutstanding > 0)
213     {
214         bin tempBin = new bin();

```

```

214         tempBin.val = (B == 1) ? '0' : '1';
215         Encoded.Add(tempBin);
216         bitsOutstanding--;
217     }
218 }
219
220 public void EncodeTerminate(uint bin)
221 {
222     try
223     {
224         codIRange = codIRange - 2;
225         if (bin != 0)
226         {
227             codILow = codILow + codIRange;
228             EncodeFlush();
229         }
230         else
231         {
232             RenormE();
233         }
234     }
235     catch (Exception ex)
236     {
237         MessageBox.Show(ex.ToString());
238     }
239 }
240
241 public void EncodeFlush()
242 {
243     try
244     {
245         codIRange = 2;
246         RenormE();
247         PutBit((codILow >> 9) & 1);
248         //PutBit(((codILow >> 7) & 3) | 1); // does not work. Using a workaround for now.
249         PutBit((codILow >> 8) & 1);
250         PutBit(((codILow >> 7) & 1) | 1);
251     }
252     catch (Exception ex)
253     {
254         MessageBox.Show(ex.ToString());
255     }
256 }
257
258 }
259
260 class Syntax_element
261 {
262     public List<char> Bins = new List<char>();
263     public int ctxIdx;
264     public int currPos = 0;
265     public int nr = 0;
266     public bool bypass;
267 }
268
269 class bin
270 {
271     public char val;
272 }
273 }

```

Appendix E

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows.Forms;
7
8  namespace HEVC_CABAC_Verification_Tool
9  {
10     class CABAC_decoder
11     {
12         private List<bin> bins_ = new List<bin>();
13         private int bins_index;
14         private int syntax_element_index;
15
16
17         private char[] Decoded;
18         private int Decodedindex;
19
20         public RichTextBox binValTarget;
21         public RichTextBox debugg;
22         uint binValIndex;
23
24         public uint offset;
25
26         static uint qCodIRangeidx, CodIrangeLPS, codIRange, codIOffset, codILow;
27         static uint bitsOutstanding;
28         static bool firstBitFlag;
29
30
31         static uint ctxIdxTable_depth = 120;
32
33         public uint[,] rangeTabLPS = new uint[64, 4];
34         public uint[] transIdxMPS = new uint[64];
35         public uint[] transIdxLPS = new uint[64];
36         public uint[] pStateIdxTable = new uint[ctxIdxTable_depth * 3 * 52];
37         public uint[] MPSIdxTable = new uint[ctxIdxTable_depth * 3 * 52];
38
39
40         private void ResetCodeVals()
41         {
42             try
43             {
44                 //The status of the arithmetic decoding engine is represented by the variables codIRange and
45                 //↳ codIOffset.
46                 //In the initialization procedure of the arithmetic decoding process,
47                 //codIRange is set equal to 510 and codIOffset is set equal to the value returned from
48                 //↳ read_bits(9)
49                 //interpreted as a 9 bit binary representation of an unsigned integer with most significant
50                 //↳ bit written first.
51                 codIRange = 510;
52                 codILow = 0;
53                 qCodIRangeidx = 0;
54                 CodIrangeLPS = 0;
55                 codIOffset = 0;
56                 codILow = 0;
57
58                 char[] tempA = "XXXXXXXXX".ToCharArray();
59                 for (int i = 0; i < 9; i++)
60                 {
61                     tempA[i] = (read_bit() == 1) ? '1' : '0';
62                 }
63                 codIOffset = Convert.ToInt32(new string(tempA), 2); // read 9 first binary vals
64             }
65             catch (Exception ex)
66             {
67                 MessageBox.Show(ex.ToString());
68             }
69         }
70     }
71 }
```



```

69     public string Decode(List<bin> bins)
70     {
71         Decoded = new char[65535];
72         Decodedindex = 0;
73
74         bins_ = bins;
75         bins_index = 0;
76         syntax_element_index = 0;
77
78         ResetCodeVals();
79
80         try
81         {
82             return new string(Decoded).Substring(0, Decodedindex);
83         }
84         catch (Exception ex)
85         {
86             MessageBox.Show(ex.ToString());
87         }
88
89         return "empty";
90     }
91
92
93     public uint read_bit()
94     {
95         try
96         {
97             while (bins_index < bins_.Count)
98             {
99                 return (bins_[bins_index++ ].val == '1') ? (uint)1 : 0;
100             }
101         }
102         catch (Exception ex)
103         {
104             MessageBox.Show(ex.ToString());
105         }
106         MessageBox.Show("read bit called when finished");
107         return 0;
108     }
109
110
111
112     private void DecodeDecicison()
113     {
114         try
115         {
116             qCodIRangeidx = (codIRange >> 6) & 3;
117             CodIrangeLPS = rangeTabLPS[pStateIdxTable[offset], qCodIRangeidx];
118             codIRange = codIRange - CodIrangeLPS;
119             if (codIOffset >= codIRange)
120             {
121                 Decoded[Decodedindex++] = (MPSIdxTable[offset] == 1) ? '0' : '1';
122                 codIOffset = codIOffset - codIRange;
123                 codIRange = CodIrangeLPS;
124                 if (pStateIdxTable[offset] == 0)
125                 {
126                     MPSIdxTable[offset] = 1 - MPSIdxTable[offset];
127                 }
128                 pStateIdxTable[offset] = transIdxLPS[pStateIdxTable[offset]];
129             }
130             else
131             {
132                 Decoded[Decodedindex++] = ((MPSIdxTable[offset] == 1) ? '1' : '0');
133                 pStateIdxTable[offset] = transIdxMPS[pStateIdxTable[offset]];
134             }
135
136             RenormD();
137         }
138         catch (Exception ex)
139         {
140             MessageBox.Show(ex.ToString());
141         }

```

```

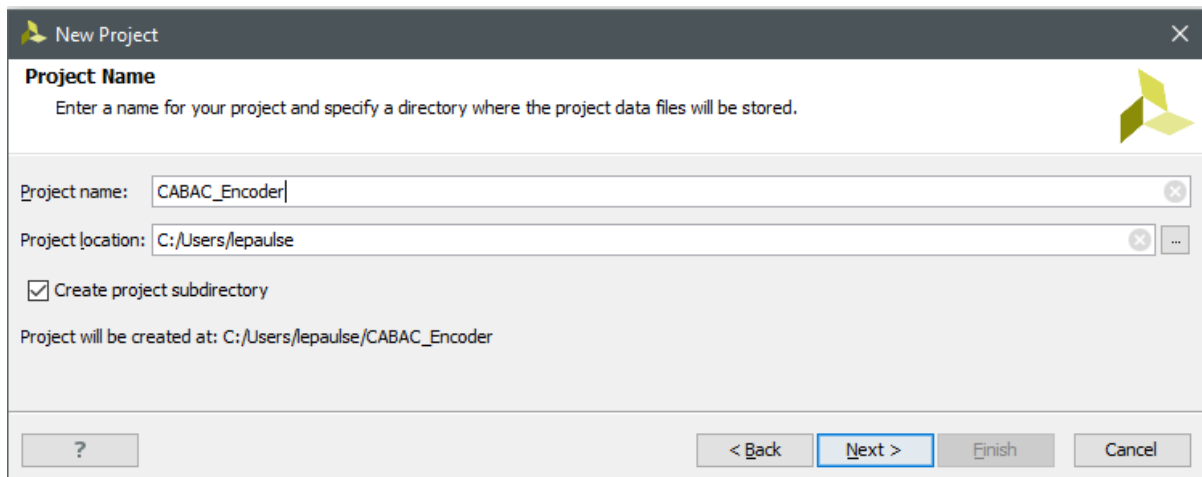
142     }
143 }
144
145 private void DecodeBypass()
146 {
147     codIOffset = codIOffset << 1;
148     codIOffset = codIOffset | read_bit();
149     if (codIOffset >= codIRange)
150     {
151         Decoded[Decodedindex++] = '1';
152         codIOffset = codIOffset - codIRange;
153     }
154     else
155     {
156         Decoded[Decodedindex++] = '0';
157     }
158 }
159
160 private void RenormD()
161 {
162     try
163     {
164         while (codIRange < 256)
165         {
166             codIRange = codIRange << 1;
167             codIOffset = codIOffset << 1;
168             codIOffset = codIOffset | read_bit();
169         }
170         if (codIOffset >= codIRange)
171         {
172             MessageBox.Show("Decoding error:\n The bitstream shall not contain data that result in
173                 ↳ a value of " +
174                             "codIOffset being greater than or equal to codIRange upon completion of
175                 ↳ this process.");
176         }
177     }
178     catch (Exception ex)
179     {
180         MessageBox.Show(ex.ToString());
181     }
182 }
183 }

```

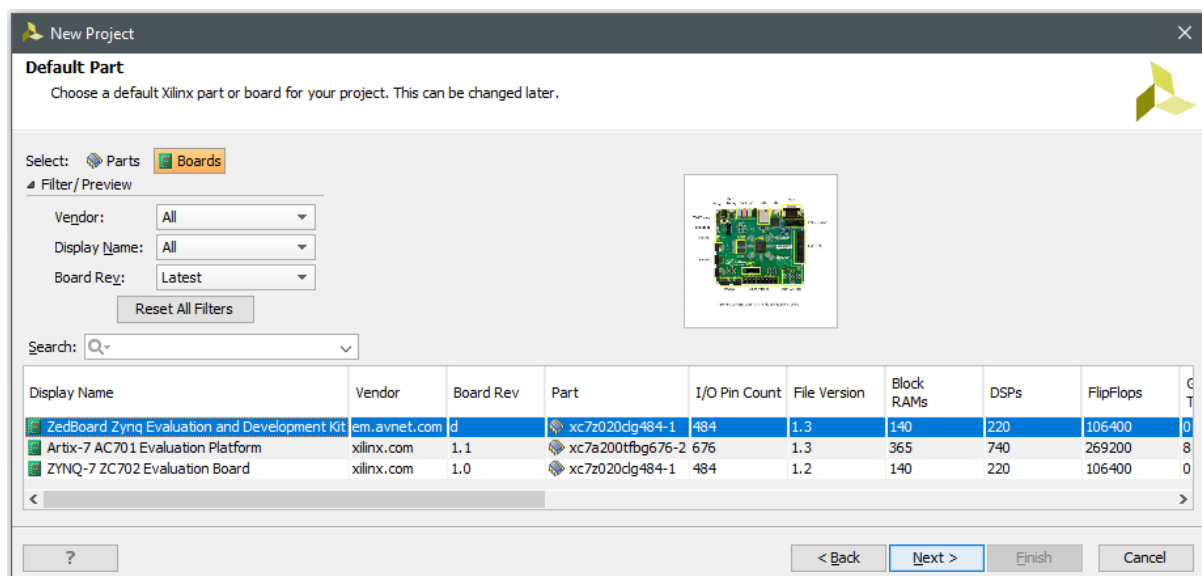
Appendix F

This Appendix shows how to setup the different hardware modules and testbenches. It also shows how the software model can be used to verify hardware encoder output. The systems was developed using Vivado 2016.4 WebPack Edition. Relevant files is included in the delivery folder.

CABAC Hardware Encoder Test System Setup

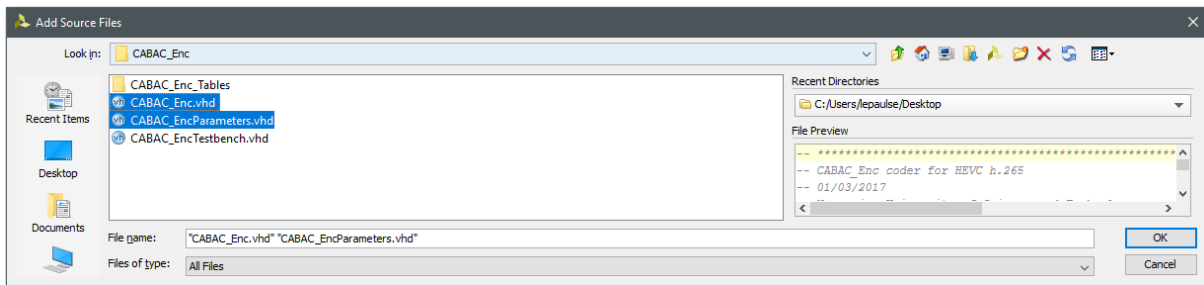


Create a new project in Vivado. Give it a suitable name and click next. Select "RTL Project" and "Do not specify sources at this time", and click next.



Select the Zedboard in the Boards Tab. Click next, then click finish.

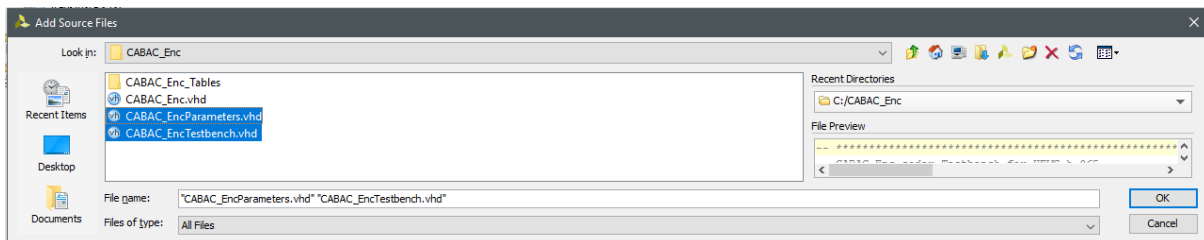
Navigate to the sources tab, and add the following files to design sources:



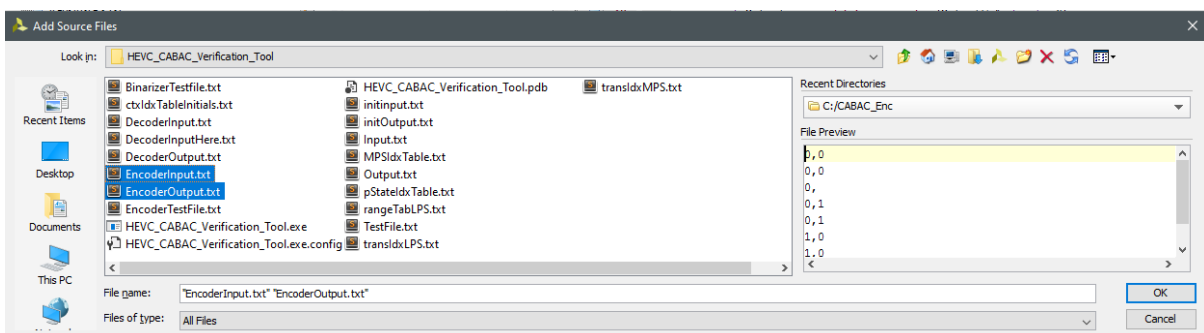
Then add the tables in the tables folder to design sources:



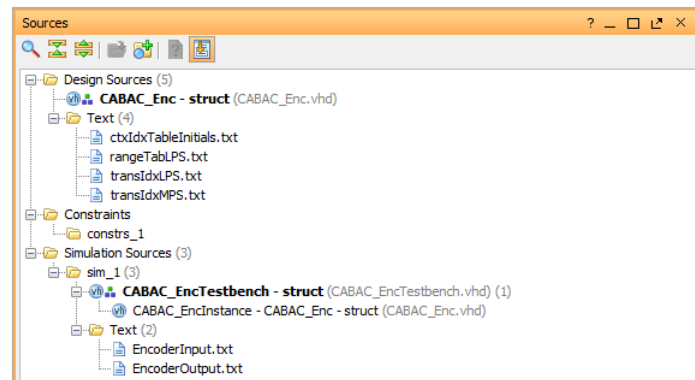
Then add the following files to simulation sources:



Then add the following files to simulation sources. Note that these files are located in the HEVC_CABAC_Verification_Tool folder.



The Source tab should now look like this.



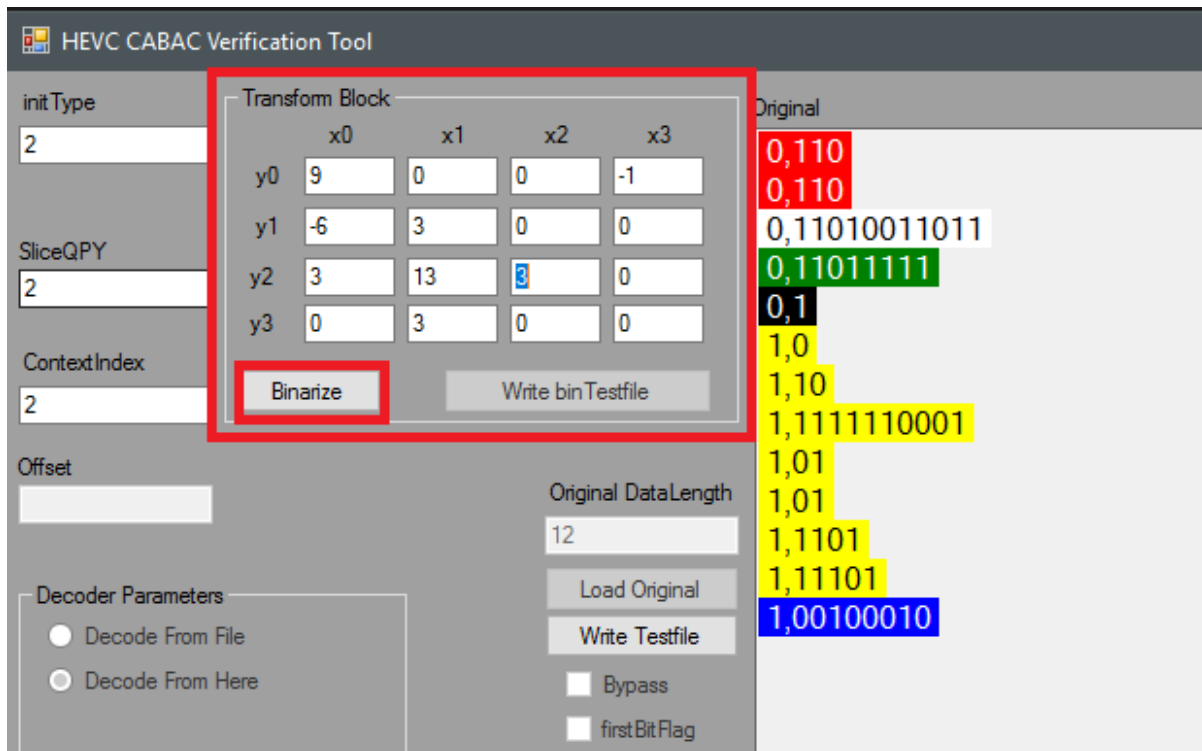
Open the "CABAC_EncTestbench.vhd" file and verify that the "EncoderInput.txt" and "EncoderOutput.txt" linked in the code is in the same folder as the "HEVC_CABAC_Verification_Tool.exe" executable file that is used. It should look something like this:

```
-- Include EncoderInput.txt in Vivado simulation sources.
file TestFile : TEXT open read_mode is "C:\Users\lepaulse\Desktop\CABACFiles\HEVC_CABAC_Verification_Tool\EncoderInput.txt";
-- Include EncoderOutput.txt in Vivado simulation sources.
file TestFileOut : TEXT open write_mode is "C:\Users\lepaulse\Desktop\CABACFiles\HEVC_CABAC_Verification_Tool\EncoderOutput.txt";
```

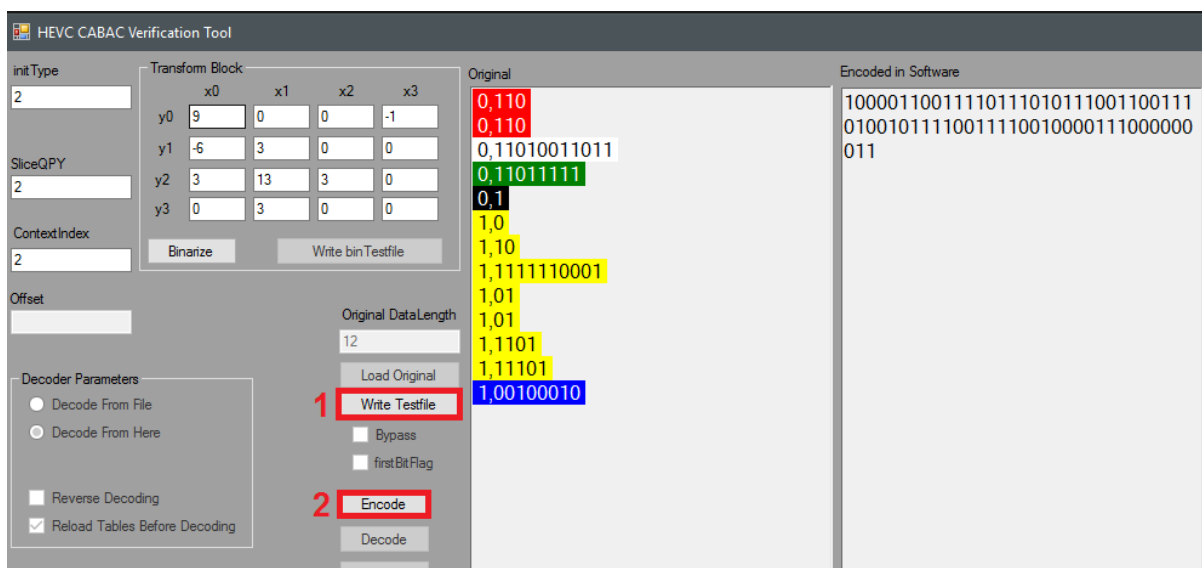
It is now possible to generate Encoder test data using the software model. First start the executable file.

<< Desktop > CABACFiles > HEVC_CABAC_Verification_Tool				Search HEVC_CABAC_Verifica...
Name	Date modified	Type	Size	
BinarizerTestfile.txt	03.08.2017 20.49	TXT File	1 KB	
ctxIdxTableInitials.txt	30.05.2017 10.43	TXT File	141 KB	
DecoderInput.txt	03.08.2017 21.56	TXT File	0 KB	
DecoderInputHere.txt	03.08.2017 22.04	TXT File	0 KB	
DecoderOutput.txt	30.05.2017 10.43	TXT File	0 KB	
EncoderInput.txt	03.08.2017 22.04	TXT File	1 KB	
EncoderOutput.txt	03.08.2017 22.05	TXT File	1 KB	
EncoderTestFile.txt	03.08.2017 21.04	TXT File	1 KB	
HEVC_CABAC_Verification_Tool.exe	03.08.2017 22.04	Application	47 KB	
HEVC_CABAC_Verification_Tool.exe.config	30.05.2017 10.43	XML Configuratio...	1 KB	
HEVC_CABAC_Verification_Tool.pdb	03.08.2017 22.04	Program Debug D...	80 KB	
initinput.txt	08.06.2017 12.45	TXT File	2 KB	
initOutput.txt	08.06.2017 12.53	TXT File	66 KB	

Create some binarized bins by changing the values in the transform block, or simply clicking Binarize.



1: First write the current binarization to the EncoderInput.txt file by pressing the Write TestFile button. **2:** Then encode this data with the software encoder by pressing the Encode Button.



C:\Users\lepaulse\CABAC\CABAC_Encoder.xpr - Vivado 2016.4

File Edit Flow Tools Window Layout View Run Help

Flow Navigator

Project Manager

- Project Settings
- Add Sources
- Language Templates
- IP Catalog

IP Integrator

- Create Block Design
- Open Block Design
- Generate Block Design

Simulation

- Simulation Settings
- Run Simulation

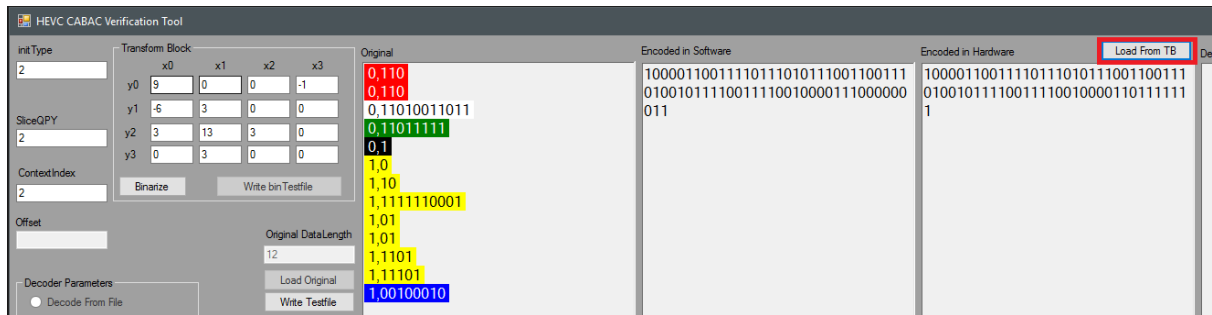
Behavioral Simulation - Functional - sim_1 - CABAC_EncTestbench

Sources

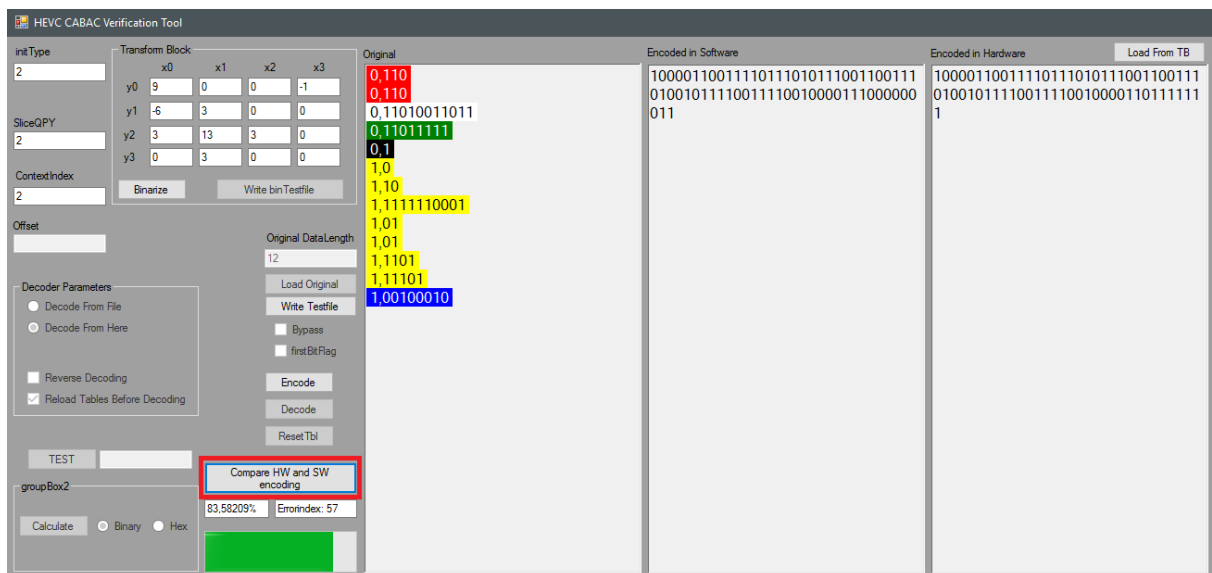
- Design Sources (5)
 - CABAC_Enc - struct (CABAC_Enc.vhd)
 - Text (4)
 - ctxIdxTableInitials.txt
 - rangeTabLPS.txt
 - transIdxLPS.txt
 - transIdxMPS.txt
 - Constraints
 - constrs_1
 - Simulation Sources (3)
 - sim_1 (3)
 - CABAC_EncTestbench - struct (CABAC_EncTestbench.vhd) (1)
 - CABAC_EncInstance - CABAC_Enc - struct (CABAC_Enc.vhd)
 - Text (2)
 - EncoderInput.txt
 - EncoderOutput.txt

5

Head back to the HEVC_CABAC_Verification_Tool.exe software and press the Load From TB button. This will load the hardware encoder result.



With both hardware and software encoder result showing, it is now possible to press the Compare HW and SW encoding button. This will show how many percent of the encoding is correct, as well as the index of the first conflicting character.



Note that some errors may occur either due to improper termination(as seen above) or if the BitsOutstanding Loop in the hardware encoder is completed without the BitsOutstanding register reaching a value of 0.

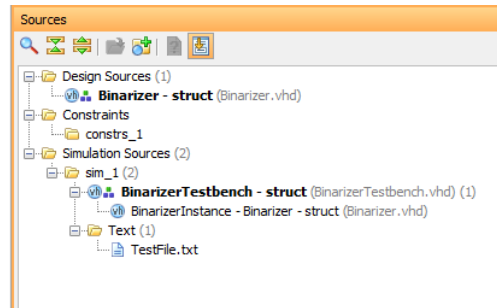
The Bypassflag is signaled to the encoders by appending it to each bins as shown below:

The encoders will use the context index signaled by initType, SliceQPY and ctxIdx. For the encoders to return the same result, these variables should be set to the same values, as shown above and below.

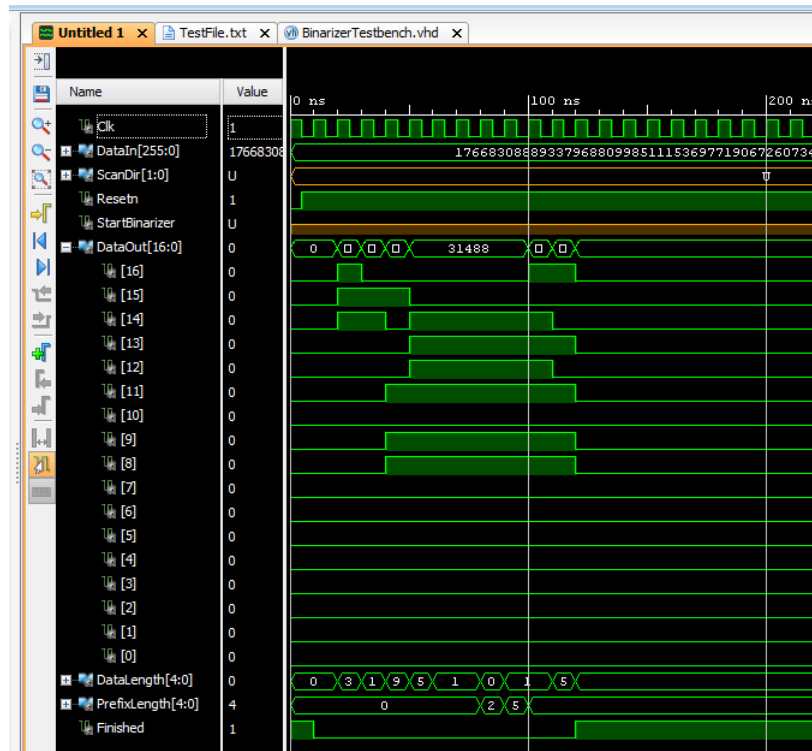
Note that it is possible to manually edit “EncoderInput.txt” to create a custom test pattern. But it is required to include the Bypassflag on each line.

Binarizer

Create a new project in Vivado and give it a suitable name for the Binarizer module(same steps as for the CABAC Encoder, adding zedboard as target). Add the Binarizer files from the "Binarizer_Case-Based_ALRem" folder as shown below.



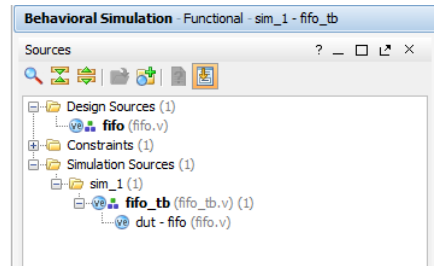
Press run simulation, and make sure it is able to run for long enough.



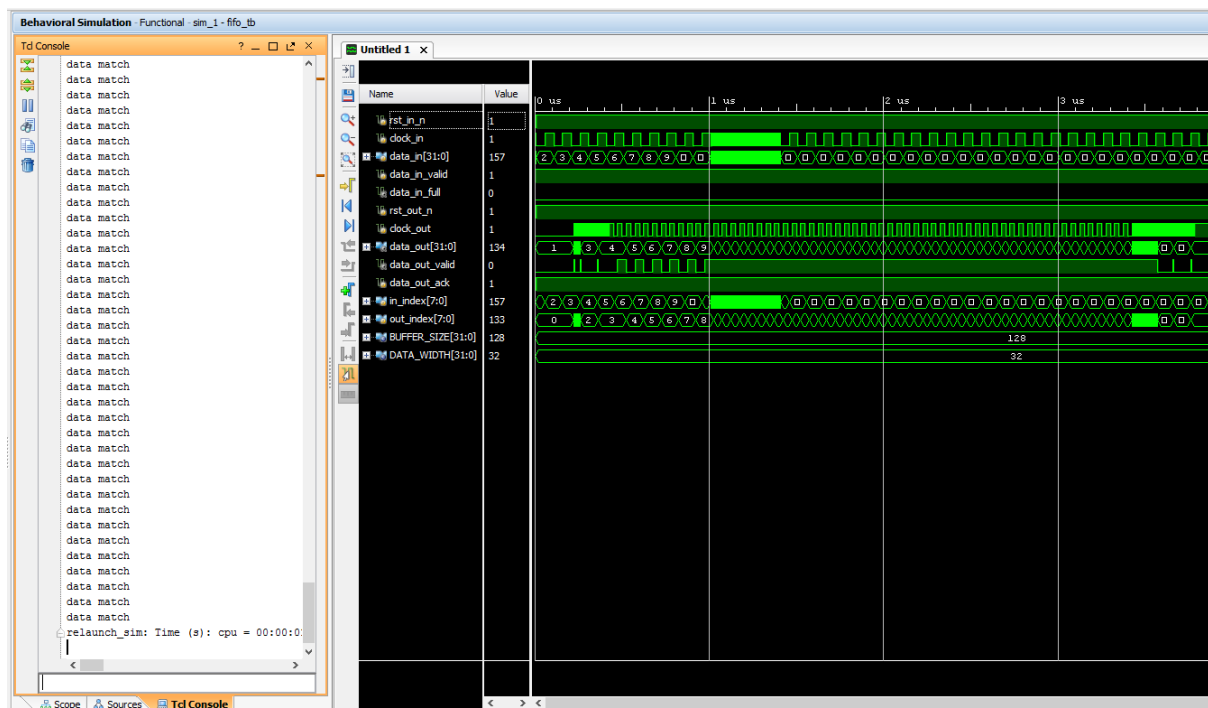
The binarizer will output the finished binarized elements in the order of LASTx, LASTy, SIG, ABS1, ABS2, ALRem and SIGN.

Fifo

Create a new project in Vivado and give it a suitable name for the Fifo module(same steps as for the CABAC Encoder, adding zedboard as target). Add the fifo files as shown below.



Press run simulation, and make sure it is able to run for long enough.



The testbench will run the asynchronous fifo using three different frequencies on both the input and output clocks. It will check that the output matches the expected result.

Appendix G

Context table initial value generation functions in C#(.NET). Including overview of residual coding syntax element initvals for all initTypes.

```
1  // Binary initial value generation for HEVC CABAC context modeling.
2  void initTable_file_generation()
3  {
4      // Input file containing initvalues copied from the HEVC standard document.
5      // Should contain initvalues for initType 0, 1 and 2.
6      // Each line should contain an initvalue on each line, starting with
7      // every initvalue from initType 0, then immediately followed by every initvalue
8      // from initType 1, and finally then every initvalue from initType2.
9      StreamWriter input = initializeReadFile("initinput.txt");
10
11     // Output file containing Binary(1/0 ASCII chars) formatted initvalues for any value of sliceQPY and
12     ↪ initType.
13     StreamWriter Output = initializeWriteFile("initOutput.txt");
14
15     Int32 initValue, slopeIdx, intersecIdx, m, n, preCtxState, valMPS, pStateIdx;
16
17     try
18     {
19         for (int SliceQPY = 0; SliceQPY < 52; SliceQPY++)
20         {
21             while (!input.EndOfStream)
22             {
23                 initValue = int.Parse(input.ReadLine());
24                 slopeIdx = initValue >> 4;
25                 intersecIdx = initValue & 15;
26                 m = (slopeIdx * 5) - 45;
27                 n = (intersecIdx << 3) - 16;
28                 preCtxState = Clip3(1, 126, ((m * Clip3(0, 51, SliceQPY)) >> 4) + n);
29                 valMPS = (preCtxState <= 63) ? 0 : 1;
30                 pStateIdx = (valMPS > 0) ? (preCtxState - 64) : (63 - preCtxState);
31
32                 // Binary(1/0 ASCII chars) formatted output
33                 Output.WriteLine(String.Format("{0:X}", valMPS) +
34                                 Convert.ToString(pStateIdx, 2).PadLeft(6, '0'));
35             }
36             input.DiscardBufferedData();
37             input.BaseStream.Seek(0, System.IO.SeekOrigin.Begin);
38         }
39     }
40     catch (Exception ex)
41     {
42         MessageBox.Show(ex.ToString());
43     }
44     Output.Close();
45     input.Close();
46 }
47
48 int Clip3(int x, int y, int z)
49 {
50     if (z < x) return x;
51     else if (z > y) return y;
52     else return z;
53 }
54
55
56
57
58
59
60
61
62
63
64
```

```

65 // Initialize write file to current folder.
66 StreamWriter initializeWriteFile(string filename)
67 {
68     try
69     {
70         FileStream fs;
71         StreamWriter file;
72         string startUpPath;
73         string currentLogFileName;
74         string logfolder;
75         currentLogFileName = string.Format(filename);
76         startUpPath = Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().Location);
77         logfolder = Path.Combine(startUpPath, currentLogFileName);
78         File.Delete(logfolder);
79         fs = File.Create(logfolder);
80         file = new StreamWriter(fs);
81         return file;
82     }
83     catch (IOException)
84     {
85         MessageBox.Show("Unable to access: " + filename);
86     }
87     catch (Exception ex)
88     {
89         MessageBox.Show(ex.ToString());
90     }
91     return null;
92 }
93
94 // Initialize read file to current folder.
95 StreamReader initializeReadFile(string filename)
96 {
97     try
98     {
99         FileStream fs;
100         StreamReader file;
101         string startUpPath;
102         string currentLogFileName;
103         string logfolder;
104         currentLogFileName = string.Format(filename);
105         startUpPath = Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().Location);
106         logfolder = Path.Combine(startUpPath, currentLogFileName);
107         fs = File.OpenRead(logfolder);
108         file = new StreamReader(fs);
109         return file;
110     }
111     catch (IOException)
112     {
113         MessageBox.Show("Unable to access: " + filename);
114     }
115     catch (Exception ex)
116     {
117         MessageBox.Show(ex.ToString());
118     }
119     return null;
120 }

```

syntax element	ctxTable	initType					
		0		1		2	
		ctxTable index	initValue	Index	ctxTable index	initValue	Index
last_sig_coeff_x_prefix	9-26	0	110	0	18	125	120
last_sig_coeff_x_prefix	9-26	1	110	1	19	110	121
last_sig_coeff_x_prefix	9-26	2	124	2	20	94	122
last_sig_coeff_x_prefix	9-26	3	125	3	21	110	123
last_sig_coeff_x_prefix	9-26	4	140	4	22	95	124
last_sig_coeff_x_prefix	9-26	5	153	5	23	79	125
last_sig_coeff_x_prefix	9-26	6	125	6	24	125	126
last_sig_coeff_x_prefix	9-26	7	127	7	25	111	127
last_sig_coeff_x_prefix	9-26	8	140	8	26	110	128
last_sig_coeff_x_prefix	9-26	9	109	9	27	78	129
last_sig_coeff_x_prefix	9-26	10	111	10	28	110	130
last_sig_coeff_x_prefix	9-26	11	143	11	29	111	131
last_sig_coeff_x_prefix	9-26	12	127	12	30	111	132
last_sig_coeff_x_prefix	9-26	13	111	13	31	95	133
last_sig_coeff_x_prefix	9-26	14	79	14	32	94	134
last_sig_coeff_x_prefix	9-26	15	108	15	33	108	135
last_sig_coeff_x_prefix	9-26	16	123	16	34	123	136
last_sig_coeff_x_prefix	9-26	17	63	17	35	108	137
last_sig_coeff_y_prefix	9-27	0	110	18	18	125	138
last_sig_coeff_y_prefix	9-27	1	110	19	19	110	139
last_sig_coeff_y_prefix	9-27	2	124	20	20	94	140
last_sig_coeff_y_prefix	9-27	3	125	21	21	110	141
last_sig_coeff_y_prefix	9-27	4	140	22	22	95	142
last_sig_coeff_y_prefix	9-27	5	153	23	23	79	143
last_sig_coeff_y_prefix	9-27	6	125	24	24	125	144
last_sig_coeff_y_prefix	9-27	7	127	25	25	111	145
last_sig_coeff_y_prefix	9-27	8	140	26	26	110	146
last_sig_coeff_y_prefix	9-27	9	109	27	27	78	147
last_sig_coeff_y_prefix	9-27	10	111	28	28	110	148
last_sig_coeff_y_prefix	9-27	11	143	29	29	111	149
last_sig_coeff_y_prefix	9-27	12	127	30	30	111	150
last_sig_coeff_y_prefix	9-27	13	111	31	31	95	151
last_sig_coeff_y_prefix	9-27	14	79	32	32	94	152
last_sig_coeff_y_prefix	9-27	15	108	33	33	108	153
last_sig_coeff_y_prefix	9-27	16	123	34	34	123	154
last_sig_coeff_y_prefix	9-27	17	63	35	35	108	155
coded_sub_block_flag	9-28	0	91	36	4	121	156
coded_sub_block_flag	9-28	1	171	37	5	140	157
coded_sub_block_flag	9-28	2	134	38	6	61	158
coded_sub_block_flag	9-28	3	141	39	7	154	159
sig_coeff_flag	9-29	0	111	40	42	155	160
sig_coeff_flag	9-29	1	111	41	43	154	161
sig_coeff_flag	9-29	2	125	42	44	139	162
sig_coeff_flag	9-29	3	110	43	45	153	163
sig_coeff_flag	9-29	4	110	44	46	139	164
sig_coeff_flag	9-29	5	94	45	47	123	165
sig_coeff_flag	9-29	6	124	46	48	123	166
sig_coeff_flag	9-29	7	108	47	49	167	167
sig_coeff_flag	9-29	8	124	48	50	153	168
sig_coeff_flag	9-29	9	107	49	51	166	169
sig_coeff_flag	9-29	10	125	50	52	183	170
sig_coeff_flag	9-29	11	141	51	53	140	171
sig_coeff_flag	9-29	12	179	52	54	136	172
sig_coeff_flag	9-29	13	153	53	55	153	173
sig_coeff_flag	9-29	14	125	54	56	154	174
sig_coeff_flag	9-29	15	107	55	57	166	175
sig_coeff_flag	9-29	16	125	56	58	183	176
sig_coeff_flag	9-29	17	141	57	59	140	177
sig_coeff_flag	9-29	18	179	58	60	136	178
sig_coeff_flag	9-29	19	153	59	61	153	179
sig_coeff_flag	9-29	20	125	60	62	154	180
sig_coeff_flag	9-29	21	107	61	63	166	181
sig_coeff_flag	9-29	22	125	62	64	183	182
sig_coeff_flag	9-29	23	141	63	65	140	183
sig_coeff_flag	9-29	24	179	64	66	136	184
sig_coeff_flag	9-29	25	153	65	67	153	185
sig_coeff_flag	9-29	26	125	66	68	154	186
sig_coeff_flag	9-29	27	140	67	69	170	187
sig_coeff_flag	9-29	28	139	68	70	153	188
sig_coeff_flag	9-29	29	182	69	71	123	189
sig_coeff_flag	9-29	30	182	70	72	123	190
sig_coeff_flag	9-29	31	152	71	73	107	191
sig_coeff_flag	9-29	32	136	72	74	121	192
sig_coeff_flag	9-29	33	152	73	75	107	193
sig_coeff_flag	9-29	34	136	74	76	121	194
sig_coeff_flag	9-29	35	153	75	77	167	195
sig_coeff_flag	9-29	36	136	76	78	151	196
sig_coeff_flag	9-29	37	139	77	79	183	197
sig_coeff_flag	9-29	38	111	78	80	140	198
sig_coeff_flag	9-29	39	136	79	81	151	199
sig_coeff_flag	9-29	40	139	80	82	183	200
sig_coeff_flag	9-29	41	111	81	83	140	201
sig_coeff_flag	9-29	126	141	82	129	140	202
sig_coeff_flag	9-29	127	111	83	130	140	203
coeff_abs_level_greater1_flag	9-30	0	140	84	24	154	204
coeff_abs_level_greater1_flag	9-30	1	92	85	25	196	205
coeff_abs_level_greater1_flag	9-30	2	137	86	26	196	206
coeff_abs_level_greater1_flag	9-30	3	138	87	27	167	207
coeff_abs_level_greater1_flag	9-30	4	140	88	28	154	208
coeff_abs_level_greater1_flag	9-30	5	152	89	29	152	209
coeff_abs_level_greater1_flag	9-30	6	138	90	30	167	210
coeff_abs_level_greater1_flag	9-30	7	139	91	31	182	211
coeff_abs_level_greater1_flag	9-30	8	153	92	32	182	212
coeff_abs_level_greater1_flag	9-30	9	74	93	33	134	213
coeff_abs_level_greater1_flag	9-30	10	149	94	34	149	214
coeff_abs_level_greater1_flag	9-30	11	92	95	35	136	215
coeff_abs_level_greater1_flag	9-30	12	139	96	36	153	216
coeff_abs_level_greater1_flag	9-30	13	107	97	37	121	217
coeff_abs_level_greater1_flag	9-30	14	122	98	38	136	218
coeff_abs_level_greater1_flag	9-30	15	152	99	39	137	219
coeff_abs_level_greater1_flag	9-30	16	140	100	40	169	220
coeff_abs_level_greater1_flag	9-30	17	179	101	41	194	221
coeff_abs_level_greater1_flag	9-30	18	166	102	42	166	222
coeff_abs_level_greater1_flag	9-30	19	182	103	43	167	223
coeff_abs_level_greater1_flag	9-30	20	140	104	44	154	224
coeff_abs_level_greater1_flag	9-30	21	227	105	45	167	225
coeff_abs_level_greater1_flag	9-30	22	122	106	46	137	226
coeff_abs_level_greater1_flag	9-31	23	197	107	47	182	227
coeff_abs_level_greater2_flag	9-31	0	138	108	6	107	228
coeff_abs_level_greater2_flag	9-31	1	153	109	7	167	229
coeff_abs_level_greater2_flag	9-31	2	136	110	8	91	230
coeff_abs_level_greater2_flag	9-31	3	167	111	9	122	231
coeff_abs_level_greater2_flag	9-31	4	152	112	10	107	232
coeff_abs_level_greater2_flag	9-31	5	152	113	11	167	233
transform_skip_flag	9-25	0	139	114	1	154	234
transform_skip_flag	9-25	3	139	115	4	154	235
explicit_rdpem_flag	9-32	0	116	0	0	139	236
explicit_rdpem_flag	9-32	1	117	2	0	139	237
explicit_rdpem_dir_flag	9-33	0	118	0	0	139	238
explicit_rdpem_dir_flag	9-33	1	119	2	0	139	239

Appendix H

The table based binarisation of the coeff_abs_level_remaining was first verified using an excel implementation. The excel sheets shows the binarization of ALRem for a given value of k and Z.

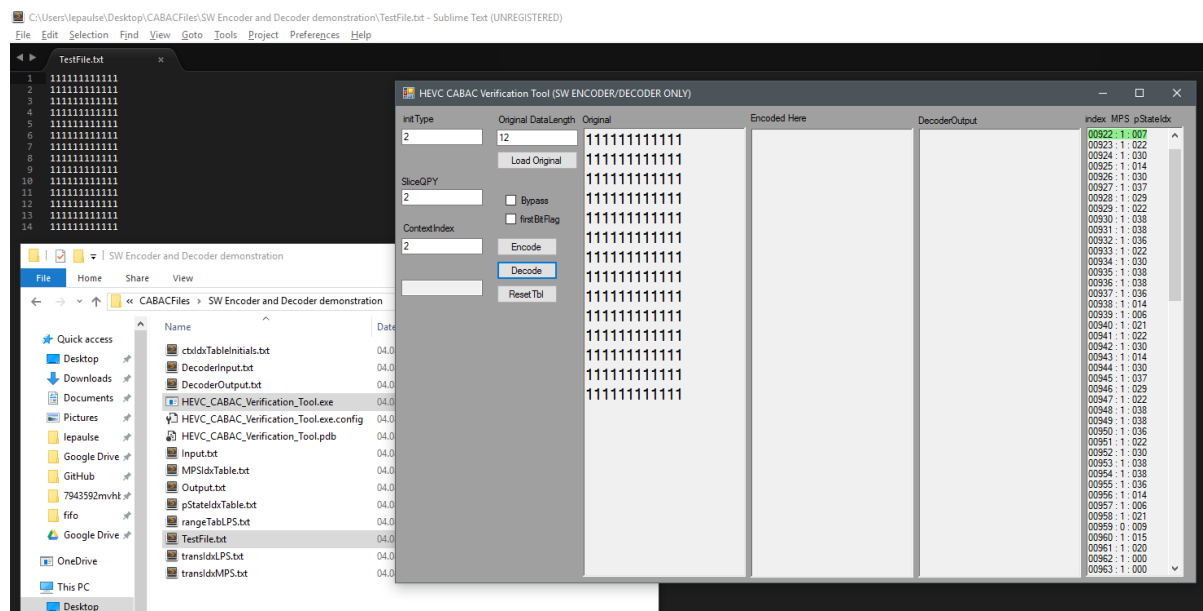
	A	B	C	D	E	F	G
1	K	N_min	N_max	Prefix bins	Suffix bins	Prefix length	Suffix length
2	2	0	3	0	-	1	2
3	Z	4	7	10	-	2	2
4	44	8	11	110	-	3	2
5		12	15	1110	-	4	2
6		16	23	11110	-	5	3
7		24	39	111110	-	6	4
8		40	71	1111110	00100	7	5
9		72	135	11111110	-	8	6
10		136	263	111111110	-	9	7
11		264	519	1111111110	-	10	8
12		520	1031	11111111110	-	11	9
13		1032	2055	111111111110	-	12	10
14		2056	4103	1111111111110	-	13	11
15		4104	8199	11111111111110	-	14	12
16		8200	16391	111111111111110	-	15	13
17		16392	32775	1111111111111110	-	16	14
18		32776	65543	11111111111111110	-	17	15
19		65544	131079	111111111111111110	-	18	16
20		131080	262151	1111111111111111110	-	19	17
21		262152	524295	11111111111111111110	-	20	18
22		524296	1048583	111111111111111111110	-	21	19

The interactive excel sheet(CoeffAbsLevelRemaining.xlsm) is included in the delivery folder.

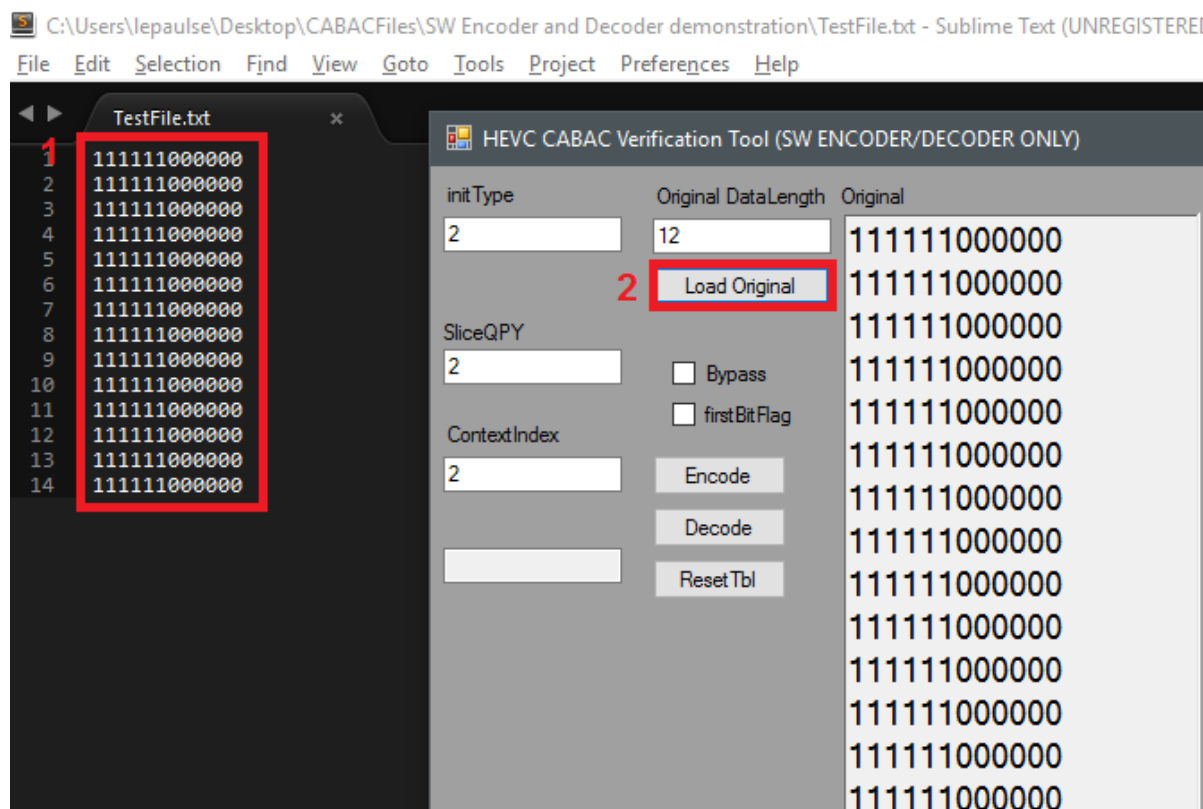
Note that due to the use of an extended DECTOBIN function(vDecimalToBinary), you might need to enable content to use the excel sheet. Because the vDecimalToBinary function only supports output of 16 or less binary symbols, Binarization of suffixes longer than 16-bits will not work properly.

Appendix I

This Appendix shows how a simple demonstration of Software CABAC encoding/decoding using the C# software model. Relevant files is included in the delivery folder. Navigate to the "SW Encoder and Decoder demonstration" folder and open "TestFile.txt" and the HEVC_CABAC_Verification_Tool as seen below.



The "TestFile.txt" contains the encoder input. Each line should contain a number of "0"s and "1" equal or longer than in the Original DataLength TextBox. To change the encoder input: 1 edit "TestFile.txt" 2 Press the Load Original button.



It is now possible to encode and decode. **1** Check the firstBitFlag checkbox(important). This will make sure the first bit is skipped in PutBit, and is required for the decoder to work **2** Press the Encode button. **3** Press the Decode button.

HEVC CABAC Verification Tool (SW ENCODER/DECODER ONLY)

initType	Original DataLength	Original	Encoded Here	DecoderOutput	index MPS pStateIdx
2	12	111111000000	000111001101	111111000000	00922 : 0 : 002
		111111000000	011110011000	111111000000	00923 : 1 : 022
		111111000000	110000100101	111111000000	00924 : 1 : 030
		111111000000	001100010000	111111000000	00925 : 1 : 014
		111111000000	000000100000	111111000000	00926 : 1 : 030
		111111000000	110111000111	111111000000	00927 : 1 : 037
		111111000000	101100010000	111111000000	00928 : 1 : 029
		111111000000	011011100011	111111000000	00929 : 1 : 022
		111111000000	110110001000	111111000000	00930 : 1 : 038
		111111000000	000110111000	111111000000	00931 : 1 : 038
		111111000000	000011011100	111111000000	00932 : 1 : 036
		111111000000	0111101100010	111111000000	00933 : 1 : 022
		111111000000	011110110011	111111000000	00934 : 1 : 030
		111111000000	0101	111111000000	00935 : 1 : 038
		111111000000		111111000000	00936 : 1 : 038
		111111000000		111111000000	00937 : 1 : 036
		111111000000		111111000000	00938 : 1 : 014
		111111000000		111111000000	00939 : 1 : 006
		111111000000		111111000000	00940 : 1 : 021
		111111000000		111111000000	00941 : 1 : 022
		111111000000		111111000000	00942 : 1 : 030
		111111000000		111111000000	00943 : 1 : 014
		111111000000		111111000000	00944 : 1 : 030
		111111000000		111111000000	00945 : 1 : 037
		111111000000		111111000000	00946 : 1 : 029
		111111000000		111111000000	00947 : 1 : 022
		111111000000		111111000000	00948 : 1 : 038
		111111000000		111111000000	00949 : 1 : 038
		111111000000		111111000000	00950 : 1 : 036
		111111000000		111111000000	00951 : 1 : 022
		111111000000		111111000000	00952 : 1 : 030
		111111000000		111111000000	00953 : 1 : 038
		111111000000		111111000000	00954 : 1 : 038
		111111000000		111111000000	00955 : 1 : 036
		111111000000		111111000000	00956 : 1 : 014
		111111000000		111111000000	00957 : 1 : 006
		111111000000		111111000000	00958 : 1 : 021
		111111000000		111111000000	00959 : 0 : 009
		111111000000		111111000000	00960 : 1 : 015
		111111000000		111111000000	00961 : 1 : 020
		111111000000		111111000000	00962 : 1 : 000
		111111000000		111111000000	00963 : 1 : 000

The context index used can be changed by editing initType, SliceQPY and Context Index. But it should be the same for both encoding and decoding.

[illegible][illegible]