NTNU

Norwegian University of
Science and Technology

Design and analysis of an H.265 entropy encoder

Lars Erik Songe Paulsen

Master of Science in Electronics
Submission date: August 2017
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Milica Orlandic, IES

Norwegian University of Science and Technology
Department of Electronic Systems

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

List of Figures

List of Tables

Appendixes:

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, 4½ 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 then 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 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.

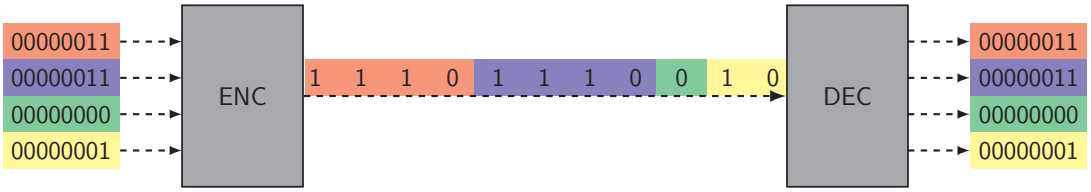


Figure 1: Simple Unary coding based compression of byte data.

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 $\{x_1, \ldots, x_n\}$ and with a accompanying probability mass function $P(X)$ as:

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

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} where 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 $\lceil \mathrm{H}(X) \rceil$.

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

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

Table 1: Entropy calculated assuming the distribution in the symbol string is representative of the actuall 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} an 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.

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

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

This simple model works on a per bin basis, while it is easily observable that there exists patters 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.

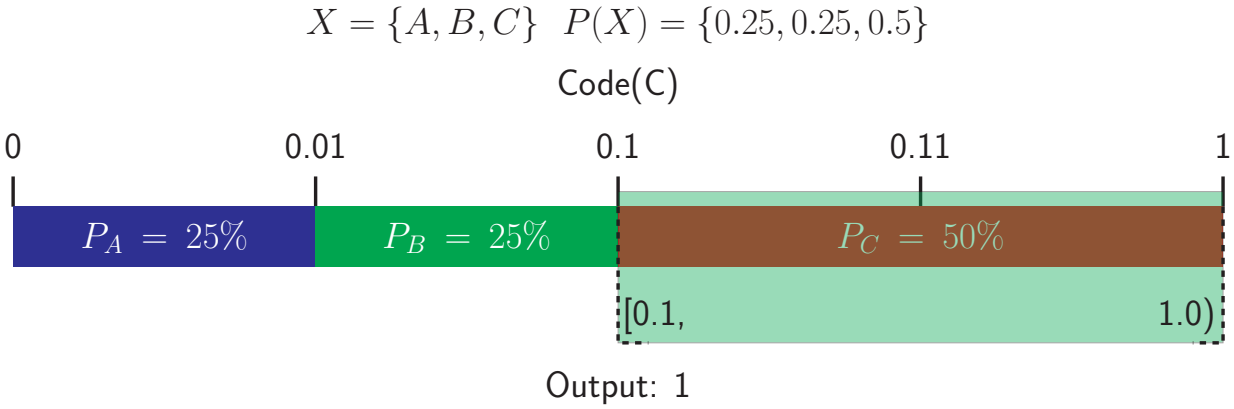


Figure 2: Encoding of a single symbol.

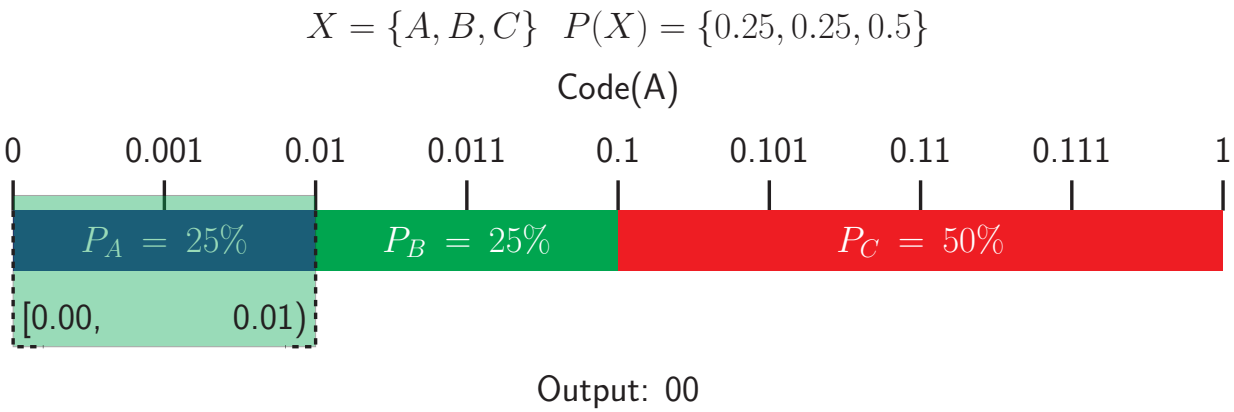


Figure 3: Encoding of a single symbol.

Figures 2 and 3 show how the different symbols are encoded. Where a 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.

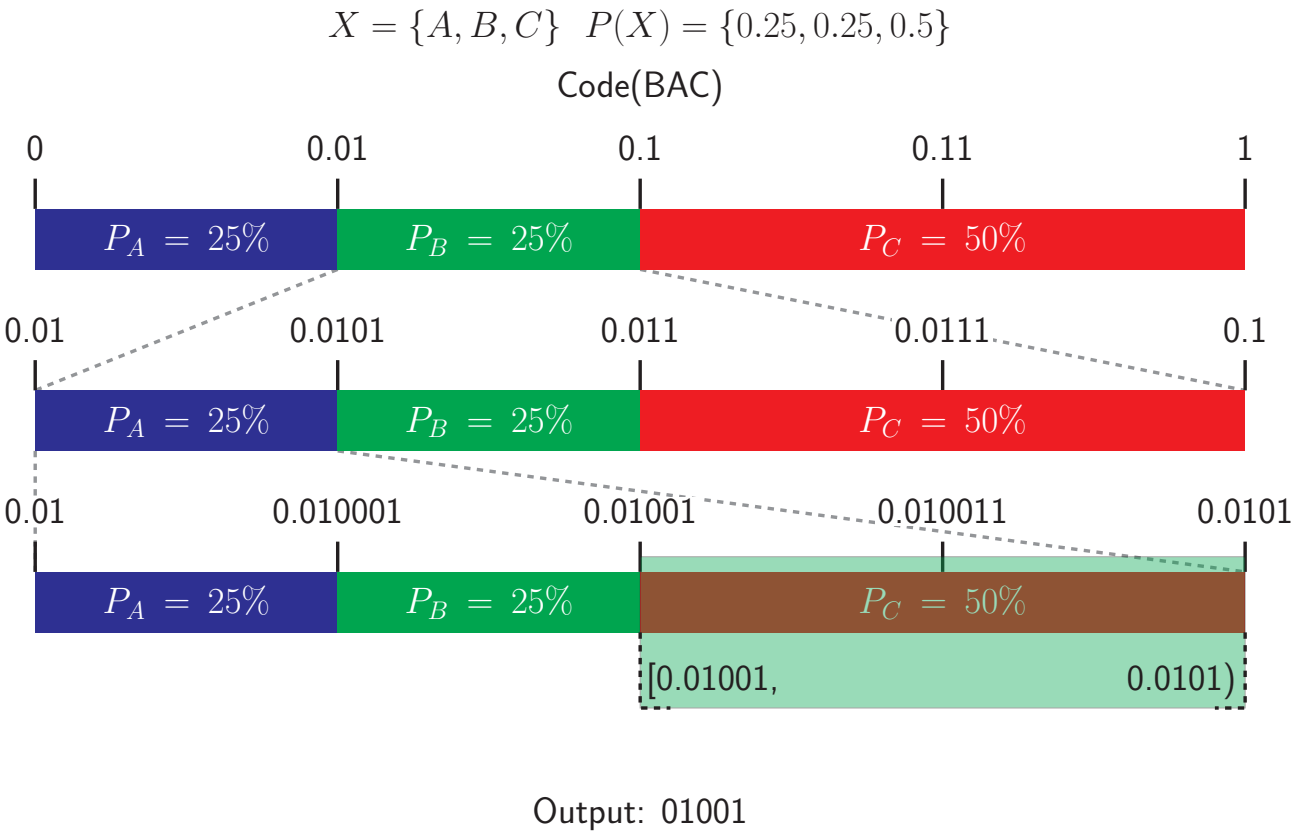


Figure 4: Sequential encoding of three symbols, using a static probability model.

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 uses the Macroblock structure, with a standard of 16×16 samples. For HEVC the macroblocks has 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 pixeldata 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 is 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 CTB 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 TB 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 though 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.

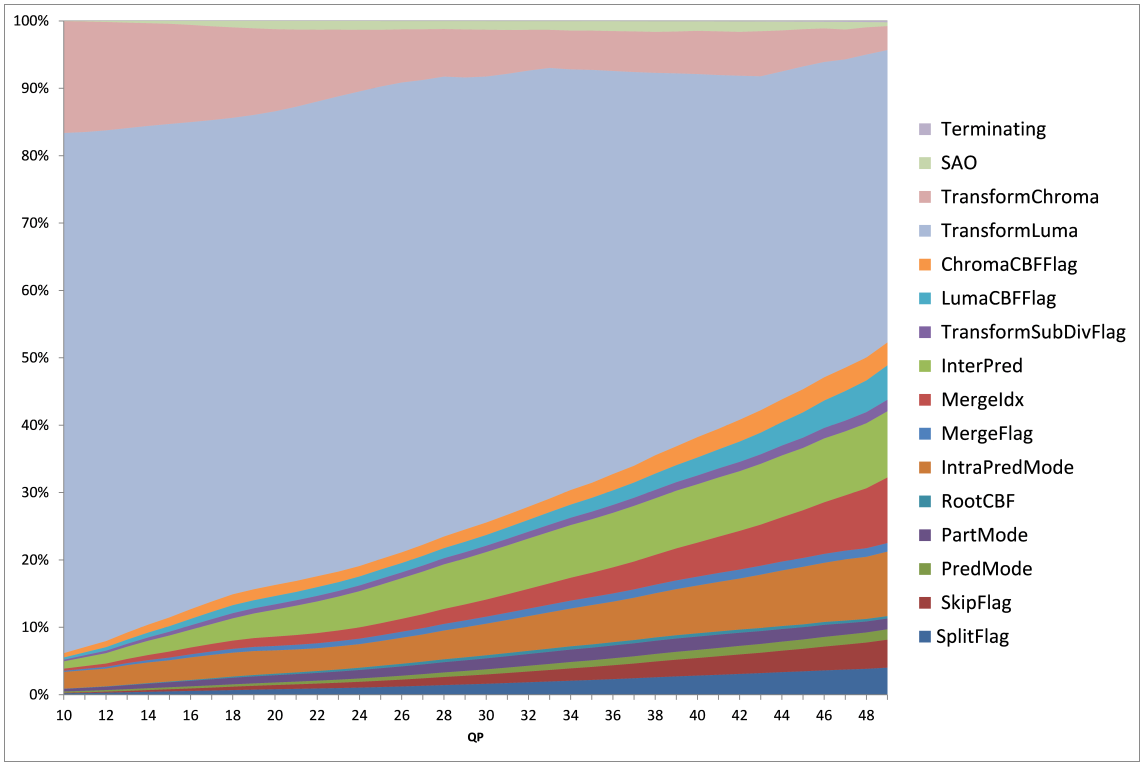


Figure 6: Bitrate distribution of Syntax Elements for varying levels of quantization in HEVC. Where quantization can be viewed as the level of lossy compression. Data shows contribution for encoding of all frame types.[6]

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.

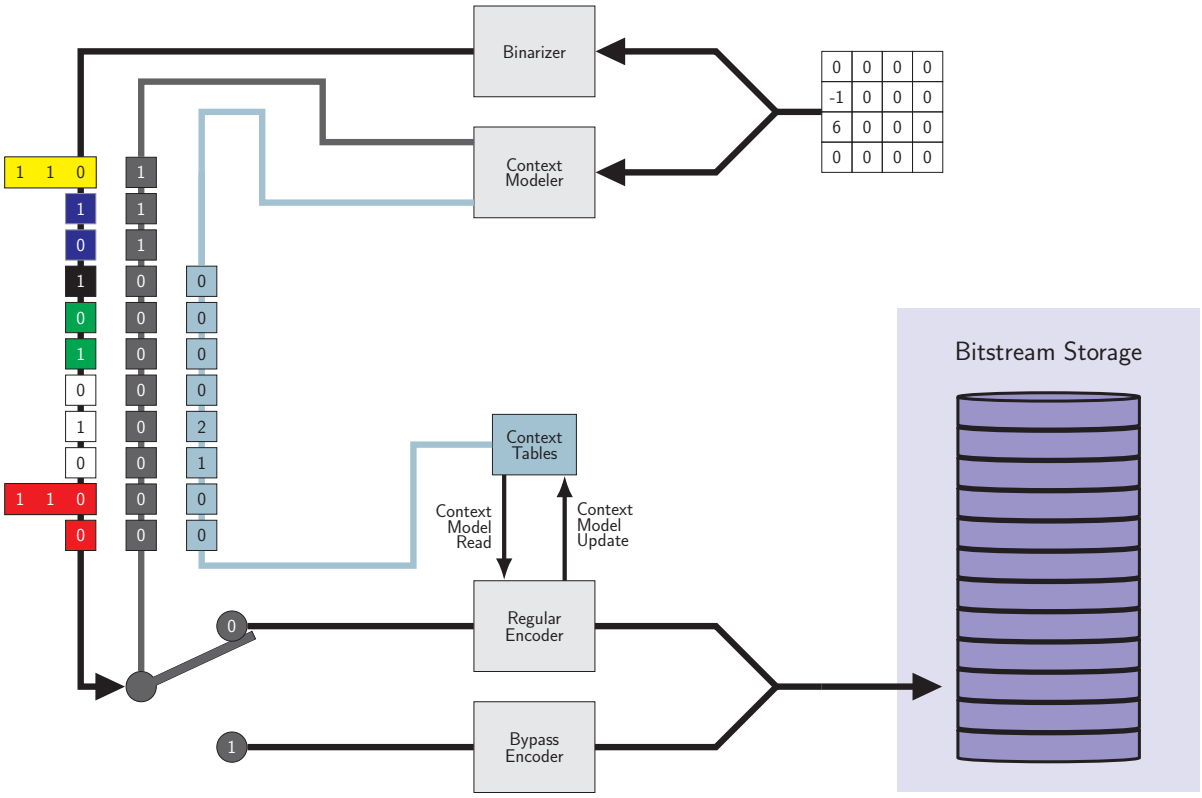


Figure 7: CABAC encoding process for a 4×4 Transform Block.

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.

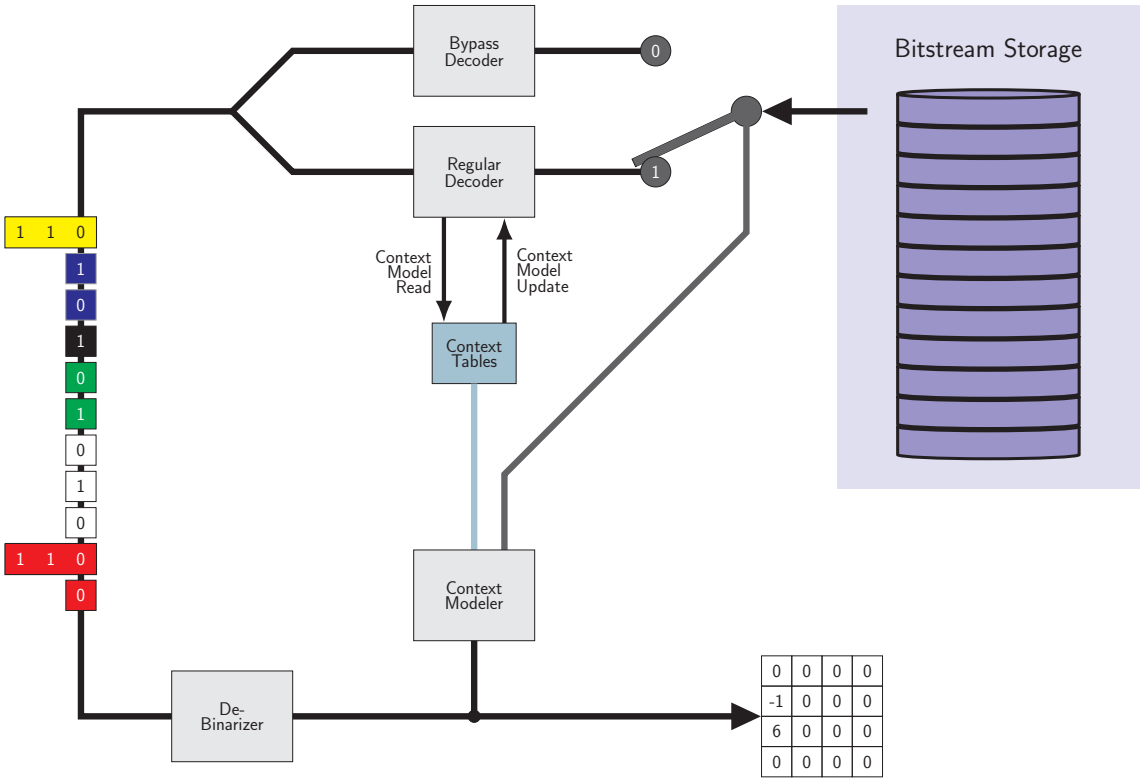


Figure 8: CABAC decoding process for a 4×4 Transform Block.

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.

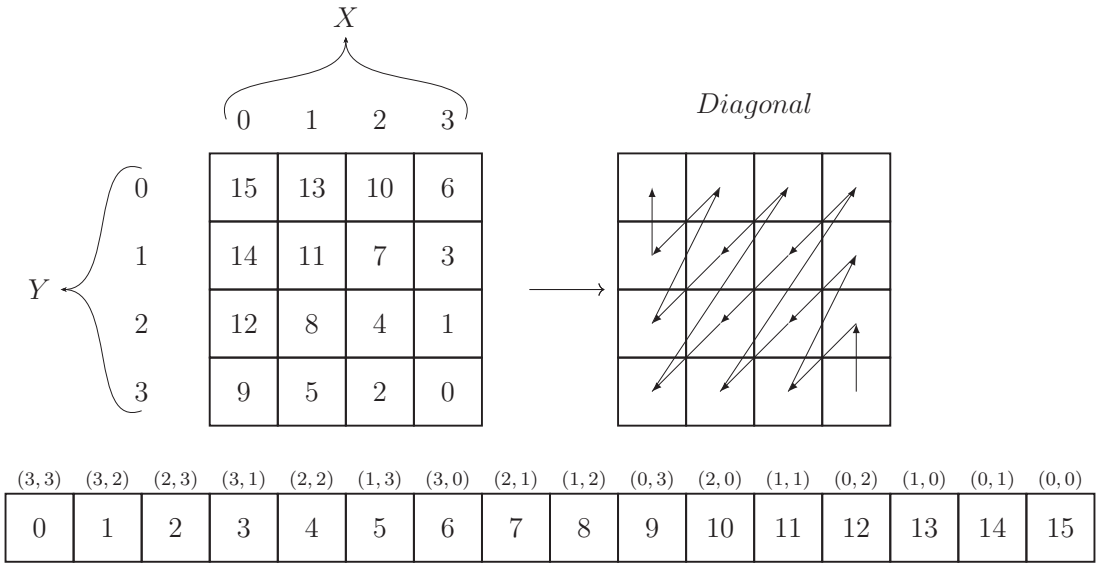


Figure 9: Scan direction for Diagonal scan.

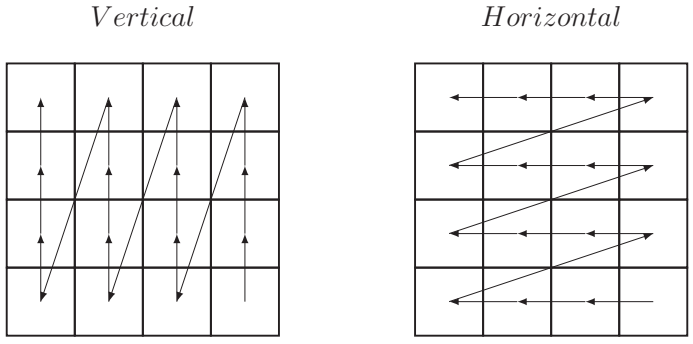


Figure 10: Scan direction for vertical and horizontal scans.

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.

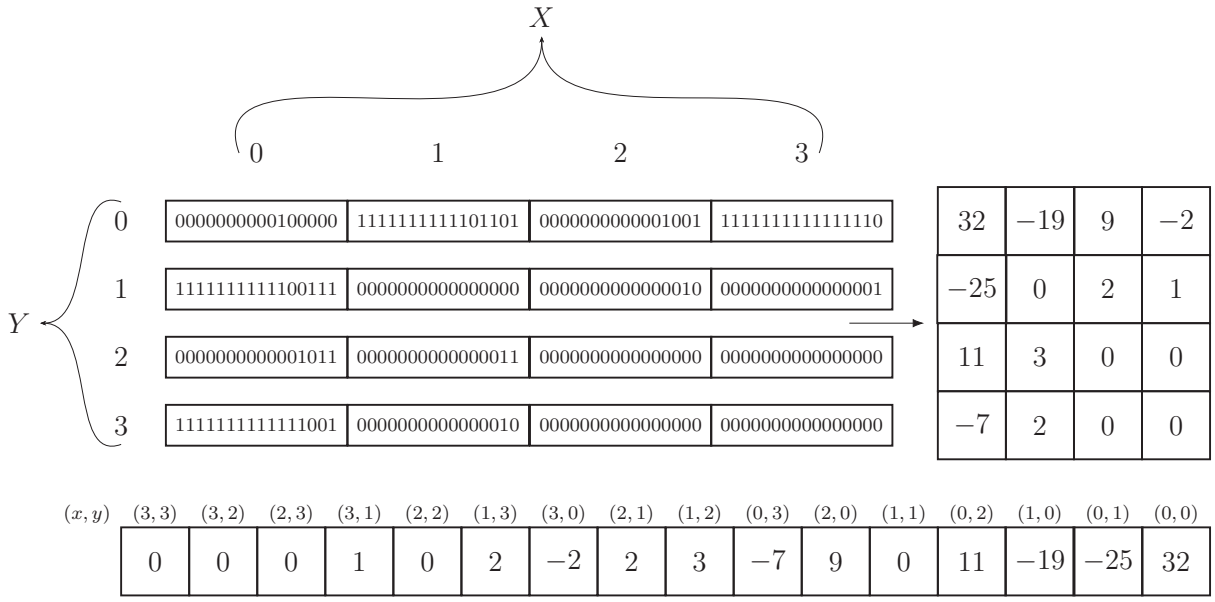


Figure 11: Residual data in the Transform Block to be Binarized

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 his will not be covered further here.

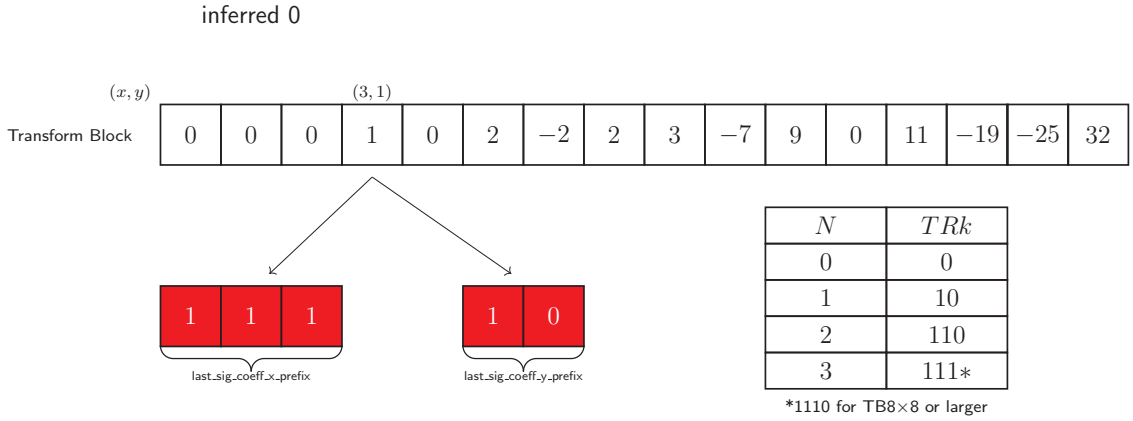


Figure 12: Binarization of last_sig_coeff_x/y_prefix in 4×4 Transform Blocks.

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.

$$Z = SIG + ALG1 + ALG2 + ALRem \tag{2}$$

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.

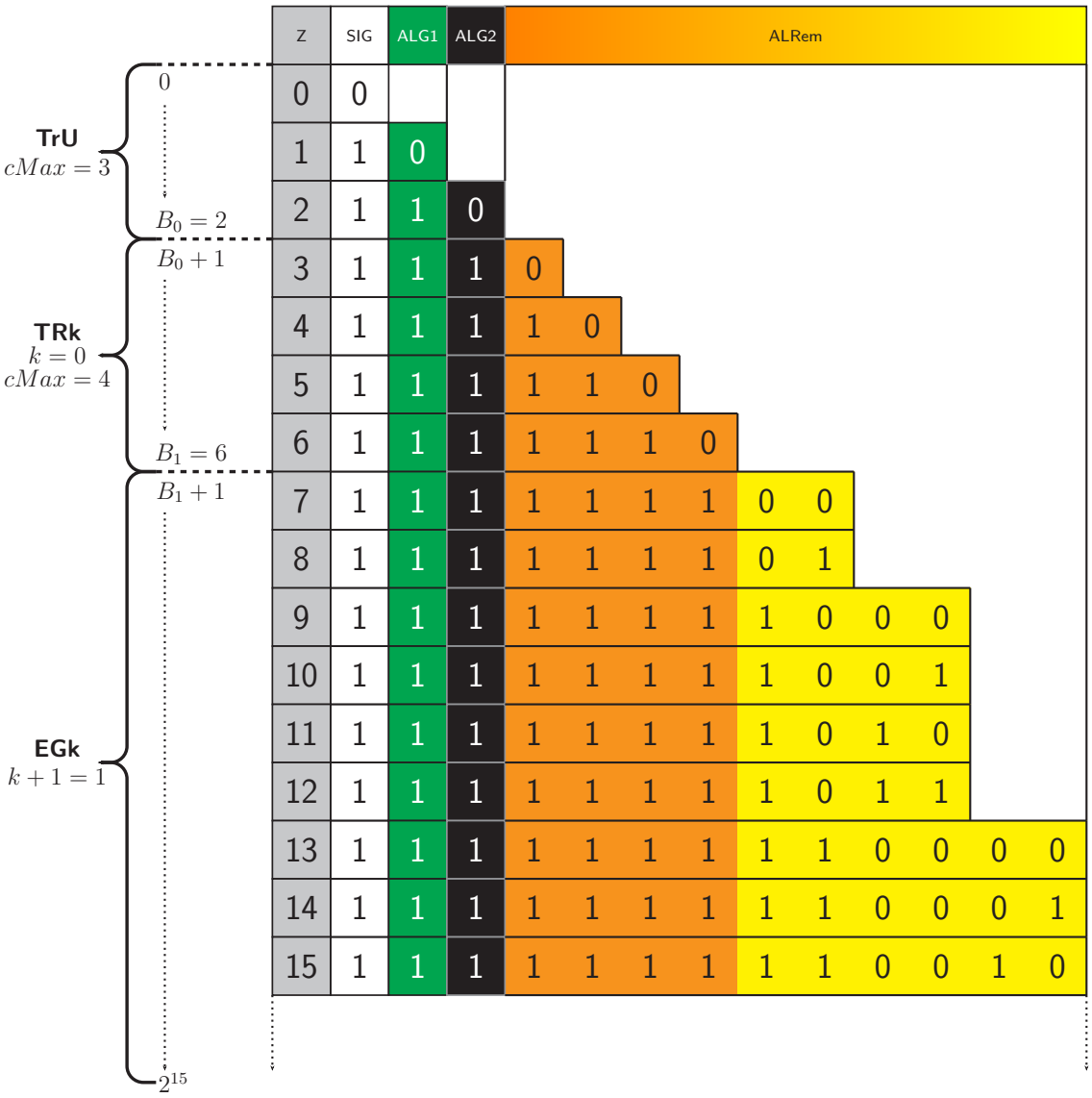


Figure 13: Binarization of Z from 0 to 15 using the SIG, ALG1, ALG2 and ALRem syntax elements. ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the initial state for the adaptive binarization scheme.

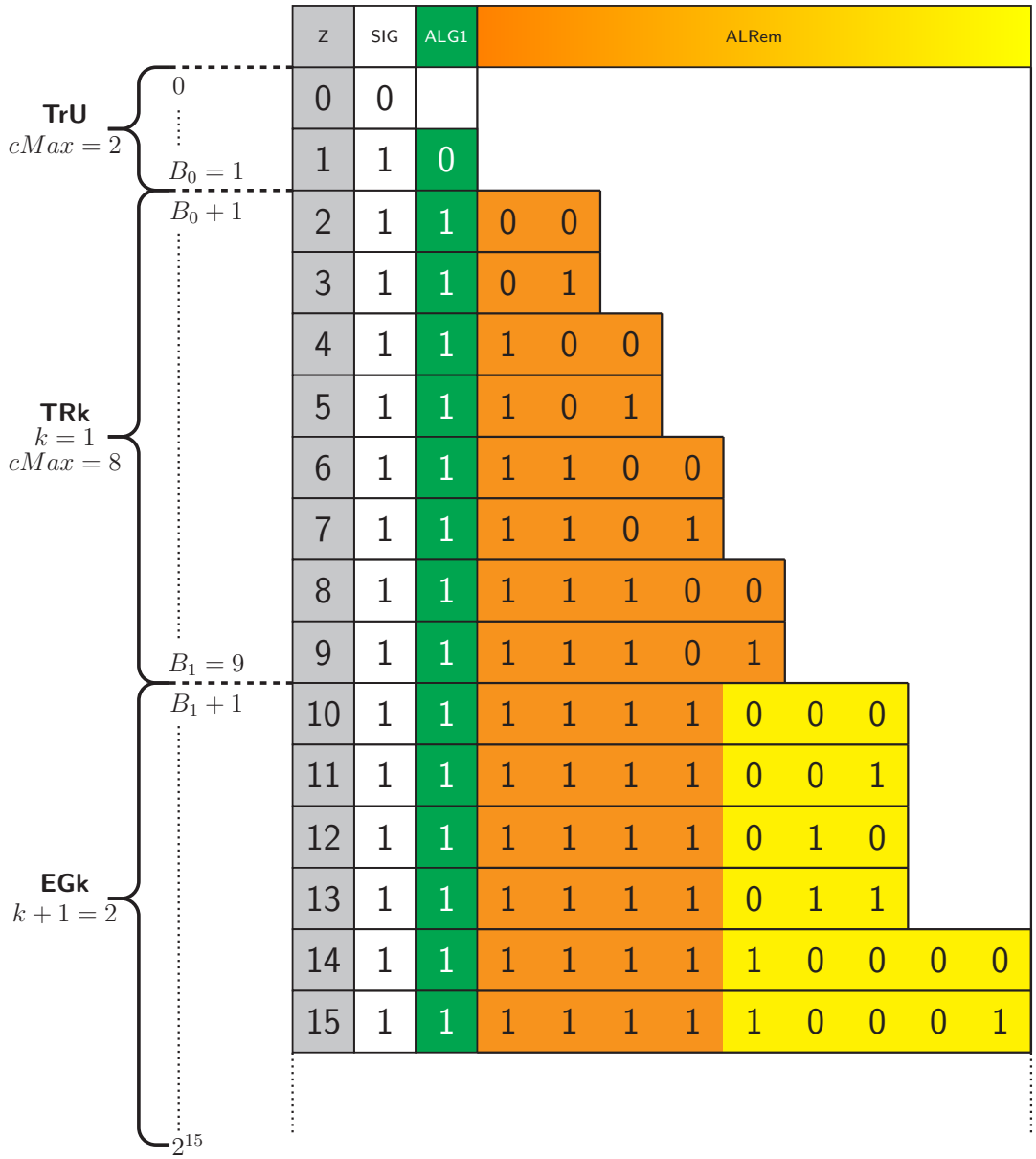


Figure 14: Binarization of Z from 0 to 15 using the SIG, ALG1, and ALRem syntax elements(ALG2 absent). ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the state after a first $Z > 1$ has occured, and before 8 total $Z > 1$ has occured. $Z > 3$ has also occured, as shown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

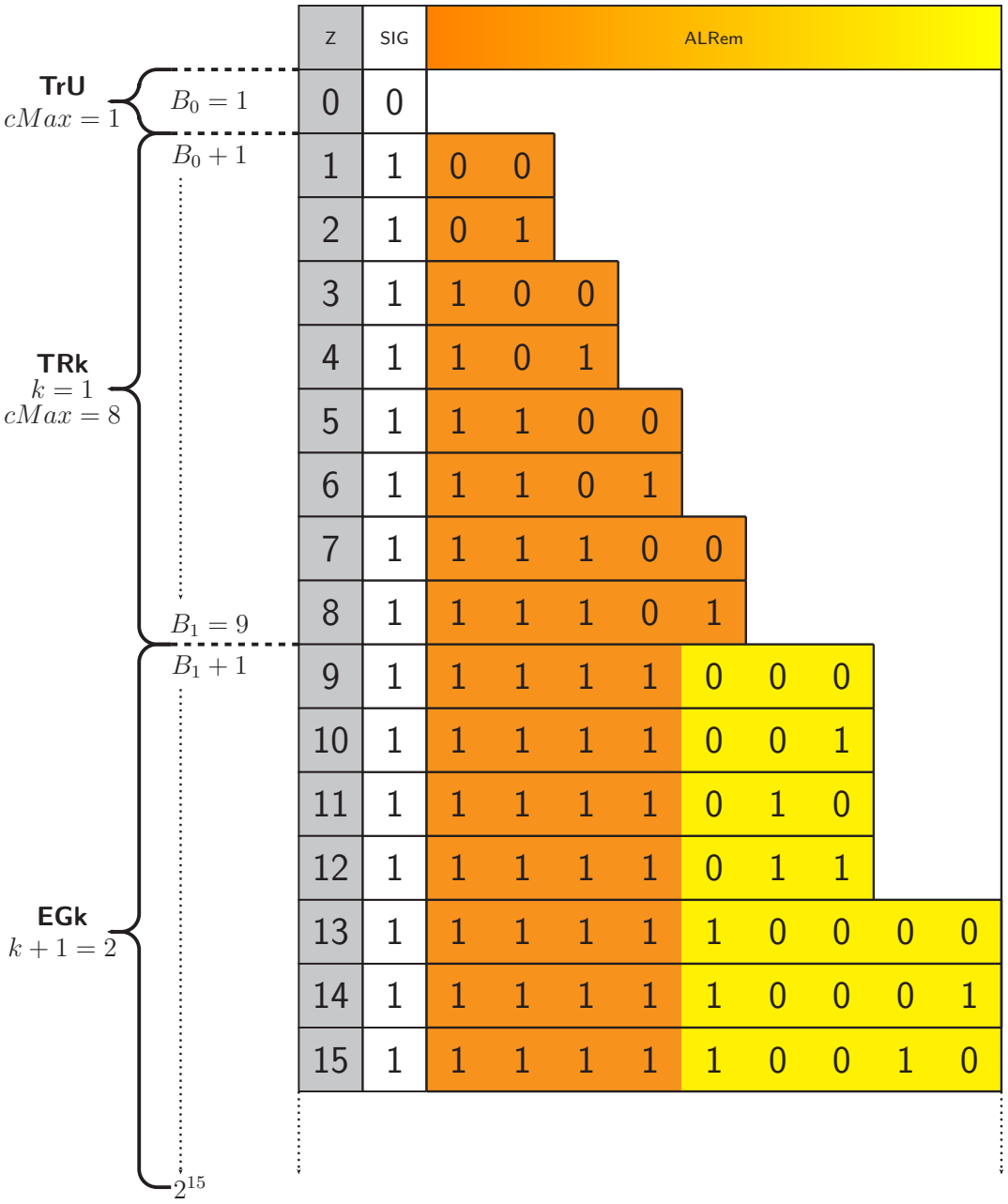


Figure 15: Binarization of Z from 0 to 15 using the SIG, ALG1, and ALRem syntax elements(ALG2 inferred 0). ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the state after a total of 8 $Z > 1$ has occured. $Z > 3$ has also occured, as shown by the fact that the adaptive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

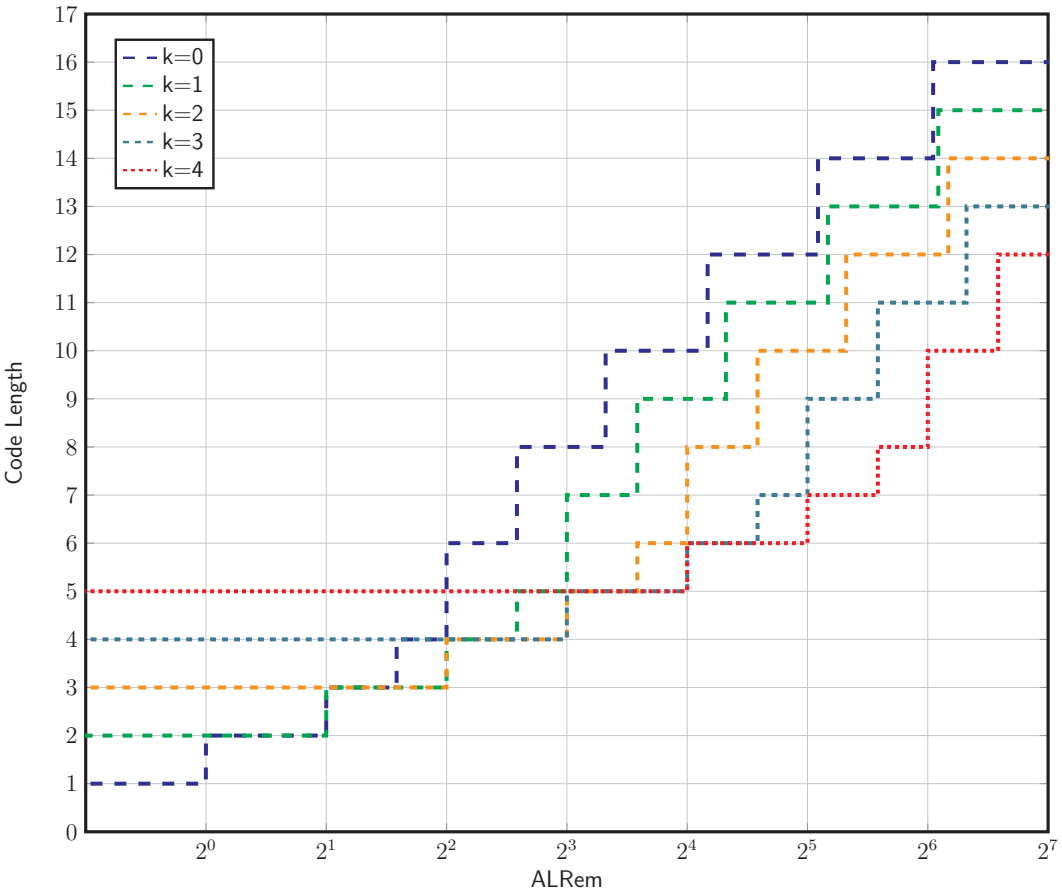


Figure 16: ALRem code length for each value of the adaptive variable k.

4.9 Coding of Sign

Sign is coded using the SIGN(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 Trandsform block, totaling 16-bits$\times 4 \times 4 = 256$-bits, is now reduced down to 81-bits using the HEVC adaptive binariaztion scheme. The size is now equal to about 32% of the original size, even before CABAC encoding is performed. Notice also how the symbols is biased towards '1' for regular coded bins(SIGN, SIG, ALG1 and ALG2). Counting 21 total '1's and 5 total '0'. This leaves a probabiliy 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.

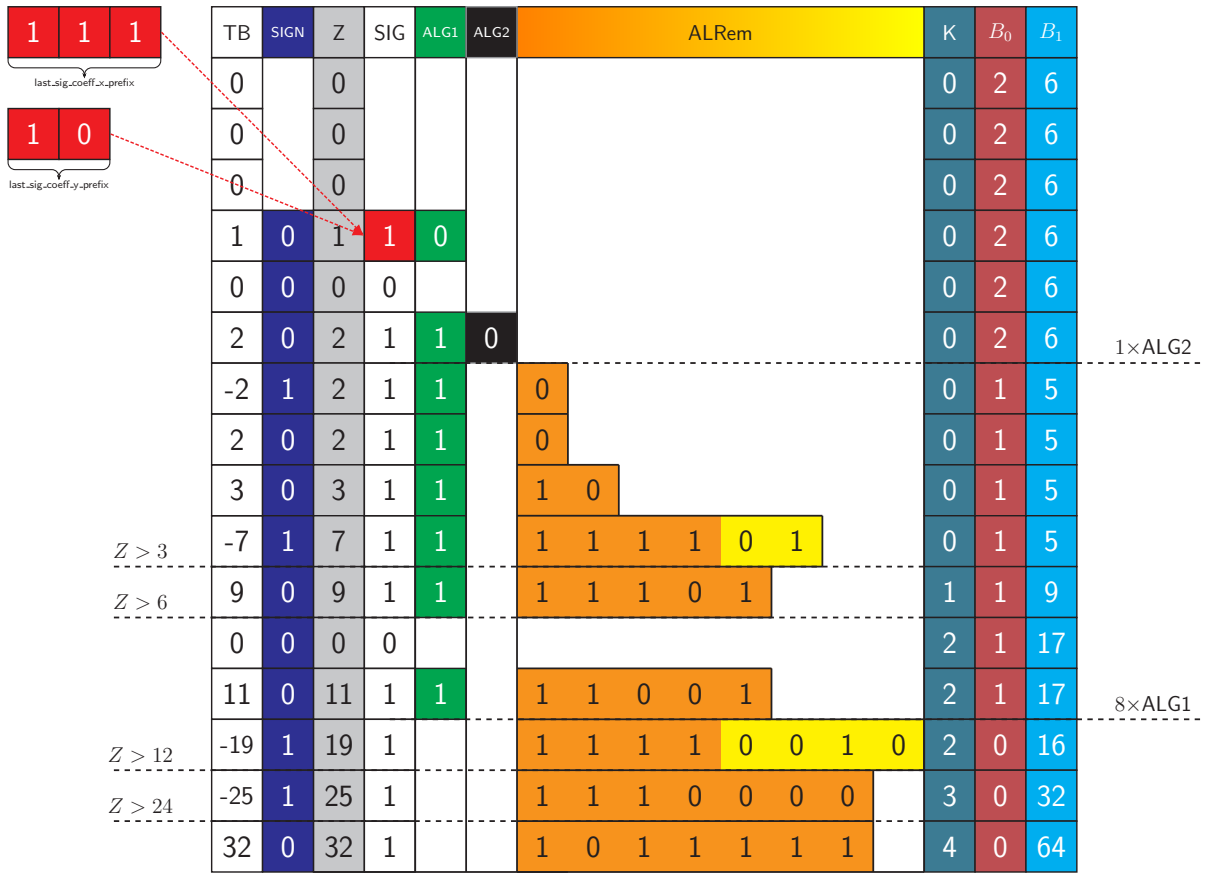


Figure 18: Complete Example, showing how the Transform Block from Figure 11 is binarized using the Diagonal Scan direction. Threshold variable values for each scan, as well as the condition that triggers a transition is also shown. Note that the red square under SIG is not signaled, but inferred directly 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 in 4×4 TBs uses a position based context selection. The context selection of SIG for larger transform blocks uses a substantialy more advanced method, where context index calculation is based on a selection of previously processed bins.

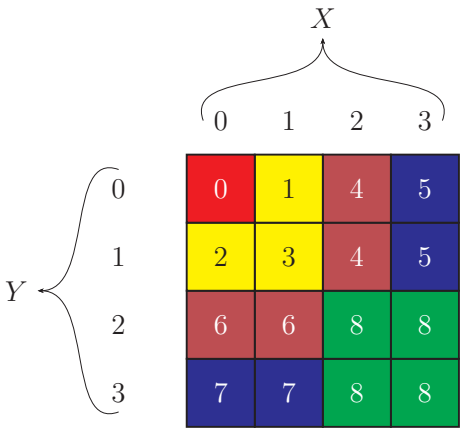


Figure 19: SIG context assignment for 4×4 Transform Blocks.[9]

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.

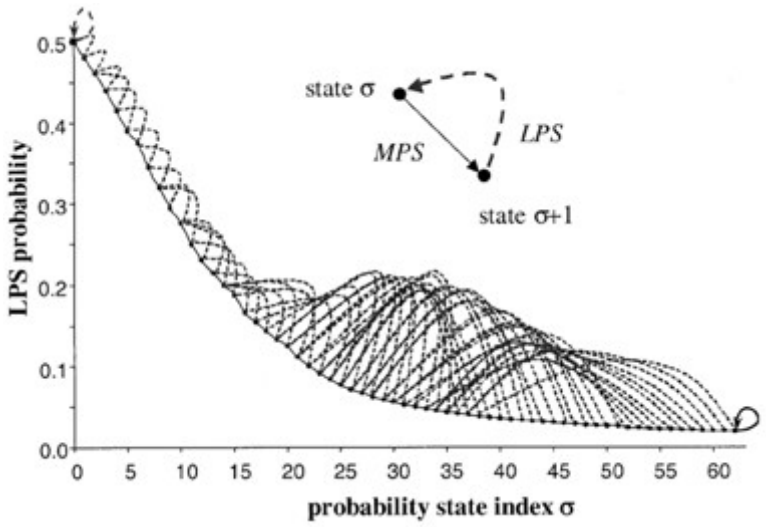


Figure 20: FSM based probability estimation in CABAC. Figure is taken from a presentation for CABAC in h.264.[3] Dotted lines are transitions performed when the bin to be encoded is not equal to valMps, and solid lines are transitions performed when the bin is equal to valMps.

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 SliceQPY variables. initType and SliceQPY can be seen as simple inputs to the context modeler. With initType ranging from 0-2 and SliceQPY 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 outputing 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 initalization/reset of the encoder or decoder should these variables be set to their initial values.

	Initial Value	
Variable	**Decoder**	**Encoder**
codIRange	510	510
codILow	0	0
qCodIRangeidx	0	0
CodIrangeLPS	0	0
codIOffset	0	read_bits(9)
codILow	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.

6.3 Encoding a Decision

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.

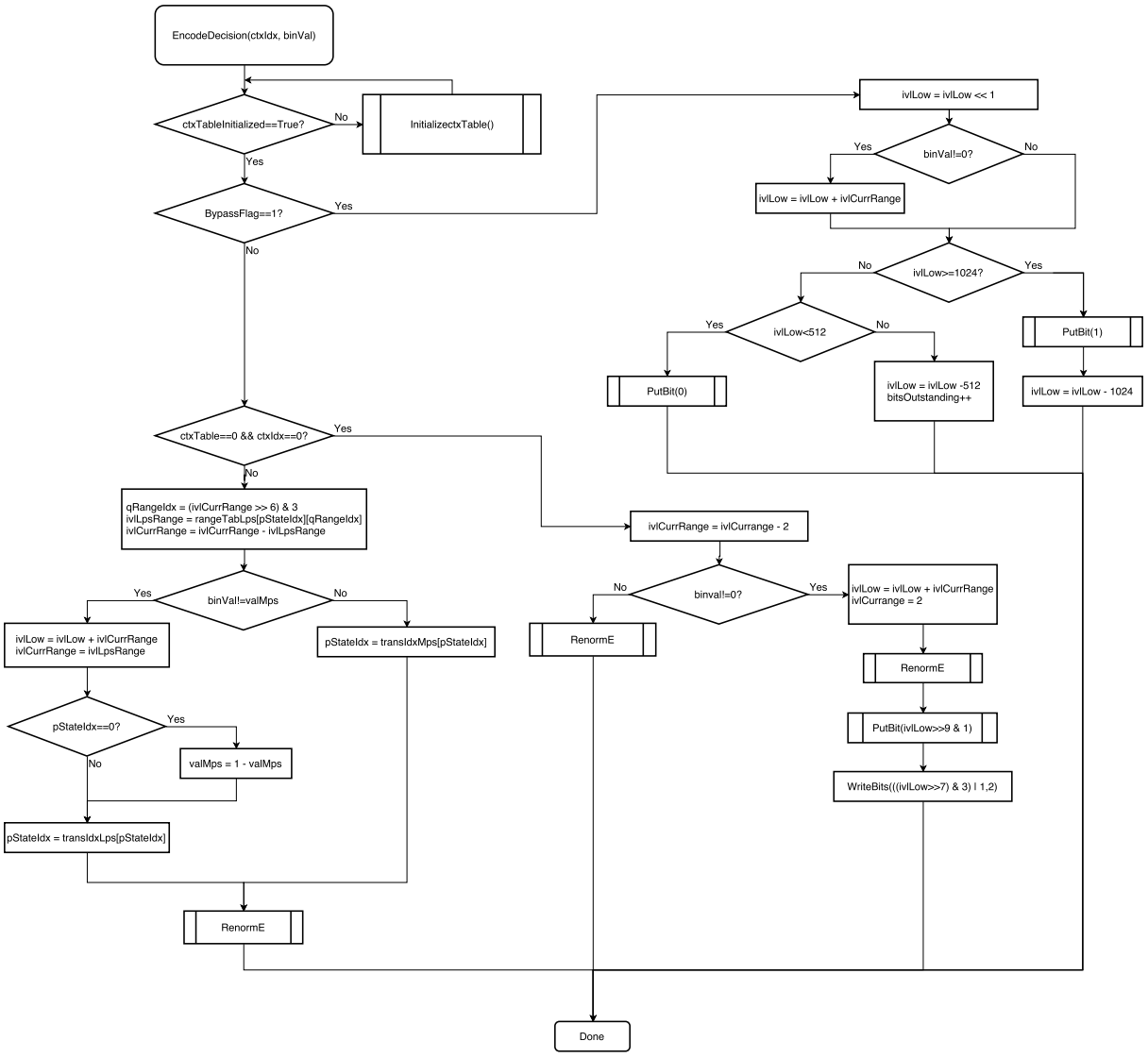


Figure 21: Encoding of a binary decision. Note that Termination is started when ctxTable==0 && ctxIdx==0. Effectively signaling the special termination context. The hardware implementation uses a simple termination flag to achieve the same result.

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.

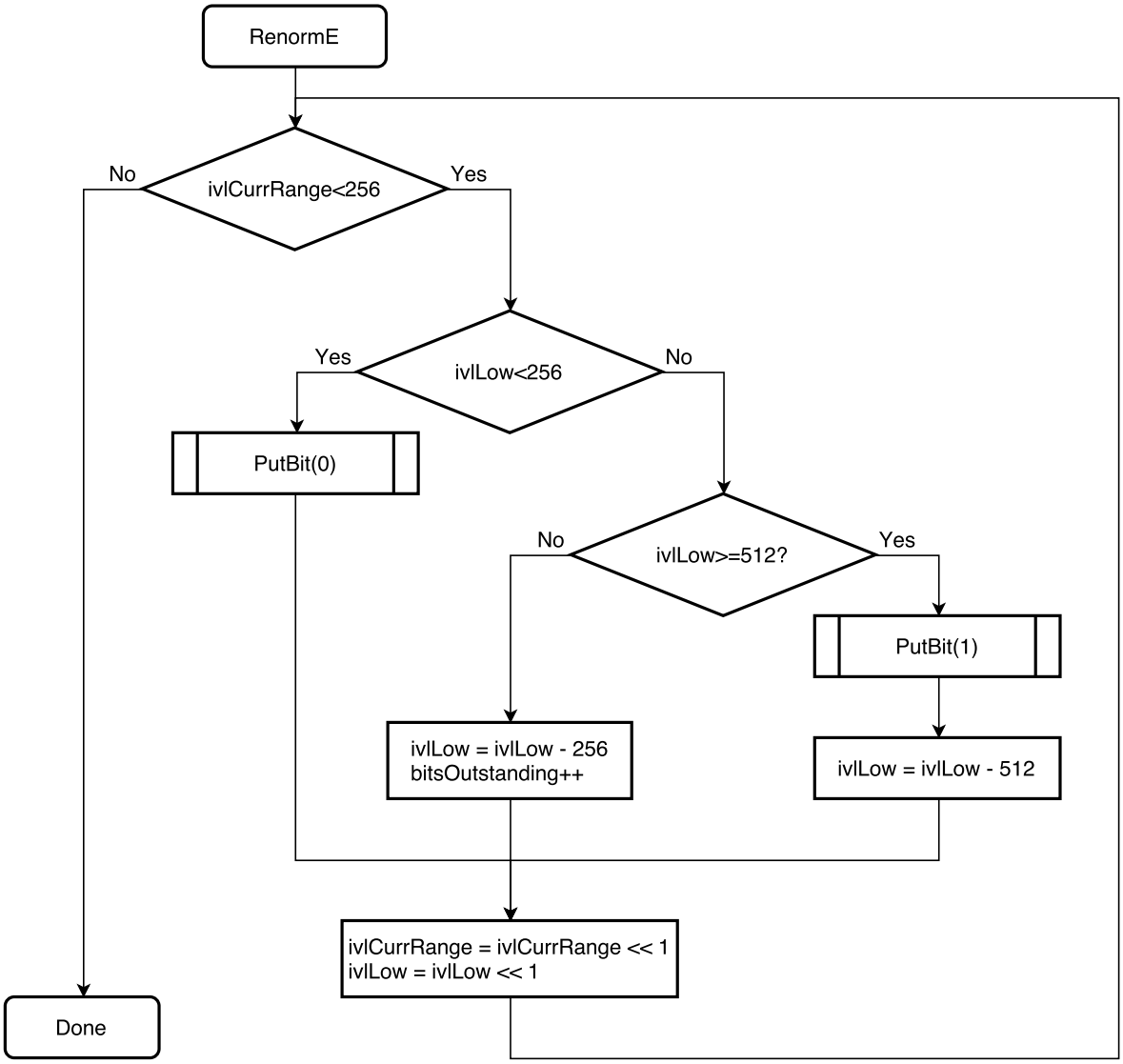


Figure 22: Renormalization during encoding of a binary decision.

6.5 Writing to Bitstream

Writing to bistream is done using the ButBit 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.

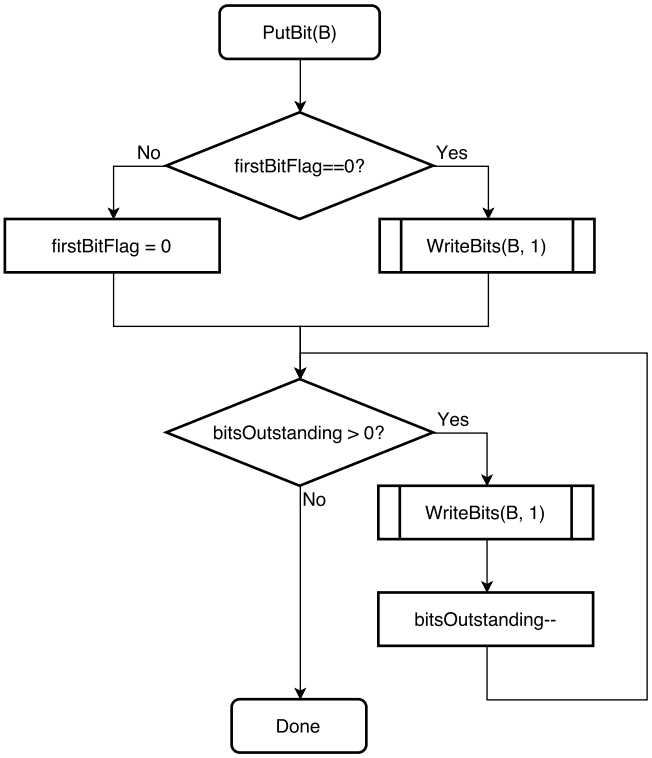


Figure 23: PutBit function. See Section 6.6 WriteBits function description.

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 essentialy contain information about the next sub-interval.

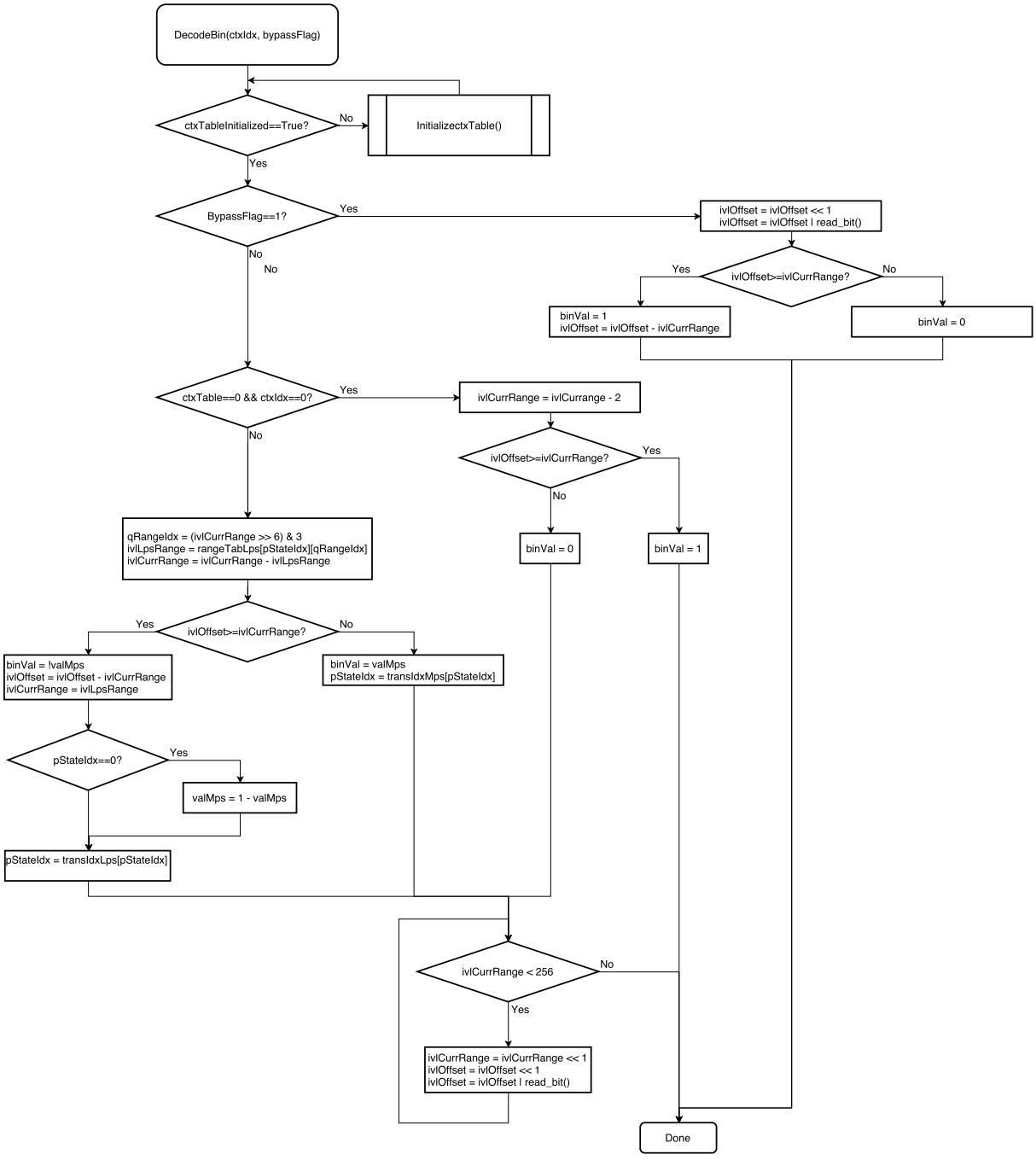


Figure 24: Decoding of a binary decision.

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.

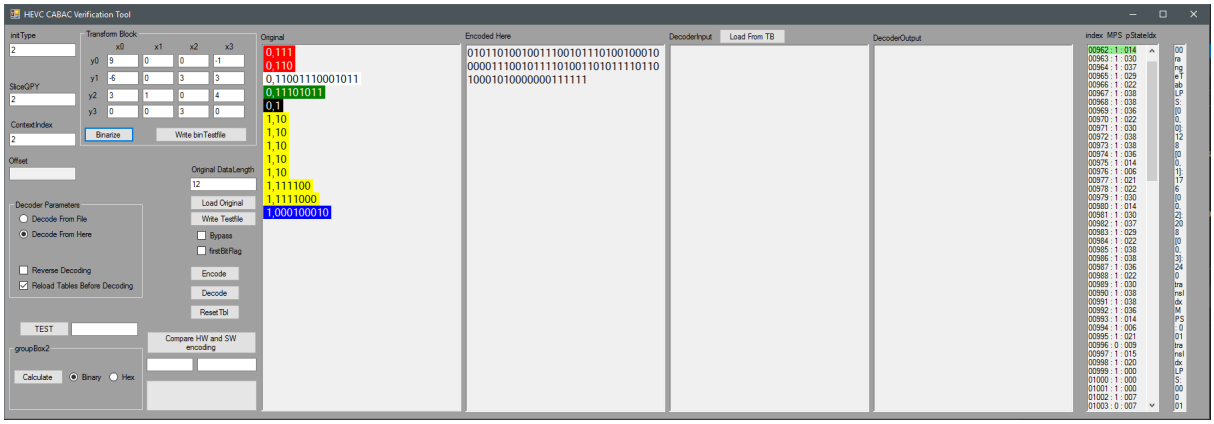


Figure 25: Software model user interface.

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 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 SliceQPY, 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 planed 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.

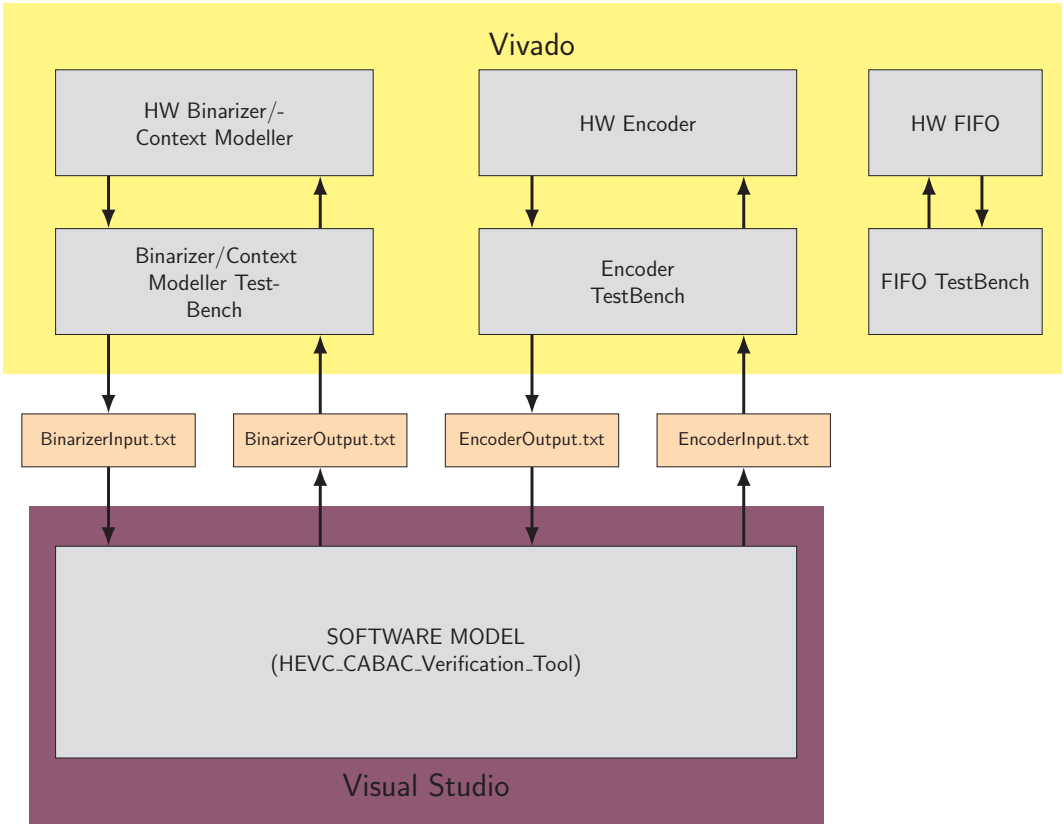


Figure 26: System structure for interfacing with between hardware and software. Including development platforms. Note that the Binarizer/Context Modeler was not completed. **Appendix G** covers setup for the completed system, as well as a tutorial on how Hardware/-Software Encoding can be compared.

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

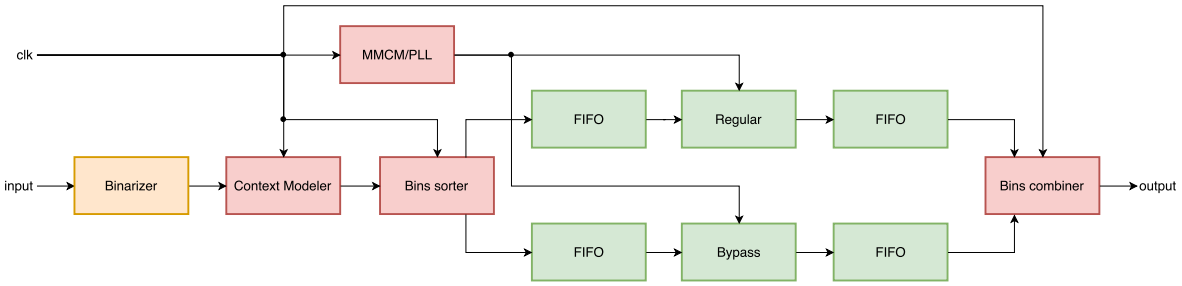


Figure 27: The planed modules for the completed CABAC encoder system. Implemented modules in green, partially implemented modules in orange, and incompleted modules in red. Note that the Regular and Bypass encoder is combined in the current hardware design.

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 where 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 uses 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 where 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.

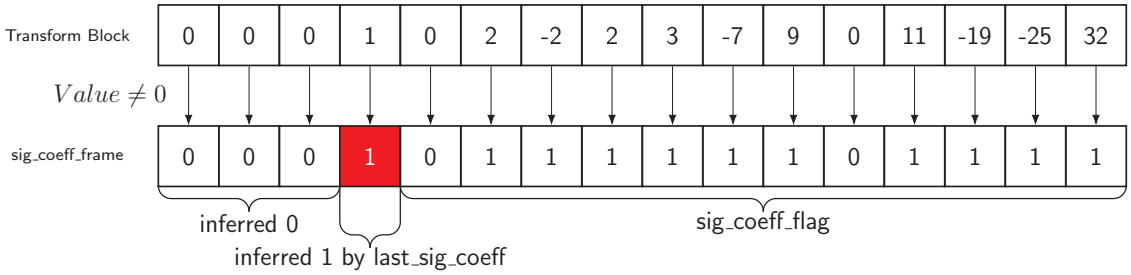


Figure 28: Binarization of sig_coeff_flag

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.

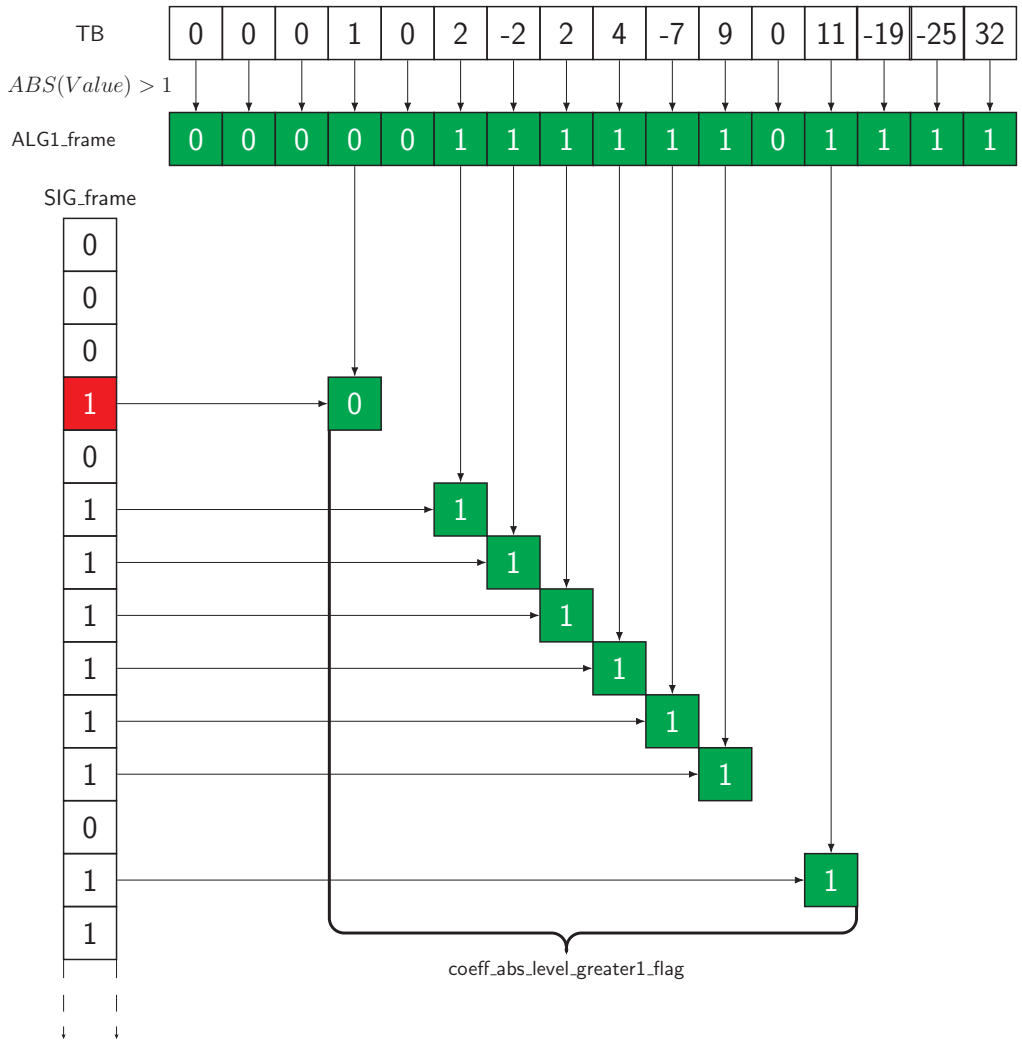


Figure 29: Binarization of coeff_abs_level_greater1_flag

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.

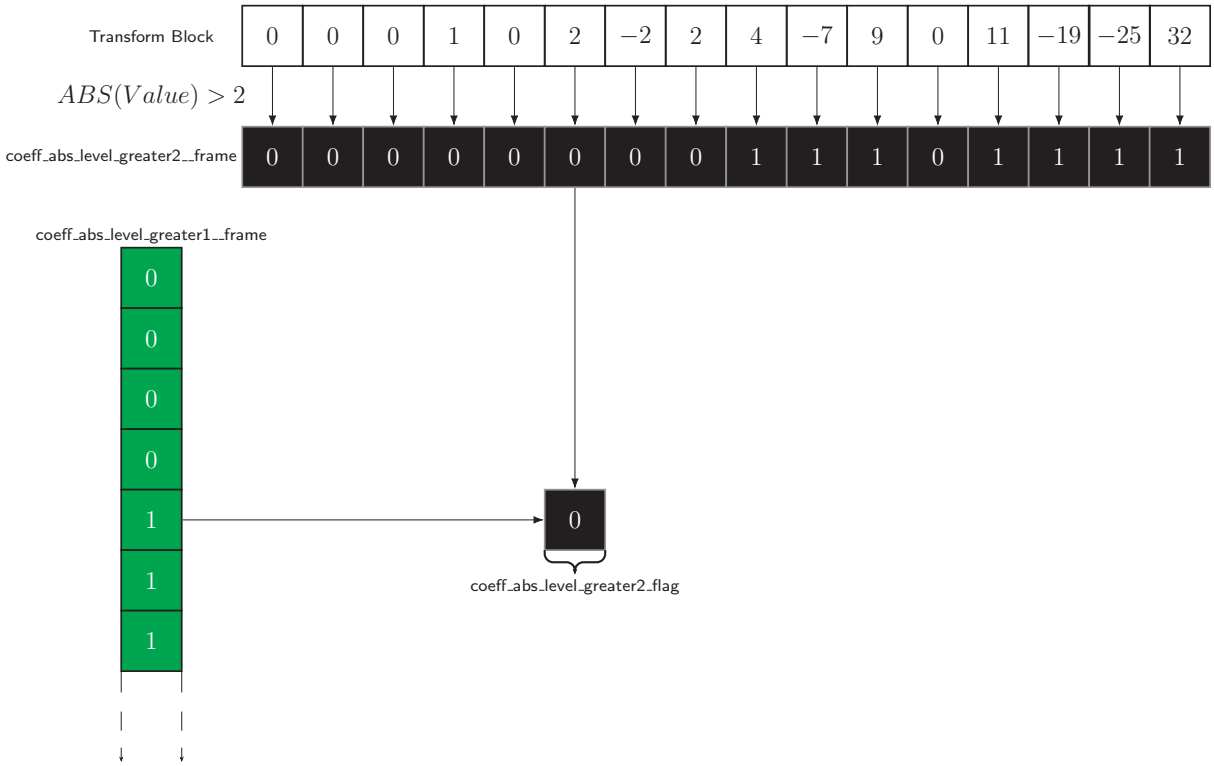


Figure 30: Binarization of coeff_abs_level_greater2_flag

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	xxxxxxxxC	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	xxxxxxxxxxC	14	10 + k	4
14	$2^k\times(2^11+2)$	$2^k\times(2^12+2)-1$	111111111111110	xxxxxxxxxxxC	15	11 + k	3
15	$2^k\times(2^12+2)$	$2^k\times(2^13+2)-1$	1111111111111110	xxxxxxxxxxxxC	16	12 + k	2
16	$2^k\times(2^13+2)$	$2^k\times(2^14+2)-1$	11111111111111110	xxxxxxxxxxxxxC	17	13 + k	1
17	$2^k\times(2^14+2)$	$2^k\times(2^15+2)-1$	111111111111111110	xxxxxxxxxxxxxxC	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.

TabIdx	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 TabIdx is a binary number with lenght indicated by the Suffix Length at that specific TabIdx

TabIdx	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

TabIdx	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.

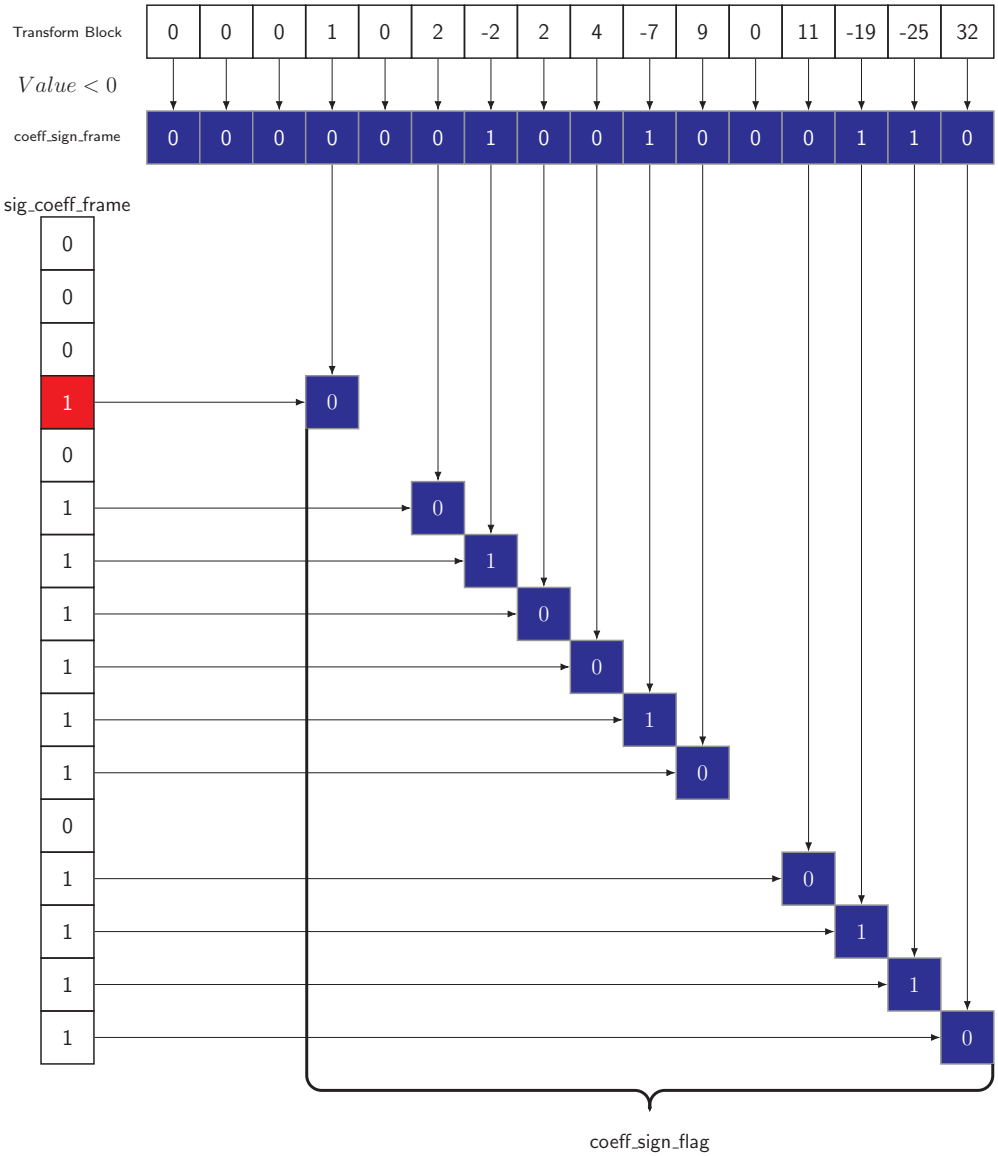


Figure 31: Binarization of coeff_abs_level_sign_flag

8.5 Context Index Calculator

The context index calculator module was not implemented. Although this module was first planed to 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.

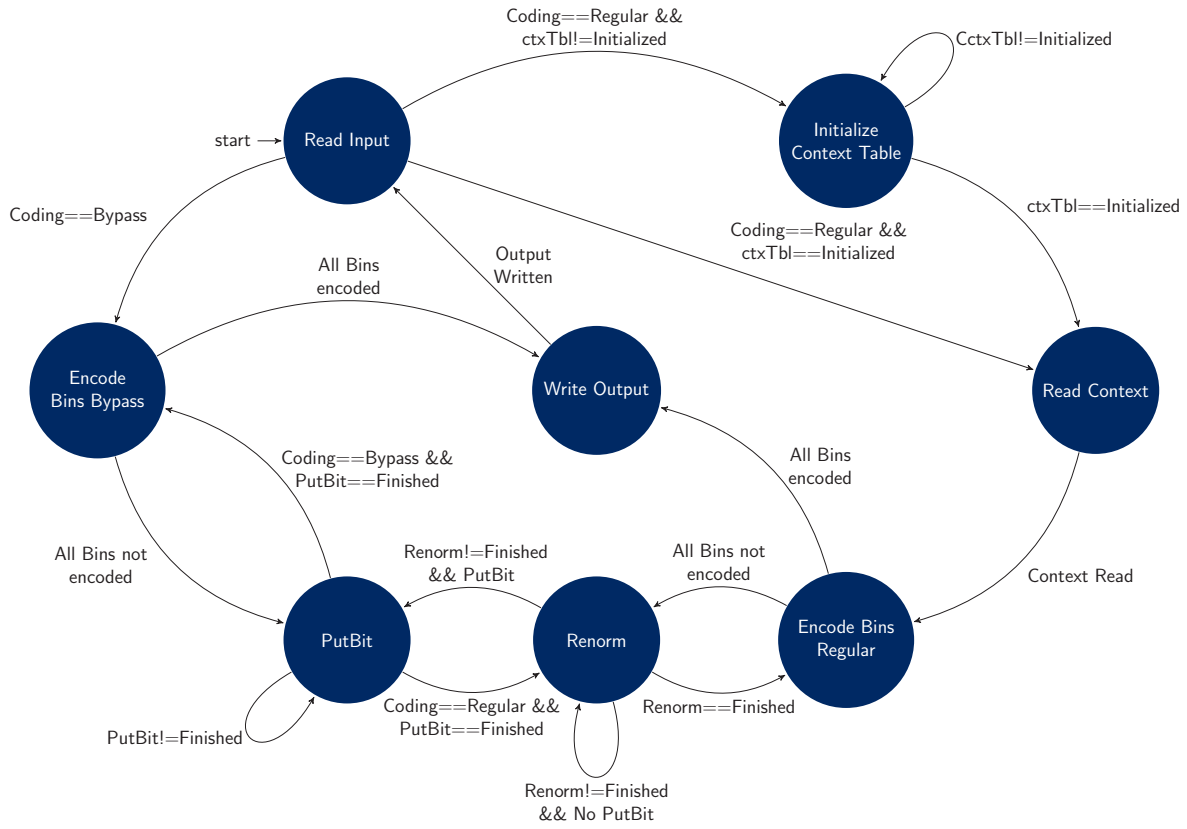


Figure 32: Simplified state machine diagram for the hardware encoder. Context update is written during the Write Output state.

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 where 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 35. 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

		Context Table Index		
Syntax Element	**Binarization Process**	**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 intialization variables for all values of SliceQPY and initType.
ctxIdxTable	120	Initialized from reading ctxIdxTableInitials for a given value of SliceQPY 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 pStateIdx(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 planed 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
pStateIdx	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 Fmax = 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 Fmax = 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.

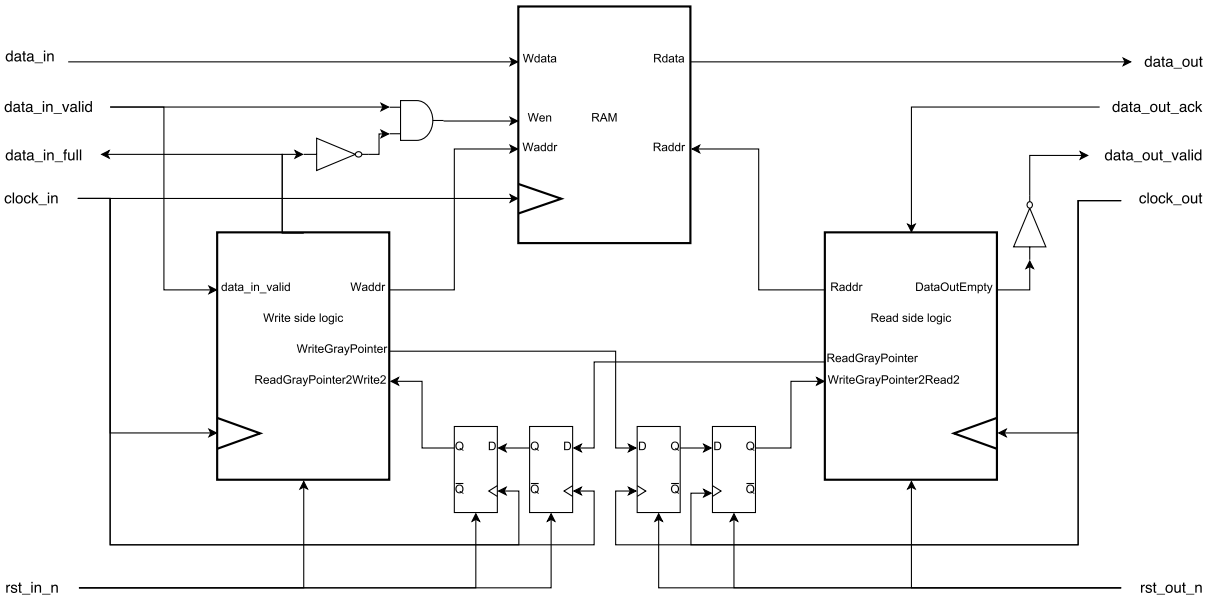


Figure 33: Asynchronous fifo.

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 standardsincluding high efficiency video coding (hevc). http://iphome.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. Hevc 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; --
    ↪  http://www1.pldworld.com/@xilinx/html/technote/tool/manual/15i_doc/fndtn/vhd/vhd10_3.htm
24
25 library work;
26 use work.BinarizerParameters.all;
27
28 entity Binarizer is
29
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
    ↪  (2**(coeffwidth-1));
65
66     signal BinarizeState : BinarizeStateType;
67
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))  <=
                             ↪   to_integer(abs(signed(DataIn(((CoeffWidth*(i+1))-1)   downto
                             ↪   (CoeffWidth*i)))));
156
157                         -- ------------------------------------------------------------------------
158                         -- Read sign.
159                         -- ------------------------------------------------------------------------
160                         coeff_sign_flag(Scan_Direction(15-i))   := DataIn((CoeffWidth*(i+1))-1);
161
162                         -- ------------------------------------------------------------------------
163                         -- Find non-zero data.
164                         -- ------------------------------------------------------------------------
165                         sig_coeff_flag(Scan_Direction(15-i))  <= or_reduce(DataIn(((CoeffWidth*(i+1))-2)
                             ↪   downto  (CoeffWidth*i)));
166
167                         -- ------------------------------------------------------------------------
168                         -- Find >1 data and >2 data. Procedure dependent on sign bit.
169                         -- ------------------------------------------------------------------------
170                         case DataIn((CoeffWidth*(i+1))-1) is
171                             when '0' =>
172                                 coeff_abs_level_greater1_flag(Scan_Direction(15-i))    <=
                                     ↪   or_reduce(DataIn(((CoeffWidth*(i+1))-2)  downto
                                     ↪   ((CoeffWidth*i)+1)));
173                                 coeff_abs_level_greater2_flag(Scan_Direction(15-i))    <=
                                     ↪   (DataIn(CoeffWidth*i)   and DataIn((CoeffWidth*i)+1))
174                                                                                         or
                                                                                             ↪   or_reduce(DataIn(((CoeffWidth*(
                                                                                             ↪   downto
                                                                                             ↪   ((CoeffWidth*i)+2)));
175                             when '1' =>
176                                 coeff_abs_level_greater1_flag(Scan_Direction(15-i))    <=
                                     ↪   nand_reduce(DataIn(((CoeffWidth*(i+1))-2)  downto  (CoeffWidth*i)));
177                                 coeff_abs_level_greater2_flag(Scan_Direction(15-i))    <=
                                     ↪   (DataIn(CoeffWidth*i)   and (not(DataIn((CoeffWidth*i)+1))))
178                                                                                         or
                                                                                             ↪   nand_reduce(DataIn(((CoeffWidth
                                                                                             ↪   downto((CoeffWidth*i)+2)));
179                             when others =>
180                         end case;
181                     end loop;
182
183                 when write_last_sig_coeff_x_prefix =>
184
185                     for i in 0 to 15 loop
186                         report "coefficients(" & integer'image(i) & "): " &
                             ↪   integer'image(coefficients(i));
187                     end loop;
188
189                     report "sig_coeff_flag: " & integer'image(conv_integer(sig_coeff_flag));
190                     report "coeff_abs_level_greater1_flag: " &
                         ↪   integer'image(conv_integer(coeff_abs_level_greater1_flag));
191                     report "coeff_abs_level_greater2_flag: " &
                         ↪   integer'image(conv_integer(coeff_abs_level_greater2_flag));
192                     report "coeff_sign_flag: " & integer'image(conv_integer(coeff_sign_flag));
193
194                     BinarizeState <= write_last_sig_coeff_y_prefix;
195                     write_last_sig_coeff_x_prefix_loop : for index in 0 to 15 loop
196                         if(sig_coeff_flag(index) = '1') then
197                             case index is
198                                 when 0  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
199                                             DataLength <=
                                                 ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
```

```
                        when 1  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 2  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 3  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 4  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 5  =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
                        when 6  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 7  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 8  =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
                        when 9  =>  DataOut(OutputWidth-1)                      <= '0';
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
                        when 10 =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 11 =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
                        when 12 =>  DataOut(OutputWidth-1)                      <= '0';
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
                        when 13 =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
                        when 14 =>  DataOut(OutputWidth-1)                      <= '0';
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
                        when 15 =>  DataOut(OutputWidth-1)                      <= '0';
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
                    end case;
                    exit write_last_sig_coeff_x_prefix_loop;
                end if;
            end loop;
        when write_last_sig_coeff_y_prefix =>
            BinarizeState <= write_sig_coeff_flag;
            write_last_sig_coeff_y_prefix_loop : for index in 0 to 15 loop
                if(sig_coeff_flag(index) = '1') then
                    case index is
                        when 0  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 1  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 2  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 3  =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
                        when 4  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 5  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
                                    DataLength <=
                                     ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
                        when 6  =>  DataOut(OutputWidth-1)                      <= '0';
```

```
252                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
253                             when 7  =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
254                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
255                             when 8  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
256                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
257                             when 9  =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
258                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
259                             when 10 =>  DataOut(OutputWidth-1)                      <= '0';
260                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
261                             when 11 =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
262                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
263                             when 12 =>  DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
264                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(3,OutputWidthLength));
265                             when 13 =>  DataOut(OutputWidth-1)                      <= '0';
266                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
267                             when 14 =>  DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
268                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(2,OutputWidthLength));
269                             when 15 =>  DataOut(OutputWidth-1)                      <= '0';
270                                     DataLength <=
                                         ↪   std_logic_vector(to_unsigned(1,OutputWidthLength));
271                         end case;
272                         exit write_last_sig_coeff_y_prefix_loop;
273                     end if;
274                 end loop;
275             when write_sig_coeff_flag =>
276                 BinarizeState <= write_coeff_abs_level_greater1;
277                 write_sig_coeff_flag_loop : for index in 0 to 15 loop
278                     if sig_coeff_flag_found = '1' then
279                         DataOut(sig_coeff_flag_index) <= sig_coeff_flag(index);
280                         sig_coeff_flag_index := sig_coeff_flag_index - 1;
281                         if(sig_coeff_flag(index) = '1') then
282                             coefficients(index) <= coefficients(index) - 1;
283                         end if;
284                     elsif(sig_coeff_flag(index) = '1') then
285                         sig_coeff_flag_found := '1';
286                         DataLength <= std_logic_vector(to_unsigned(15-index,OutputWidthLength));
287                         coefficients(index) <= coefficients(index) - 1;
288                     end if;
289                 end loop;
290
291             when write_coeff_abs_level_greater1 =>
292                 BinarizeState <= write_coeff_abs_level_greater2;
293                 write_coeff_abs_level_greater_1_loop : for index in 0 to 15 loop
294                     if sig_coeff_flag(index) = '1' and ALG1_index < 8 then
295                         if( coeff_abs_level_greater1_flag(index) = '1') then
296                             DataOut(OutputWidth-1-ALG1_index) <= '1';
297                             report "ALG1:coefficients(" & integer'image(index) & "): " &
                                 ↪   integer'image(coefficients(index));
298                             coefficients(index) <= coefficients(index) - 1;
299                             report "ALG1:coefficients(" & integer'image(index) & "): " &
                                 ↪   integer'image(coefficients(index));
300                         else
301                             DataOut(OutputWidth-1-ALG1_index) <= '0';
302                         end if;
303                         ALG1_index := ALG1_index + 1;
304                     end if;
305                 end loop;
306
307                 DataLength <= std_logic_vector(to_unsigned(ALG1_index,OutputWidthLength));
308
309             when write_coeff_abs_level_greater2 =>
310                 BinarizeState <= sync;
311                 write_coeff_abs_level_greater_2_loop : for index in 0 to 15 loop
312                     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) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-130,7));
380                             DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
381                             PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
382                         when 258 to 513 =>
383                             DataOut(OutputWidth-1 downto OutputWidth-8) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-258,8));
384                             DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
385                             PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
386                         when 514 to 1025 =>
387                             DataOut(OutputWidth-1 downto OutputWidth-9) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-514,9));
388                             DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
389                             PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
390                         when 1026 to 2049 =>
391                             DataOut(OutputWidth-1 downto OutputWidth-10) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-1026,10));
392                             DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
393                             PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
394                         when 2050 to 4097 =>
395                             DataOut(OutputWidth-1 downto OutputWidth-11) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-2050,11));
396                             DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
397                             PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
398                         when 4098 to 8193 =>
399                             DataOut(OutputWidth-1 downto OutputWidth-12) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4098,12));
400                             DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
401                             PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
402                         when 8194 to 16385 =>
403                             DataOut(OutputWidth-1 downto OutputWidth-13) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8194,13));
404                             DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
405                             PrefixLength <= std_logic_vector(to_unsigned(17,OutputWidthLength));
406                         when 16386 to 32767 =>
407                             DataOut(OutputWidth-1 downto OutputWidth-14) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16386,14));
408                             DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
409                             PrefixLength <= std_logic_vector(to_unsigned(18,OutputWidthLength));
410                         when others     =>
411                     end case;
412                 when 1 =>
413                     case (ABS_level_writeout) is
414                         when 0 to 1 =>
415                             DataOut(OutputWidth-1 downto OutputWidth-1) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout,1));
416                             DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
417                             PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
418                         when 2 to 3 =>
419                             DataOut(OutputWidth-1 downto OutputWidth-1) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-2,1));
420                             DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
421                             PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
422                         when 4 to 5 =>
423                             DataOut(OutputWidth-1 downto OutputWidth-1) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4,1));
424                             DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
425                             PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
426                         when 6 to 7 =>
427                             DataOut(OutputWidth-1 downto OutputWidth-1) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-6,1));
428                             DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
429                             PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
430                         when 8 to 11 =>
431                             DataOut(OutputWidth-1 downto OutputWidth-2) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8,2));
432                             DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
433                             PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
434                         when 12 to 19 =>
435                             DataOut(OutputWidth-1 downto OutputWidth-3) <=
                                 ↪   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) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-20,4));
440                             DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
441                             PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
442                         when 36 to 67 =>
443                             DataOut(OutputWidth-1 downto OutputWidth-5) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-36,5));
444                             DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
445                             PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
446                         when 68 to 131 =>
447                             DataOut(OutputWidth-1 downto OutputWidth-6) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-68,6));
448                             DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
449                             PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
450                         when 132 to 259 =>
451                             DataOut(OutputWidth-1 downto OutputWidth-7) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-132,7));
452                             DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
453                             PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
454                         when 260 to 515 =>
455                             DataOut(OutputWidth-1 downto OutputWidth-8) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-260,8));
456                             DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
457                             PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
458                         when 516 to 1027 =>
459                             DataOut(OutputWidth-1 downto OutputWidth-9) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-516,9));
460                             DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
461                             PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
462                         when 1028 to 2051 =>
463                             DataOut(OutputWidth-1 downto OutputWidth-10) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-1028,10));
464                             DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
465                             PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
466                         when 2052 to 4099 =>
467                             DataOut(OutputWidth-1 downto OutputWidth-11) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-2052,11));
468                             DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
469                             PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
470                         when 4100 to 8195 =>
471                             DataOut(OutputWidth-1 downto OutputWidth-12) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4100,12));
472                             DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
473                             PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
474                         when 8196 to 16387 =>
475                             DataOut(OutputWidth-1 downto OutputWidth-13) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8196,13));
476                             DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
477                             PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
478                         when 16388 to 32767 =>
479                             DataOut(OutputWidth-1 downto OutputWidth-14) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16388,14));
480                             DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
481                             PrefixLength <= std_logic_vector(to_unsigned(17,OutputWidthLength));
482                         when others     =>
483                     end case;
484                 when 2 =>
485                     case (ABS_level_writeout) is
486                         when 0 to 3 =>
487                             DataOut(OutputWidth-1 downto OutputWidth-2) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout,2));
488                             DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
489                             PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
490                         when 4 to 7 =>
491                             DataOut(OutputWidth-1 downto OutputWidth-2) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4,2));
492                             DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
493                             PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
494                         when 8 to 11 =>
495                             DataOut(OutputWidth-1 downto OutputWidth-2) <=
                                 ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8,2));
```

```
                        DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
                    when 12 to 15 =>
                        DataOut(OutputWidth-1 downto OutputWidth-2) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-12,2));
                        DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
                    when 16 to 23 =>
                        DataOut(OutputWidth-1 downto OutputWidth-3) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16,3));
                        DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
                    when 24 to 39 =>
                        DataOut(OutputWidth-1 downto OutputWidth-4) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-24,4));
                        DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
                    when 40 to 71 =>
                        DataOut(OutputWidth-1 downto OutputWidth-5) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-40,5));
                        DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
                    when 72 to 135 =>
                        DataOut(OutputWidth-1 downto OutputWidth-6) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-72,6));
                        DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
                    when 136 to 263 =>
                        DataOut(OutputWidth-1 downto OutputWidth-7) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-136,7));
                        DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
                    when 264 to 519 =>
                        DataOut(OutputWidth-1 downto OutputWidth-8) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-264,8));
                        DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
                    when 520 to 1031 =>
                        DataOut(OutputWidth-1 downto OutputWidth-9) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-520,9));
                        DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
                    when 1032 to 2055 =>
                        DataOut(OutputWidth-1 downto OutputWidth-10) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-1032,10));
                        DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
                    when 2056 to 4103 =>
                        DataOut(OutputWidth-1 downto OutputWidth-11) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-2056,11));
                        DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
                    when 4104 to 8199 =>
                        DataOut(OutputWidth-1 downto OutputWidth-12) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4104,12));
                        DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
                    when 8200 to 16391 =>
                        DataOut(OutputWidth-1 downto OutputWidth-13) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8200,13));
                        DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
                    when 16392 to 32767 =>
                        DataOut(OutputWidth-1 downto OutputWidth-14) <=
                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16392,14));
                        DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
                        PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
                    when others     =>
                end case;
            when 3 =>
                case (ABS_level_writeout) is
                    when 0 to 7 =>
                        DataOut(OutputWidth-1 downto OutputWidth-3) <=
                         ↪   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) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8,3));
560                                 DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
561                                 PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
562                             when 16 to 23 =>
563                                 DataOut(OutputWidth-1 downto OutputWidth-3) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16,3));
564                                 DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
565                                 PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
566                             when 24 to 31 =>
567                                 DataOut(OutputWidth-1 downto OutputWidth-3) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-24,3));
568                                 DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
569                                 PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
570                             when 32 to 47 =>
571                                 DataOut(OutputWidth-1 downto OutputWidth-4) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-32,4));
572                                 DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
573                                 PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
574                             when 48 to 79 =>
575                                 DataOut(OutputWidth-1 downto OutputWidth-5) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-48,5));
576                                 DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
577                                 PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
578                             when 80 to 143 =>
579                                 DataOut(OutputWidth-1 downto OutputWidth-6) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-80,6));
580                                 DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
581                                 PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
582                             when 144 to 271 =>
583                                 DataOut(OutputWidth-1 downto OutputWidth-7) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-144,7));
584                                 DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
585                                 PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
586                             when 272 to 527 =>
587                                 DataOut(OutputWidth-1 downto OutputWidth-8) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-272,8));
588                                 DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
589                                 PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
590                             when 528 to 1039 =>
591                                 DataOut(OutputWidth-1 downto OutputWidth-9) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-528,9));
592                                 DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
593                                 PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
594                             when 1040 to 2063 =>
595                                 DataOut(OutputWidth-1 downto OutputWidth-10) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-1040,10));
596                                 DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
597                                 PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
598                             when 2064 to 4111 =>
599                                 DataOut(OutputWidth-1 downto OutputWidth-11) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-2064,11));
600                                 DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
601                                 PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
602                             when 4112 to 8207 =>
603                                 DataOut(OutputWidth-1 downto OutputWidth-12) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4112,12));
604                                 DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
605                                 PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
606                             when 8208 to 16399 =>
607                                 DataOut(OutputWidth-1 downto OutputWidth-13) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8208,13));
608                                 DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
609                                 PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
610                             when 16400 to 32767 =>
611                                 DataOut(OutputWidth-1 downto OutputWidth-14) <=
                                     ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16400,14));
612                                 DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
613                                 PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
614                             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) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout,4));
620                                     DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
621                                     PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
622                                 when 16 to 31 =>
623                                     DataOut(OutputWidth-1 downto OutputWidth-4) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16,4));
624                                     DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
625                                     PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
626                                 when 32 to 47 =>
627                                     DataOut(OutputWidth-1 downto OutputWidth-4) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-32,4));
628                                     DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
629                                     PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
630                                 when 48 to 63 =>
631                                     DataOut(OutputWidth-1 downto OutputWidth-4) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-48,4));
632                                     DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
633                                     PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
634                                 when 64 to 95 =>
635                                     DataOut(OutputWidth-1 downto OutputWidth-5) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-64,5));
636                                     DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
637                                     PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
638                                 when 96 to 159 =>
639                                     DataOut(OutputWidth-1 downto OutputWidth-6) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-96,6));
640                                     DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
641                                     PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
642                                 when 160 to 287 =>
643                                     DataOut(OutputWidth-1 downto OutputWidth-7) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-160,7));
644                                     DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
645                                     PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
646                                 when 288 to 543 =>
647                                     DataOut(OutputWidth-1 downto OutputWidth-8) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-288,8));
648                                     DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
649                                     PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
650                                 when 544 to 1055 =>
651                                     DataOut(OutputWidth-1 downto OutputWidth-9) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-544,9));
652                                     DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
653                                     PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
654                                 when 1056 to 2079 =>
655                                     DataOut(OutputWidth-1 downto OutputWidth-10) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-1056,10));
656                                     DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
657                                     PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
658                                 when 2080 to 4127 =>
659                                     DataOut(OutputWidth-1 downto OutputWidth-11) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-2080,11));
660                                     DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
661                                     PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
662                                 when 4128 to 8223 =>
663                                     DataOut(OutputWidth-1 downto OutputWidth-12) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-4128,12));
664                                     DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
665                                     PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
666                                 when 8224 to 16415 =>
667                                     DataOut(OutputWidth-1 downto OutputWidth-13) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-8224,13));
668                                     DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
669                                     PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
670                                 when 16416 to 32767 =>
671                                     DataOut(OutputWidth-1 downto OutputWidth-14) <=
                                         ↪   std_logic_vector(to_unsigned(ABS_level_writeout-16416,14));
672                                     DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
673                                     PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
```

```
                                when others     =>
                            end case;
                        end case;

                        if(k<4 and (ABS_level_writeout) > (3*(2**k))) then
                            k <= k+1;
                        end if;

                    when write_coeff_sign_flag =>
                        BinarizeState <= write_finished;
                        write_coeff_sign_flag_loop : for index in 0 to 15 loop
                            if sig_coeff_flag(index) = '1' then
                                if coeff_sign_flag(index) = '1' then
                                    DataOut(OutputWidth-1 - SIGN_index) <= '1';
                                else
                                    DataOut(OutputWidth-1 - SIGN_index) <= '0';
                                end if;
                                SIGN_index := SIGN_index + 1;
                            end if;
                            DataLength <= std_logic_vector(to_unsigned(SIGN_index,OutputWidthLength));
                        end loop;
                    when others =>
                        BinarizeState <= write_finished;
                        Finished <= '1';
                        DataOut        <= (others => '0');
                        DataLength     <= (others => '0');
                end case;
            end if;
        end process;
    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     BypassO   : out  std_logic;
46     TermI     : in  std_logic;
47     TermO     : 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 InittransIdx (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 InitrangeTabLps (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 inital 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             TermO                        <= '0';
243             BypassO                      <= '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                         BypassO     <= 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
```

```
                    ivlLow := ivlLow(9 downto 0) & "0";
                    if (bins(binValI) /= '0') then
                        ivlLow := ivlLow + ivlCurrRange;
                    end if;
                    if (ivlLow>=1024) then
                        PutBitVal       := '1';
                        CABAC_EncState <= PutBit;
                        ivlLow       := ivlLow - 1024;
                    elsif (ivlLow<512) then
                        PutBitVal    := '0';
                        CABAC_EncState <= PutBit;
                    else
                        ivlLow          := ivlLow - 512;
                        bitsOutstanding := bitsOutstanding + 1;
                    end if;
                    binValI := binValI - 1;
                else
                    OutputLen    <= std_logic_vector(to_unsigned(PutBitI,OutputWLen));
                    CABAC_EncState <= w_ctx;
                end if;
            when enc_bin_r =>
                if (binValI>=(InputW-conv_integer(binsLen))) then
                    qRangeIdx    := ivlCurrRange(7 downto 6);
                    ivlLpsRange  := rangeTabLPS(conv_integer(currCtx(5 downto
                     ↪   0)))(conv_integer(qRangeIdx));
                    ivlCurrRange := ivlCurrRange - ivlLpsRange;
                    if(bins(binValI) /= currCtx(6)) then
                        ivlLow       := ivlLow + ivlCurrRange;
                        ivlCurrRange := "0" & ivlLpsRange;
                        if(currCtx(5 downto 0) = "000000") then
                            currCtx(6) <= not currCtx(6);
                        end if;
                        currCtx(5 downto 0) <= transIdxLPS(conv_integer(currCtx(5 downto 0)));
                    else
                        currCtx(5 downto 0) <= transIdxMPS(conv_integer(currCtx(5 downto 0)));
                    end if;
                    binValI  := binValI - 1;
                    CABAC_EncState <= RenormE;
                else
                    OutputLen <= std_logic_vector(to_unsigned(PutBitI,OutputWLen));
                    CABAC_EncState <= w_ctx;
                end if;
            when RenormE =>
                if (ivlCurrRange < 256) then
                    if (ivlLow < 256) then
                        PutBitVal       := '0';
                        CABAC_EncState <= PutBit;
                    elsif(ivlLow >= 512) then
                        ivlLow       := ivlLow - 512;
                        PutBitVal    := '1';
                        CABAC_EncState <= PutBit;
                    else
                        ivlLow := ivlLow - 256;
                        bitsOutstanding := bitsOutstanding + 1;
                        ivlCurrRange    := ivlCurrRange(7 downto 0) & "0";
                        ivlLow          := ivlLow(9 downto 0) & "0";
                    end if;
                else
                    if (Flushed = "11") then
                        OutputLen    <= std_logic_vector(to_unsigned(PutBitI,OutputWLen));
                        CABAC_EncState <= w_finished;
                    elsif (Flushed = "01") then
                        Flushed      := "10";
                        PutBitVal    := ivlLow(9);
                        CABAC_EncState <= PutBit;
                    else
                        CABAC_EncState <= enc_bin_r;
                    end if;
                end if;
            when PutBit =>
                if (firstBitFlag /= '0') then
                    firstBitFlag := '0';
                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                     TermO        <= '1';
404             end case;
405         end if;
406     end process;
407 end struct;
```

Appendix C

```
// ********************************************************************************
// Asynchronous fifo
// 08.05.17
// Norwegian University of Science and Technology
// Lars Erik Songe Paulsen
// ********************************************************************************

// ********************************************************************************
// TODO LIST:
// ********************************************************************************
// No "almost full" or "almost empty" signaling logic implemented
// ********************************************************************************
`timescale 1ns/1ps

module fifo #(parameter
                BUFFER_SIZE = 16,
                DATA_WIDTH = 32,
                ADDRESS_WIDTH = clogb2(BUFFER_SIZE) - 1
              )
    (
    // ------------------------------------------------------------------------------
    // Data in interface
    // ------------------------------------------------------------------------------
    input  wire rst_in_n,
    input  wire clock_in,
    input  wire [DATA_WIDTH-1:0] data_in,
    input  wire data_in_valid,
    output reg  data_in_full,

    // ------------------------------------------------------------------------------
    // Data out interface
    // ------------------------------------------------------------------------------
    input  wire rst_out_n,
    input  wire clock_out,
    output wire [DATA_WIDTH-1:0] data_out,
    output reg  data_out_valid,
    input  wire data_out_ack
    );

    // ------------------------------------------------------------------------------
    // Functions
    // ------------------------------------------------------------------------------
    // ceil log_2
    function integer clogb2;
        input integer depth;
              for (clogb2=0; depth>0; clogb2=clogb2+1)
                   depth = depth >> 1;
    endfunction

    // ------------------------------------------------------------------------------
    // Memory interface and logic
    // Low latency version(no output register)
    // See XilinxSimpleDualPort1ClockBlockRamExample.v for detailed documentation
    // ------------------------------------------------------------------------------
    reg [DATA_WIDTH-1:0] Buffer[BUFFER_SIZE-1:0];

    // Initialize memory values to all zeros
    generate
        integer ram_index;
        initial
            for(ram_index = 0; ram_index < BUFFER_SIZE; ram_index = ram_index + 1)
                Buffer[ram_index] = {DATA_WIDTH{1'b0}};
    endgenerate



    // Conditional sampling of data_in
    always @(posedge clock_in) begin
        if (data_in_valid && !data_in_full)
            Buffer[BufferWriteAddress] <= data_in;
    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 actuall 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 syncronizing across clock domains
90      wire [ADDRESS_WIDTH:0]   WriteGrayNextPointer, ReadGrayNextPointer;
91
92      // Gray coded pointers for synchronizing accross 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 adress 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
```

```
`timescale 1ns/1ps

module fifo_tb;
parameter BUFFER_SIZE = 128;//128;
parameter DATA_WIDTH = 32;

    // Data in interface
    reg rst_in_n;
    reg clock_in;
    reg [DATA_WIDTH-1:0] data_in;
    reg data_in_valid;
    wire data_in_full;

    // Data out interface
    reg rst_out_n;
    reg clock_out;
    wire [DATA_WIDTH-1:0] data_out;
    wire data_out_valid;
    reg data_out_ack;

fifo dut(.rst_in_n(rst_in_n),
         .clock_in(clock_in),
         .data_in(data_in),
         .data_in_valid(data_in_valid),
         .data_in_full(data_in_full),
         .rst_out_n(rst_out_n),
         .clock_out(clock_out),
         .data_out(data_out),
         .data_out_valid(data_out_valid),
         .data_out_ack(data_out_ack)
        );

reg [31:0] COMPARE[0:65536];
reg [7:0] in_index, out_index;

initial begin // data in
    in_index = 8'b00000000;
    clock_in = 1'b0;
    rst_in_n = 1'b0;
    data_in_valid = 1'b1;
    data_in = 32'h00000001;
    #5;
    clock_in = 1'b1;
    rst_in_n = 1'b1;
    //data_in_valid = 1'b1;
    #5;
    repeat (20) begin
        if(!data_in_full) begin
            if(clock_in) begin // change data on negedge
                if(data_in != 0) begin
                    data_in = (data_in) + 1;
                end
                else begin
                    data_in <= 1'b1;
                end
            end
            else begin
            end
        end
        clock_in = ~clock_in;
        #50;
    end




    repeat (200) begin
        if(!data_in_full) begin
            if(clock_in) begin // change data on negedge
                if(data_in != 0) begin
                    data_in = (data_in) + 1;
                end
                else begin
```

```
                    data_in <= 1'b1;
                end
            end
            else begin
            end
        end
        clock_in = ~clock_in;
        #2;
    end

    repeat (200) begin
        if(!data_in_full) begin
            if(clock_in) begin // change data on negedge
                if(data_in != 0) begin
                    data_in = (data_in) + 1;
                end
                else begin
                    data_in <= 1'b1;
                end
            end
            else begin
            end
        end
        clock_in = ~clock_in;
        #50;
    end
end

// Sample data to compare with output
always @(posedge clock_in or negedge rst_in_n) begin
    if (!rst_in_n) begin

    end
    else if (data_in_valid && !data_in_full) begin
        COMPARE[in_index] <= (data_in);
        in_index <= in_index + 1;
    end
end

initial begin // data out
    out_index = 8'b00000000;
    clock_out = 1'b0;
    rst_out_n = 1'b0;
    data_out_ack = 1'b0;
    #5;
    rst_out_n = 1'b1;
    data_out_ack = 1'b1;
    #220;
    clock_out = 1'b1;









    repeat (200) begin
        #1;
        if (data_out_valid) begin
            if(clock_out) begin
                if (data_out == COMPARE[out_index]) begin
                    $monitor("data match");
                end
                else begin
                    $monitor("data missmatch: out_index: %d: %d != %d",out_index,
                    COMPARE[out_index], data_out);
                end
            end
            else begin
                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 missmatch: 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 missmatch: 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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace HEVC_CABAC_Verification_Tool
{
    class CABAC_encoder
    {
        private List<Syntax_element> S_E_ = new List<Syntax_element>();
        private int S_E_index;
        private List<bin> Encoded;

        public uint offset;

        static uint qCodIRangeidx, CodIrangeLPS, codIRange, codIOffset, codILow;
        static uint bitsOutstanding;
        public bool firstBitFlag;


        static uint ctxIdxTable_depth = 120;

        public uint[,] rangeTabLPS = new uint[64, 4];
        public uint[] transIdxMPS = new uint[64];
        public uint[] transIdxLPS = new uint[64];
        public uint[] pStateIdxTable = new uint[ctxIdxTable_depth * 3 * 52];
        public uint[] MPSIdxTable = new uint[ctxIdxTable_depth * 3 * 52];

        public List<bin> Encode(List<Syntax_element> S_E)
        {
            S_E_ = S_E;
            Encoded = new List<bin>();
            bitsOutstanding = 0;
            S_E_index = 0;

            uint bin;
            bool bypass;

            ResetCodeVals();

            try
            {
                while (S_E_index < S_E.Count)
                {
                    bin = read_bit(out bypass);
                    if (bypass)
                    {
                        EncodeBypass(bin);
                    }
                    else
                    {
                        EncodeDecision(bin);
                    }
                }
                EncodeTerminate(1);

                return Encoded;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }

            return new List<bin>();
        }
```

```
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
                ↪   codIOffset.
108             //In the initialization procedure of the arithmetic decoding process,
109             //codIRange is set equal to 510 and codIOffset is set equal to the value returned from
                ↪   read_bits(9)
110             //interpreted as a 9 bit binary representation of an unsigned integer with most significant bit
                ↪   written first.
111             codIRange = 510;
112             codILow = 0;
113             qCodIRangeidx = 0;
114             CodIrangeLPS = 0;
115             codIOffset = 0;
116             codILow = 0;
117         }
118
119
120         public void EncodeDecision(uint bin)
121         {
122             qCodIRangeidx = (codIRange >> 6) & 3;
123             CodIrangeLPS = rangeTabLPS[pStateIdxTable[offset], qCodIRangeidx];
124             codIRange = codIRange - CodIrangeLPS;
125             if (bin != MPSIdxTable[offset])
126             {
127                 codILow = codILow + codIRange;
128                 codIRange = CodIrangeLPS;
129                 if (pStateIdxTable[offset] == 0)
130                 {
131                     MPSIdxTable[offset] = 1 - MPSIdxTable[offset];
132                 }
133                 pStateIdxTable[offset] = transIdxLPS[pStateIdxTable[offset]];
134             }
135             else
136             {
137                 pStateIdxTable[offset] = transIdxMPS[pStateIdxTable[offset]];
138             }
139             RenormE();
140         }
141
```

```
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() +
                 ↪   "\n Only 0 or 1 is valid arguments"); }
200 
201             if (firstBitFlag)
202             {
203                 firstBitFlag = false;
204             }
205             else
206             {
207                 bin tempBin = new bin();
208                 tempBin.val = (B == 1) ? '1' : '0';
209                 Encoded.Add(tempBin);
210             }
211             while (bitsOutstanding > 0)
212             {
213                 bin tempBin = new bin();
```

```
                tempBin.val = (B == 1) ? '0' : '1';
                Encoded.Add(tempBin);
                bitsOutstanding--;
            }
        }

        public void EncodeTerminate(uint bin)
        {
            try
            {
                codIRange = codIRange - 2;
                if (bin != 0)
                {
                    codILow = codILow + codIRange;
                    EncodeFlush();
                }
                else
                {
                    RenormE();
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }

        public void EncodeFlush()
        {
            try
            {
                codIRange = 2;
                RenormE();
                PutBit((codILow >> 9) & 1);
                //PutBit(((codILow >> 7) & 3) | 1); // does not work. Using a workarround for now.
                PutBit((codILow >> 8) & 1);
                PutBit(((codILow >> 7) & 1) | 1);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }

    }

    class Syntax_element
    {
        public List<char> Bins = new List<char>();
        public int ctxIdx;
        public int currPos = 0;
        public int nr = 0;
        public bool bypass;
    }

    class bin
    {
        public char val;
    }
}
```

Appendix E

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace HEVC_CABAC_Verification_Tool
{
    class CABAC_decoder
    {
        private List<bin> bins_ = new List<bin>();
        private int bins_index;
        private int syntax_element_index;


        private char[] Decoded;
        private int Decodedindex;

        public RichTextBox binValTarget;
        public RichTextBox debugg;
        uint binValIndex;

        public uint offset;

        static uint qCodIRangeidx, CodIrangeLPS, codIRange, codIOffset, codILow;
        static uint bitsOutstanding;
        static bool firstBitFlag;


        static uint ctxIdxTable_depth = 120;

        public uint[,] rangeTabLPS = new uint[64, 4];
        public uint[] transIdxMPS = new uint[64];
        public uint[] transIdxLPS = new uint[64];
        public uint[] pStateIdxTable = new uint[ctxIdxTable_depth * 3 * 52];
        public uint[] MPSIdxTable = new uint[ctxIdxTable_depth * 3 * 52];


        private void ResetCodeVals()
        {
            try
            {
                //The status of the arithmetic decoding engine is represented by the variables codIRange and
                 ↪   codIOffset.
                //In the initialization procedure of the arithmetic decoding process,
                //codIRange is set equal to 510 and codIOffset is set equal to the value returned from
                 ↪   read_bits(9)
                //interpreted as a 9 bit binary representation of an unsigned integer with most significant
                 ↪   bit written first.
                codIRange = 510;
                codILow = 0;
                qCodIRangeidx = 0;
                CodIrangeLPS = 0;
                codIOffset = 0;
                codILow = 0;

                char[] tempA = "XXXXXXXXX".ToCharArray();
                for (int i = 0; i < 9; i++)
                {
                    tempA[i] = (read_bit() == 1) ? '1' : '0';
                }
                codIOffset = Convert.ToUInt32(new string(tempA), 2); // read 9 first binary vals
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }

        }

```

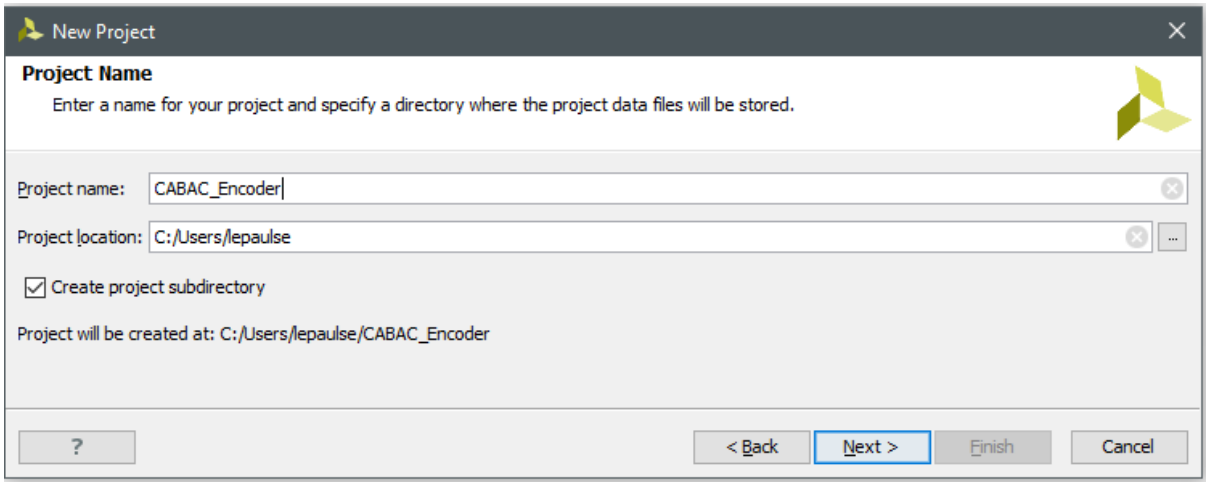
```
69          public string Decode(List<bin> bins)
70          {
71              Decoded = new char[65535];
72              Decodedindex = 0;
73
74              bins_ = bins;
75              bins_index = 0;
76              syntax_element_index = 0;
77
78              ResetCodeVals();
79
80              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
                         ↪  a value of " +
173                                     "codIOffset being greater than or equal to codIRange upon completion of
                                         ↪  this process.");
174                 }
175             }
176             catch (Exception ex)
177             {
178                 MessageBox.Show(ex.ToString());
179             }
180
181         }
182     }
183 }
```

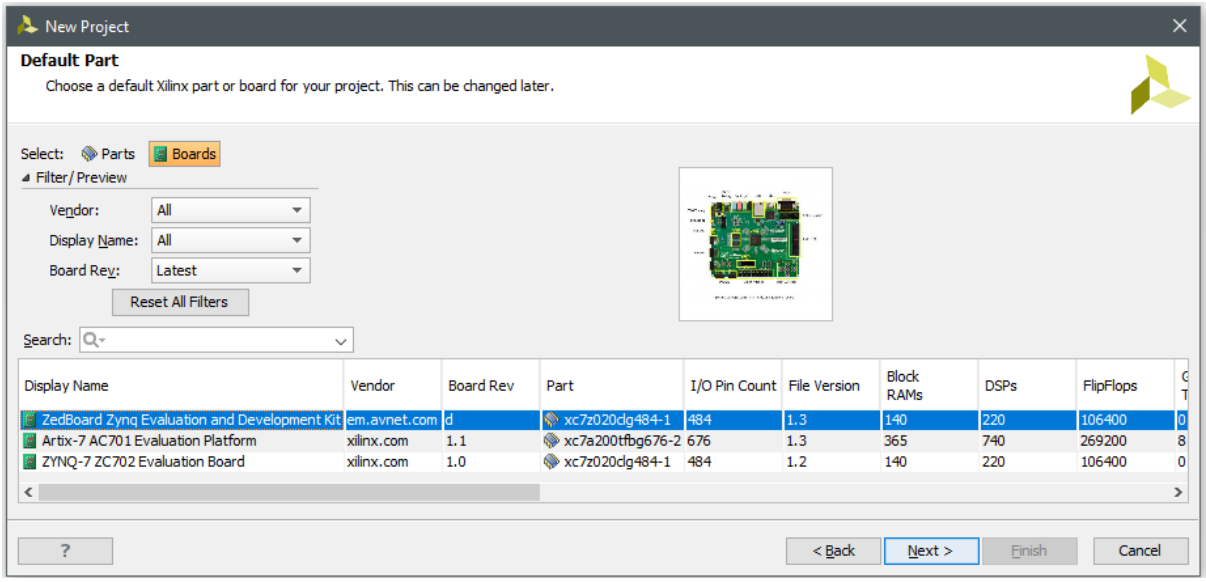
Appendix F

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

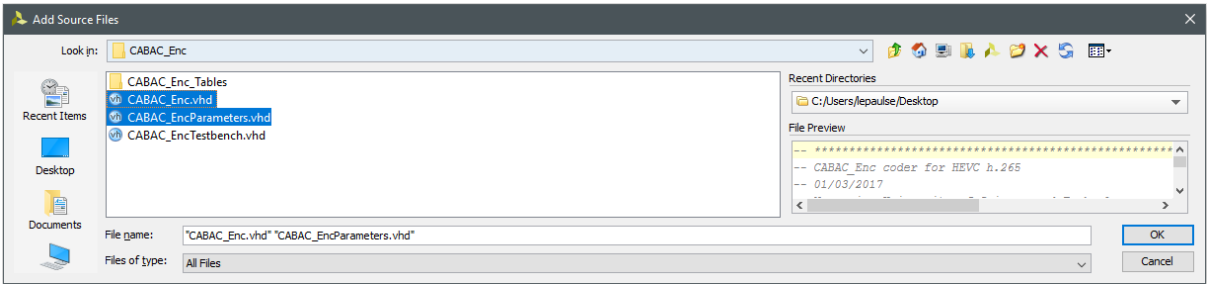


Create a new project in Vivado. Give it a suitable name and click next. Select "RTL Project" and "Do not specify sources at this time", and click next.

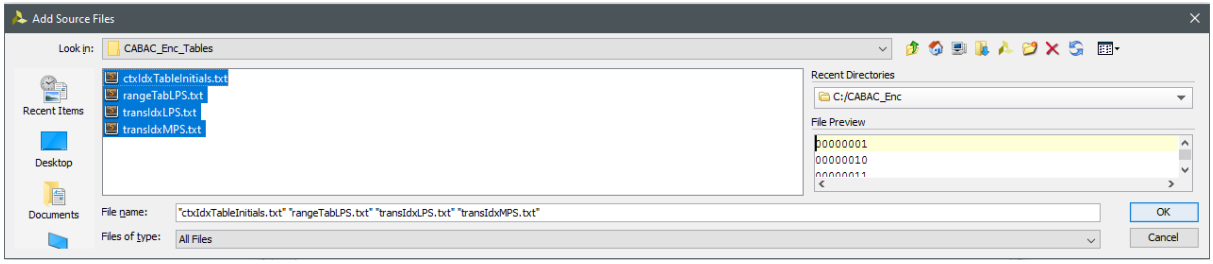


Select the Zedboard in the Boards Tab. Click next, then click finish.

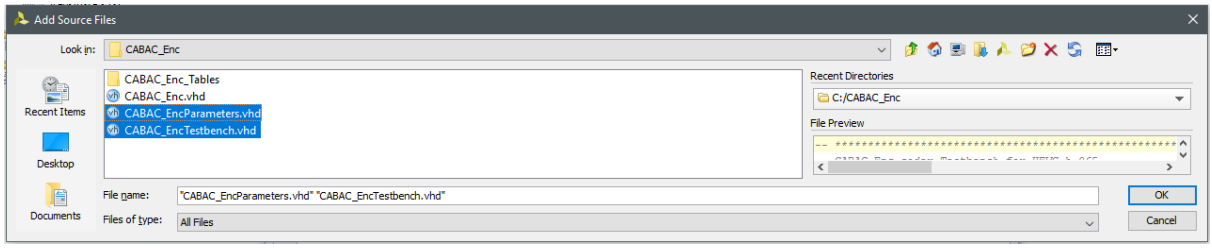
Navigate to the sources tab, and add the following files to design sources:



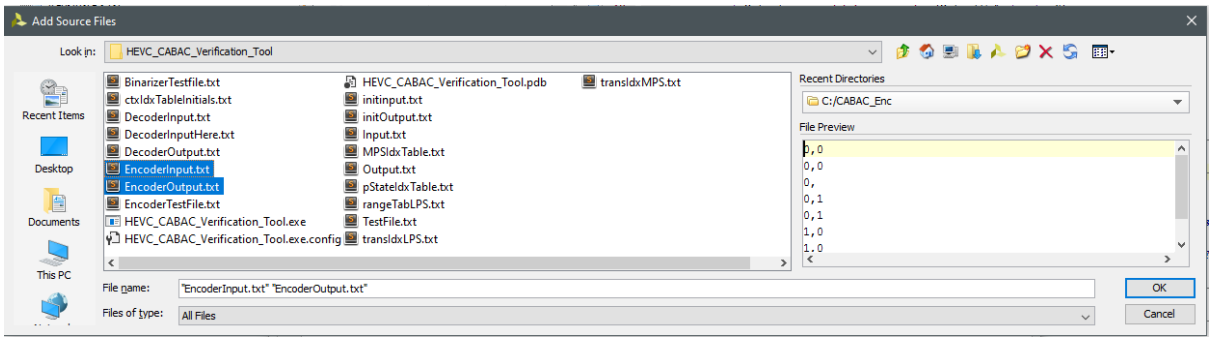
Then add the tables in the tables folder to design sources:



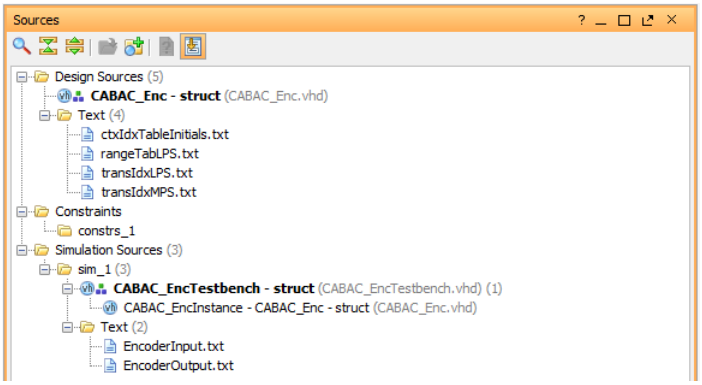
Then add the following files to simulation sources:



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



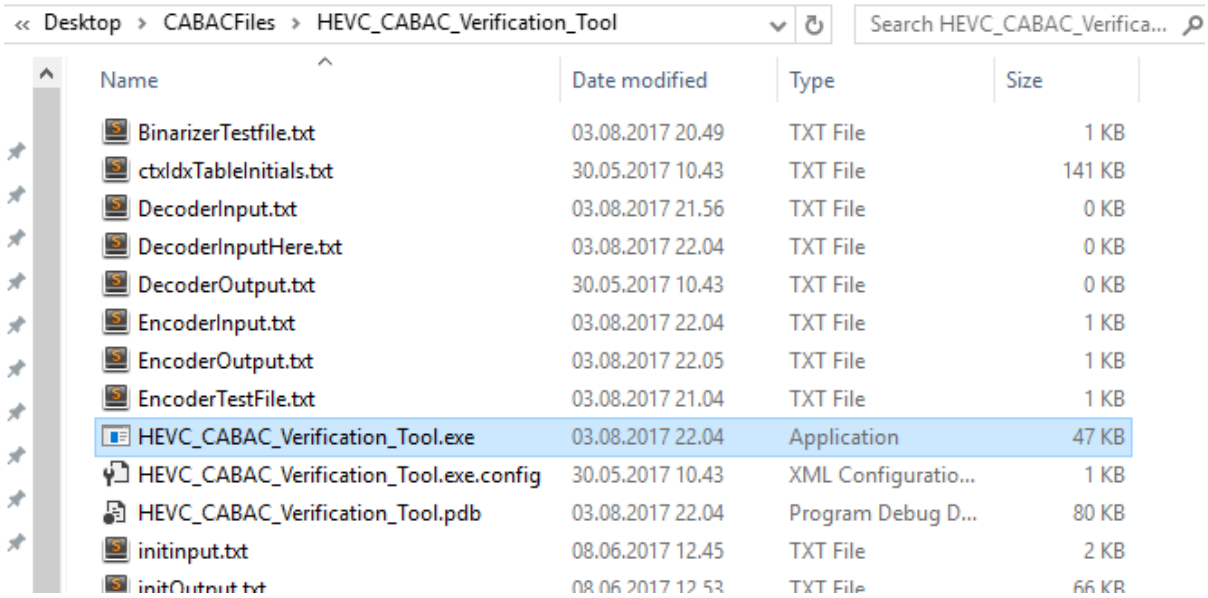
The Source tab should now look like this.



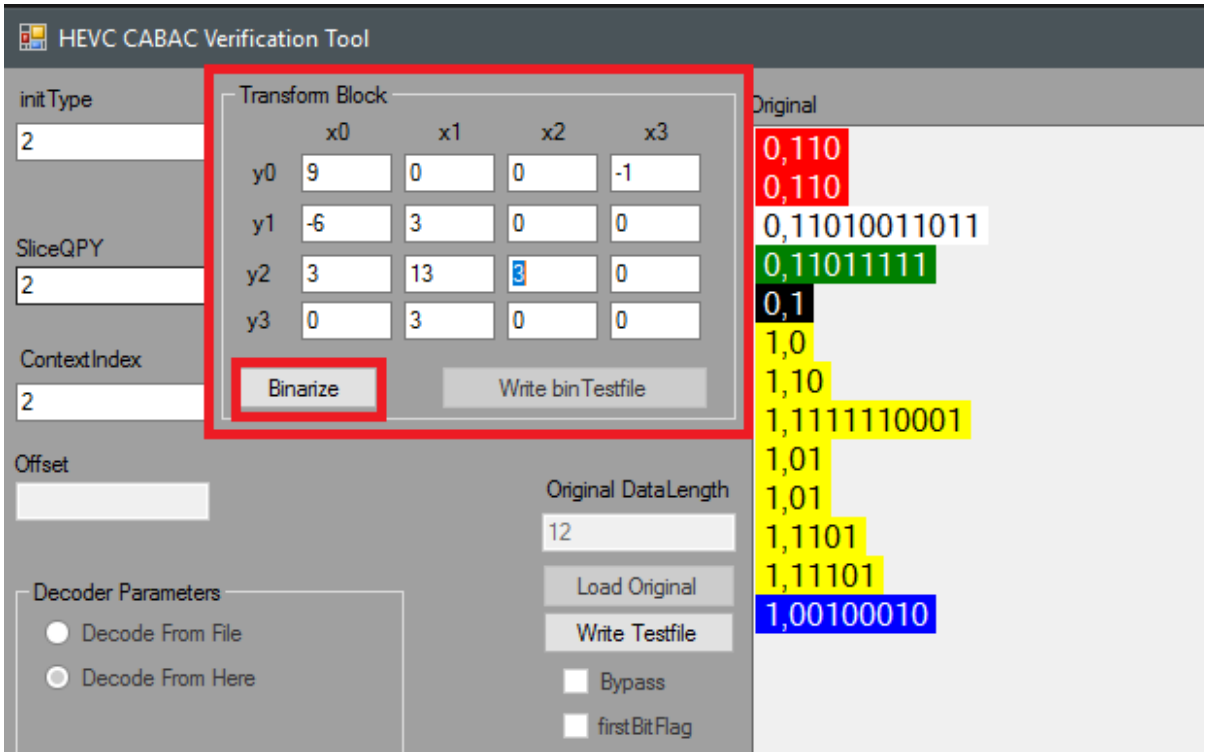
Open the "CABAC_EncTestbench.vhd" file and verify that the "EncoderInput.txt" and "EncoderOutput.txt" linked in the code is in the same folder as the "HEVC_CABAC_Verification_Tool.exe" executable file that is used. It should look something like this:

```
-- 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";
```

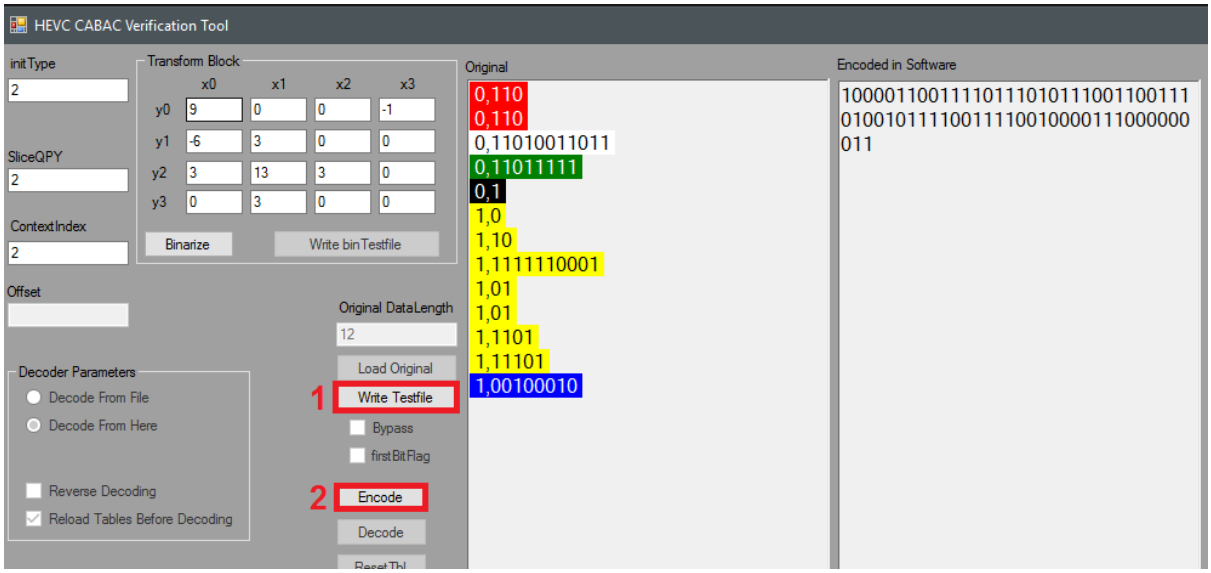
It is now possible to generate Encoder test data using the software model. First start the executable file.



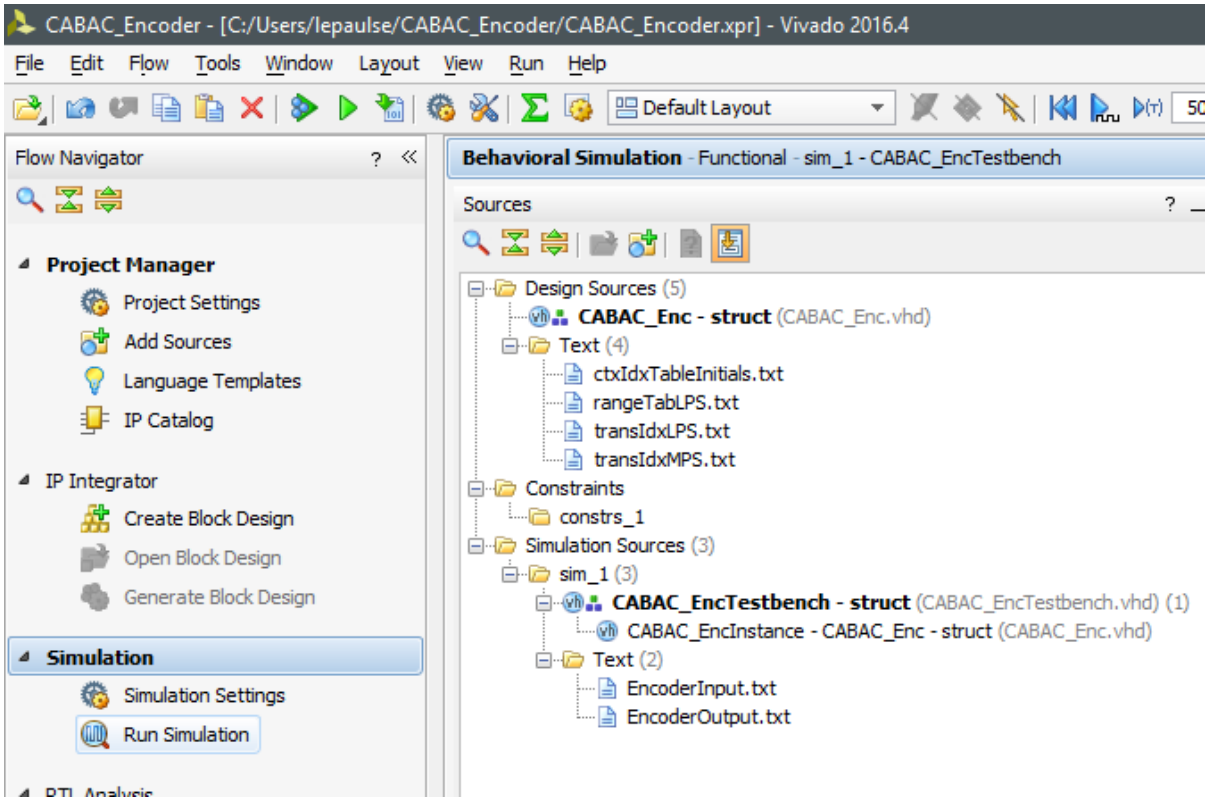
Create some binarized bins by changing the values in the transform block, or simply clicking Binarize.



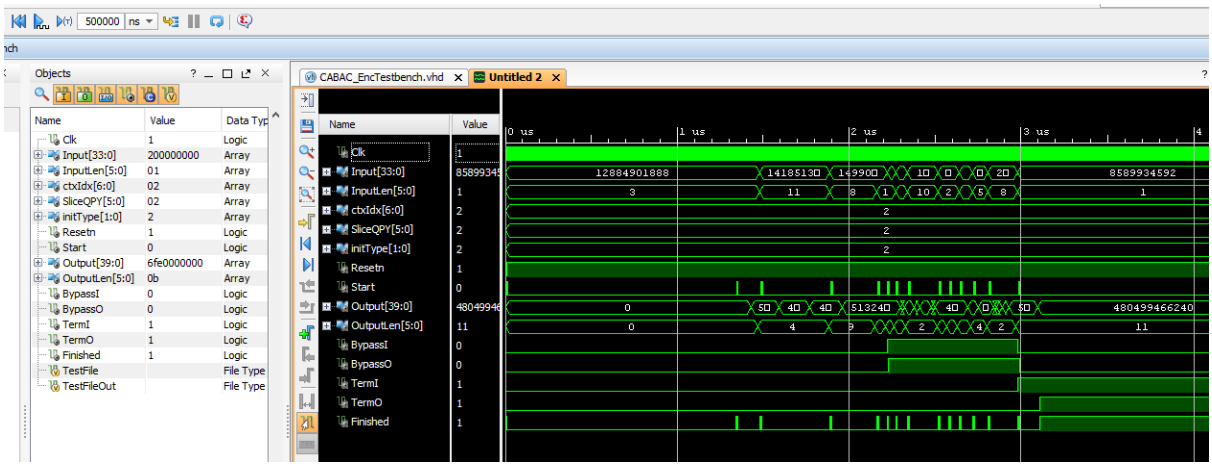
1: First write the current binarization the the EncoderInput.txt file by pressing the Write TestFile button. **2:** Then encode this data with the software encoder by pressing the Encode Butten.



Go back to back to Vivado and Press run simulation. This will perform CABAC hardware encoding using the same test file as written to earlier.

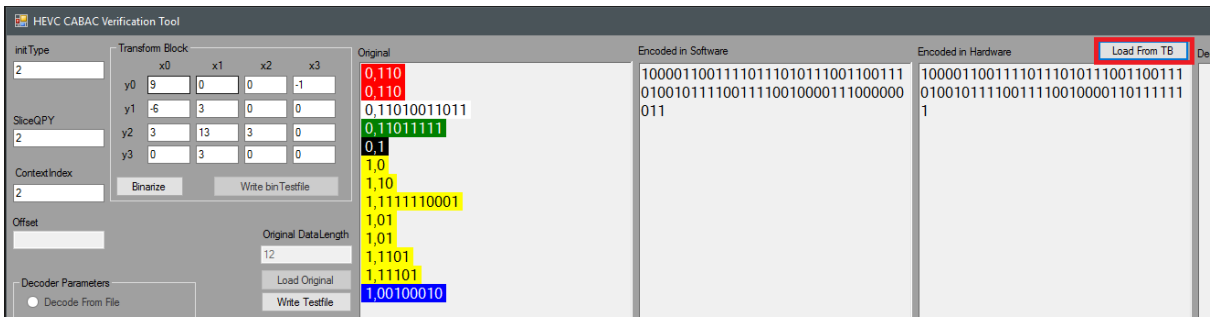


The Simulation should result in something like this:

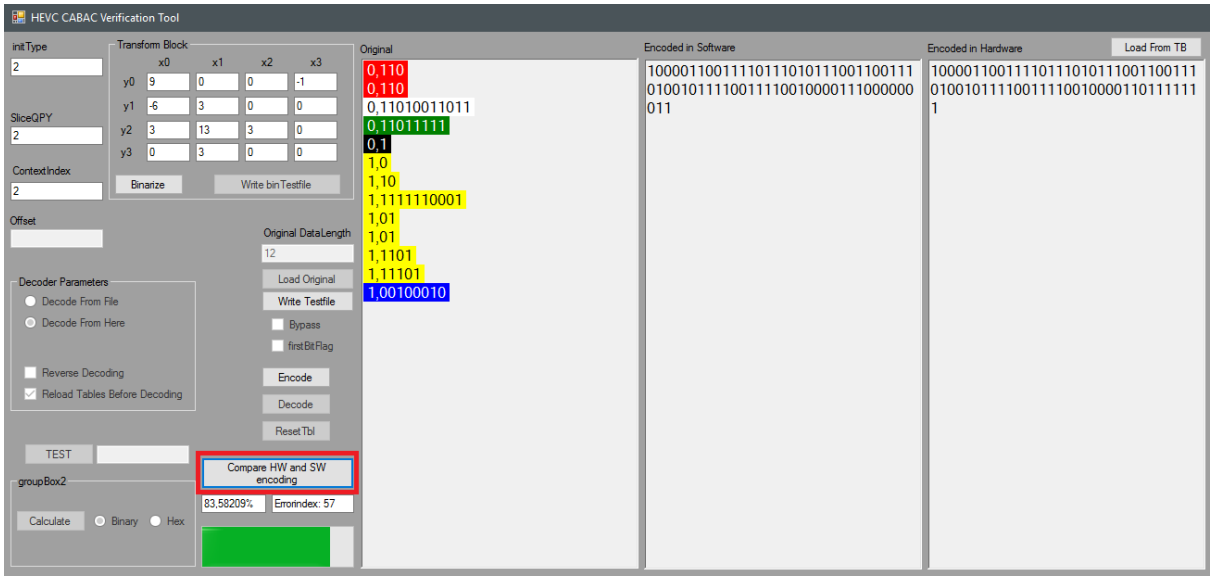


Make sure that the simulation is run long enough to finish encoding.

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

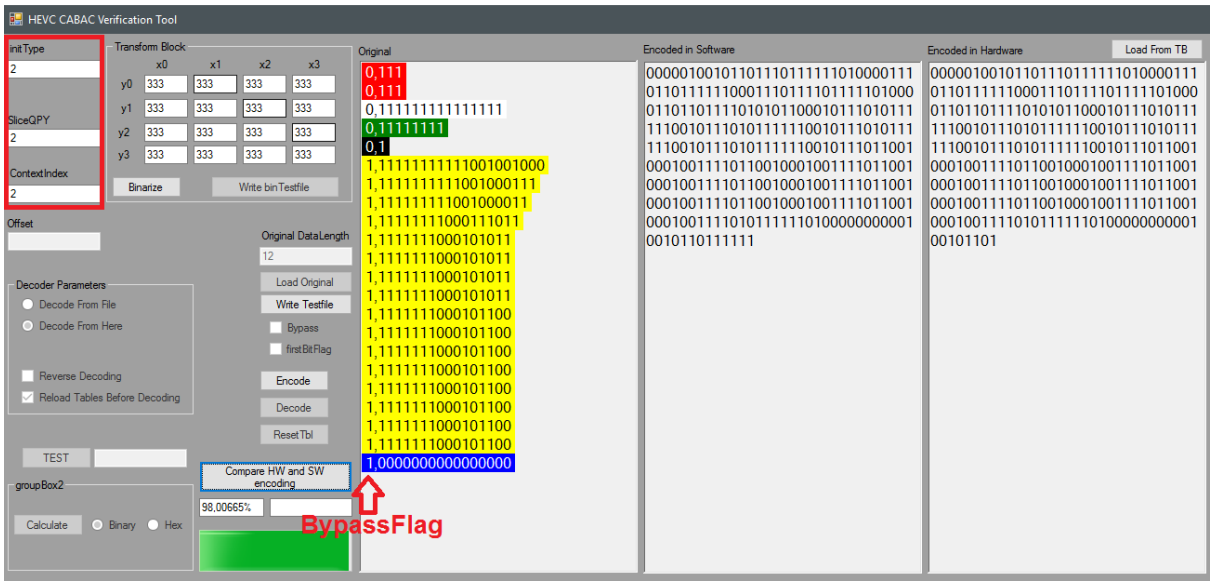


With both hardware and software encoder result showing, it is now possible to press the Compare HW and SW encoding button. This will show how many percent of the encoding is correct, as well as the index of the first conflicting character.

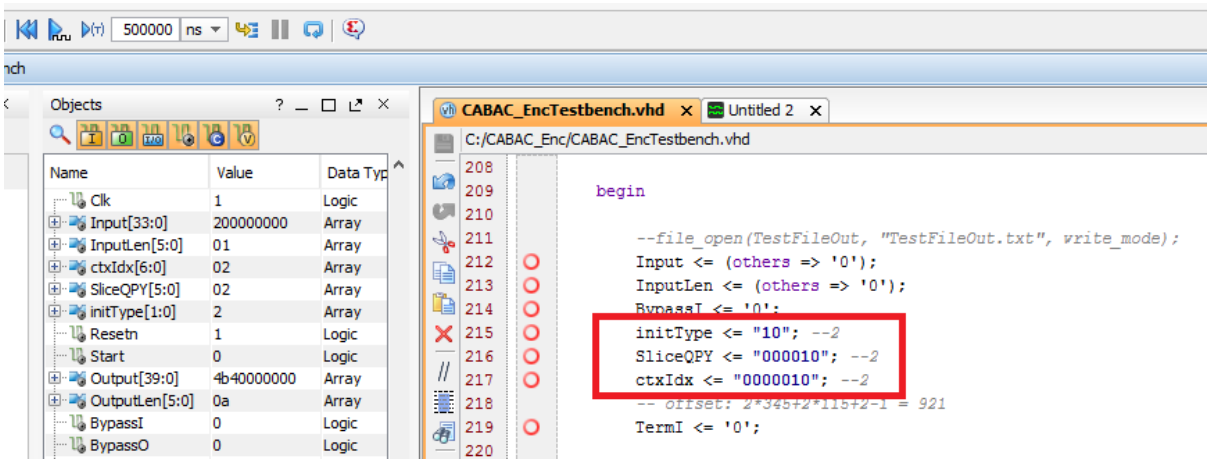


Note that some errors may occur either due to improper termination(as seen above) or if the BitsOutstanding Loop in the hardware encoder is completed without the BitsOutstanding register reaching a value of 0.

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



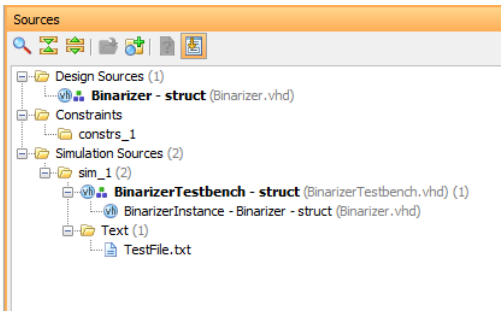
The encoders will use the context index signaled by initType, SliceQPY and ctxIdx. For the encoders to return the same result, these variables should be set to the same values, as shown above and below.



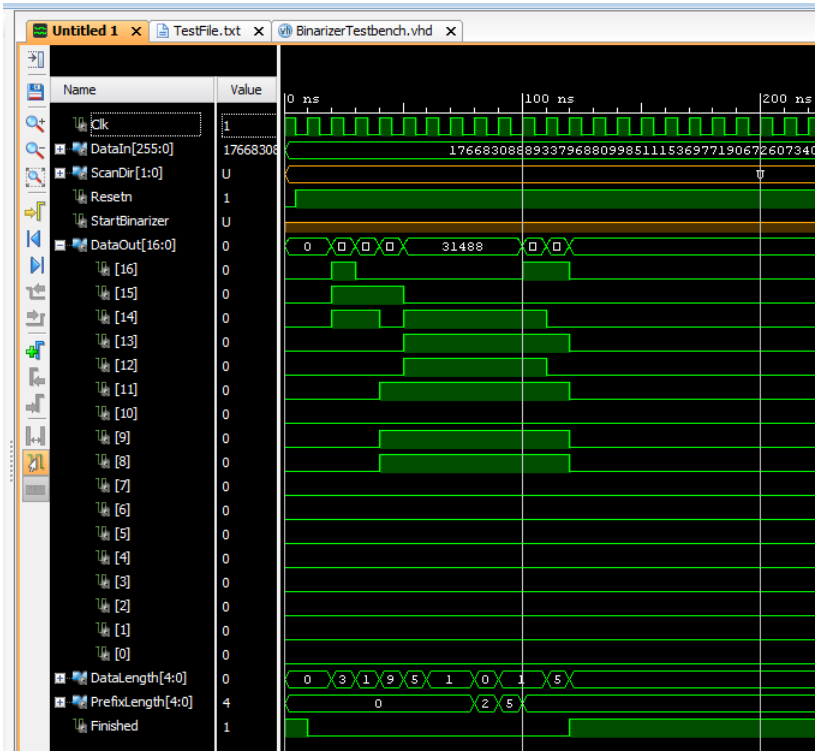
Note that it is possible to manually edit "EncoderInput.txt" to create a custom test pattern. But it is required to include the Bypassflag on each line.

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.



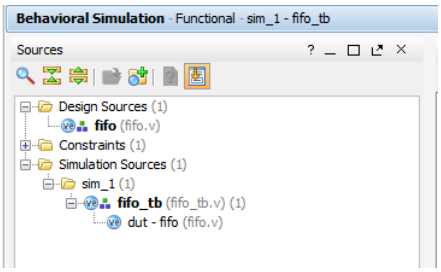
Press run simulation, and make sure it is able to run for long enough.



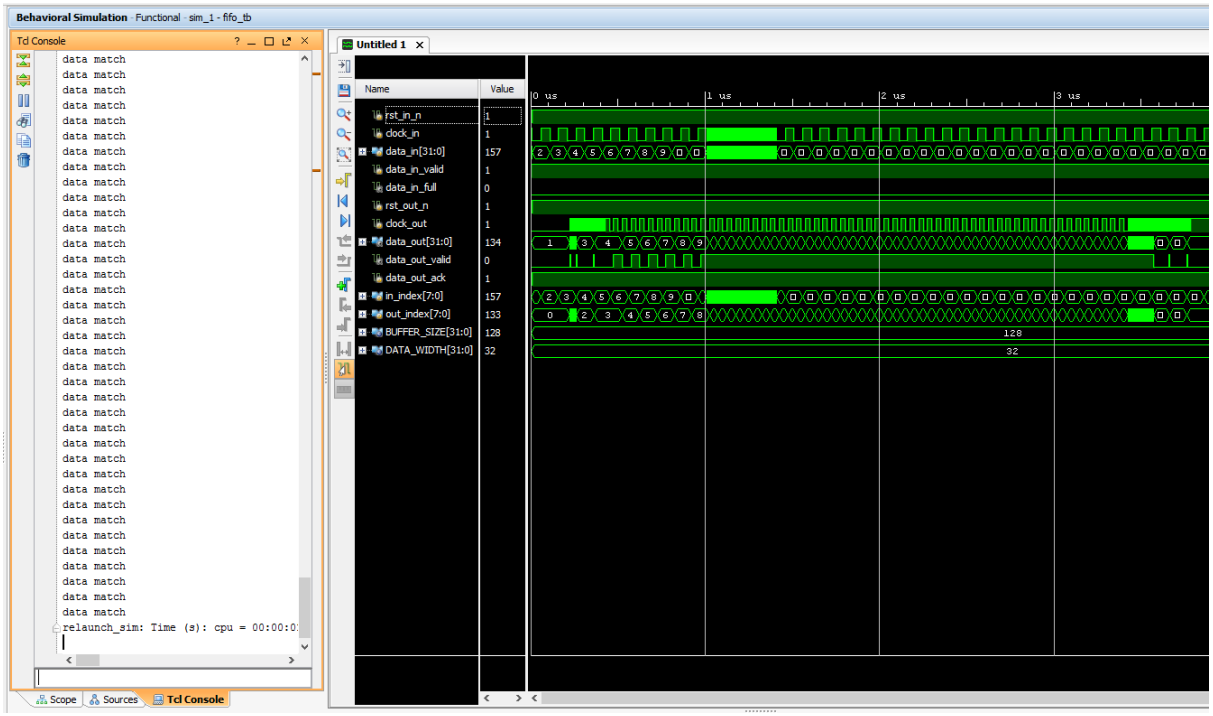
The binarizer will output the finished binarized elements in the order of LASTx, LASTy, SIG, ABS1, ABS2, ALRem and SIGN.

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.

```
// Binary initial value generation for HEVC CABAC context modeling.
void initTable_file_generation()
{
    // Input file containing initvalues copied from the HEVC standard document.
    // Should contain initvalues for initType 0, 1  and 2.
    // Each line should contain an initvalue on each line, starting with
    // every initvalue from initType 0, then imeadiatly followed by every initvalue
    // from initType 1, and finaly then every initvalue from initType2.
    StreamWriter input = initializeReadFile("initinput.txt");

    // Output file containing Binary(1/0 ASCII chars) formated initvalues for any value of sliceQPY and
     ↪   initType.
    StreamWriter Output = initializeWriteFile("initOutput.txt");

    Int32 initValue, slopeIdx, intersecIdx, m, n, preCtxState, valMPS, pStateIdx;

    try
    {
        for (int SliceQPY = 0; SliceQPY < 52; SliceQPY++)
        {

            while (!input.EndOfStream)
            {
                initValue = int.Parse(input.ReadLine());
                slopeIdx = initValue >> 4;
                intersecIdx = initValue & 15;
                m = (slopeIdx * 5) - 45;
                n = (intersecIdx << 3) - 16;
                preCtxState = Clip3(1, 126, ((m * Clip3(0, 51, SliceQPY)) >> 4) + n);
                valMPS = (preCtxState <= 63) ? 0 : 1;
                pStateIdx = (valMPS > 0) ? (preCtxState - 64) : (63 - preCtxState);

                // Binary(1/0 ASCII chars) formated output
                Output.WriteLine(String.Format("{0:X}", valMPS) +
                                 Convert.ToString(pStateIdx, 2).PadLeft(6, '0'));
            }
            input.DiscardBufferedData();
            input.BaseStream.Seek(0, System.IO.SeekOrigin.Begin);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
    Output.Close();
    input.Close();
}

int Clip3(int x, int y, int z)
{
    if (z < x) return x;
    else if (z > y) return y;
    else return z;
}
```

```
// Initialize write file to current folder.
StreamWriter initializeWriteFile(string filename)
{
    try
    {
        FileStream fs;
        StreamWriter file;
        string startUpPath;
        string currentLogFileName;
        string logfolder;
        currentLogFileName = string.Format(filename);
        startUpPath = Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().Location);
        logfolder = Path.Combine(startUpPath, currentLogFileName);
        File.Delete(logfolder);
        fs = File.Create(logfolder);
        file = new StreamWriter(fs);
        return file;
    }
    catch (IOException)
    {
        MessageBox.Show("Unable to access: " + filename);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
    return null;
}

// Initialize read file to current folder.
StreamReader initializeReadFile(string filename)
{
    try
    {
        FileStream fs;
        StreamReader file;
        string startUpPath;
        string currentLogFileName;
        string logfolder;
        currentLogFileName = string.Format(filename);
        startUpPath = Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().Location);
        logfolder = Path.Combine(startUpPath, currentLogFileName);
        fs = File.OpenRead(logfolder);
        file = new StreamReader(fs);
        return file;
    }
    catch (IOException)
    {
        MessageBox.Show("Unable to access: " + filename);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
    return null;
}
```

		initType								
		0			1			2		
syntax element	**ctxTable**	**ctxTable index**	**initValue**	**Index**	**ctxTable index**	**initValue**	**Index**	**ctxTable index**	**initValue**	**Index**
last_sig_coeff_x_prefix	9-26	0	110	0	18	125	120	36	125	240
last_sig_coeff_x_prefix	9-26	1	110	1	19	110	121	37	110	241
last_sig_coeff_x_prefix	9-26	2	124	2	20	94	122	38	124	242
last_sig_coeff_x_prefix	9-26	3	125	3	21	110	123	39	110	243
last_sig_coeff_x_prefix	9-26	4	140	4	22	95	124	40	95	244
last_sig_coeff_x_prefix	9-26	5	153	5	23	79	125	41	94	245
last_sig_coeff_x_prefix	9-26	6	125	6	24	125	126	42	125	246
last_sig_coeff_x_prefix	9-26	7	127	7	25	111	127	43	111	247
last_sig_coeff_x_prefix	9-26	8	140	8	26	110	128	44	111	248
last_sig_coeff_x_prefix	9-26	9	109	9	27	78	129	45	79	249
last_sig_coeff_x_prefix	9-26	10	111	10	28	110	130	46	125	250
last_sig_coeff_x_prefix	9-26	11	143	11	29	111	131	47	126	251
last_sig_coeff_x_prefix	9-26	12	127	12	30	111	132	48	111	252
last_sig_coeff_x_prefix	9-26	13	111	13	31	95	133	49	111	253
last_sig_coeff_x_prefix	9-26	14	79	14	32	94	134	50	79	254
last_sig_coeff_x_prefix	9-26	15	108	15	33	108	135	51	108	255
last_sig_coeff_x_prefix	9-26	16	123	16	34	123	136	52	123	256
last_sig_coeff_x_prefix	9-26	17	63	17	35	108	137	53	93	257
last_sig_coeff_y_prefix	9-27	0	110	18	18	125	138	36	125	258
last_sig_coeff_y_prefix	9-27	1	110	19	19	110	139	37	110	259
last_sig_coeff_y_prefix	9-27	2	124	20	20	94	140	38	124	260
last_sig_coeff_y_prefix	9-27	3	125	21	21	110	141	39	110	261
last_sig_coeff_y_prefix	9-27	4	140	22	22	95	142	40	95	262
last_sig_coeff_y_prefix	9-27	5	153	23	23	79	143	41	94	263
last_sig_coeff_y_prefix	9-27	6	125	24	24	125	144	42	125	264
last_sig_coeff_y_prefix	9-27	7	127	25	25	111	145	43	111	265
last_sig_coeff_y_prefix	9-27	8	140	26	26	110	146	44	111	266
last_sig_coeff_y_prefix	9-27	9	109	27	27	78	147	45	79	267
last_sig_coeff_y_prefix	9-27	10	111	28	28	110	148	46	125	268
last_sig_coeff_y_prefix	9-27	11	143	29	29	111	149	47	126	269
last_sig_coeff_y_prefix	9-27	12	127	30	30	111	150	48	111	270
last_sig_coeff_y_prefix	9-27	13	111	31	31	95	151	49	111	271
last_sig_coeff_y_prefix	9-27	14	79	32	32	94	152	50	79	272
last_sig_coeff_y_prefix	9-27	15	108	33	33	108	153	51	108	273
last_sig_coeff_y_prefix	9-27	16	123	34	34	123	154	52	123	274
last_sig_coeff_y_prefix	9-27	17	63	35	35	108	155	53	93	275
coded_sub_block_flag[ ][ ]	9-28	0	91	36	4	121	156	8	121	276
coded_sub_block_flag[ ][ ]	9-28	1	171	37	5	140	157	9	140	277
coded_sub_block_flag[ ][ ]	9-28	2	134	38	6	61	158	10	61	278
coded_sub_block_flag[ ][ ]	9-28	3	141	39	7	154	159	11	154	279
sig_coeff_flag[ ][ ]	9-29	0	111	40	42	155	160	84	170	280
sig_coeff_flag[ ][ ]	9-29	1	111	41	43	154	161	85	154	281
sig_coeff_flag[ ][ ]	9-29	2	125	42	44	139	162	86	139	282
sig_coeff_flag[ ][ ]	9-29	3	110	43	45	153	163	87	153	283
sig_coeff_flag[ ][ ]	9-29	4	110	44	46	139	164	88	139	284
sig_coeff_flag[ ][ ]	9-29	5	94	45	47	123	165	89	123	285
sig_coeff_flag[ ][ ]	9-29	6	124	46	48	123	166	90	123	286
sig_coeff_flag[ ][ ]	9-29	7	108	47	49	63	167	91	63	287
sig_coeff_flag[ ][ ]	9-29	8	124	48	50	153	168	92	124	288
sig_coeff_flag[ ][ ]	9-29	9	107	49	51	166	169	93	166	289
sig_coeff_flag[ ][ ]	9-29	10	125	50	52	183	170	94	183	290
sig_coeff_flag[ ][ ]	9-29	11	141	51	53	140	171	95	140	291
sig_coeff_flag[ ][ ]	9-29	12	179	52	54	136	172	96	136	292
sig_coeff_flag[ ][ ]	9-29	13	153	53	55	153	173	97	153	293
sig_coeff_flag[ ][ ]	9-29	14	125	54	56	154	174	98	154	294
sig_coeff_flag[ ][ ]	9-29	15	107	55	57	166	175	99	166	295
sig_coeff_flag[ ][ ]	9-29	16	125	56	58	183	176	100	183	296
sig_coeff_flag[ ][ ]	9-29	17	141	57	59	140	177	101	140	297
sig_coeff_flag[ ][ ]	9-29	18	179	58	60	136	178	102	136	298
sig_coeff_flag[ ][ ]	9-29	19	153	59	61	153	179	103	153	299
sig_coeff_flag[ ][ ]	9-29	20	125	60	62	154	180	104	154	300
sig_coeff_flag[ ][ ]	9-29	21	107	61	63	166	181	105	166	301
sig_coeff_flag[ ][ ]	9-29	22	125	62	64	183	182	106	183	302
sig_coeff_flag[ ][ ]	9-29	23	141	63	65	140	183	107	140	303
sig_coeff_flag[ ][ ]	9-29	24	179	64	66	136	184	108	136	304
sig_coeff_flag[ ][ ]	9-29	25	153	65	67	153	185	109	153	305
sig_coeff_flag[ ][ ]	9-29	26	125	66	68	154	186	110	154	306
sig_coeff_flag[ ][ ]	9-29	27	140	67	69	170	187	111	170	307
sig_coeff_flag[ ][ ]	9-29	28	139	68	70	153	188	112	153	308
sig_coeff_flag[ ][ ]	9-29	29	182	69	71	123	189	113	138	309
sig_coeff_flag[ ][ ]	9-29	30	182	70	72	123	190	114	138	310
sig_coeff_flag[ ][ ]	9-29	31	152	71	73	107	191	115	122	311
sig_coeff_flag[ ][ ]	9-29	32	136	72	74	121	192	116	121	312
sig_coeff_flag[ ][ ]	9-29	33	152	73	75	107	193	117	122	313
sig_coeff_flag[ ][ ]	9-29	34	136	74	76	121	194	118	121	314
sig_coeff_flag[ ][ ]	9-29	35	153	75	77	167	195	119	167	315
sig_coeff_flag[ ][ ]	9-29	36	136	76	78	151	196	120	151	316
sig_coeff_flag[ ][ ]	9-29	37	139	77	79	183	197	121	183	317
sig_coeff_flag[ ][ ]	9-29	38	111	78	80	140	198	122	140	318
sig_coeff_flag[ ][ ]	9-29	39	136	79	81	151	199	123	151	319
sig_coeff_flag[ ][ ]	9-29	40	139	80	82	183	200	124	183	320
sig_coeff_flag[ ][ ]	9-29	41	111	81	83	140	201	125	140	321
sig_coeff_flag[ ][ ]	9-29	126	141	82	129	140	202	130	140	322
sig_coeff_flag[ ][ ]	9-29	127	111	83	130	140	203	131	140	323
coeff_abs_level_greater1_flag[ ]	9-30	0	140	84	24	154	204	48	154	324
coeff_abs_level_greater1_flag[ ]	9-30	1	92	85	25	196	205	49	196	325
coeff_abs_level_greater1_flag[ ]	9-30	2	137	86	26	196	206	50	167	326
coeff_abs_level_greater1_flag[ ]	9-30	3	138	87	27	167	207	51	167	327
coeff_abs_level_greater1_flag[ ]	9-30	4	140	88	28	154	208	52	154	328
coeff_abs_level_greater1_flag[ ]	9-30	5	152	89	29	152	209	53	152	329
coeff_abs_level_greater1_flag[ ]	9-30	6	138	90	30	167	210	54	167	330
coeff_abs_level_greater1_flag[ ]	9-30	7	139	91	31	182	211	55	182	331
coeff_abs_level_greater1_flag[ ]	9-30	8	153	92	32	182	212	56	182	332
coeff_abs_level_greater1_flag[ ]	9-30	9	74	93	33	134	213	57	134	333
coeff_abs_level_greater1_flag[ ]	9-30	10	149	94	34	149	214	58	149	334
coeff_abs_level_greater1_flag[ ]	9-30	11	92	95	35	136	215	59	136	335
coeff_abs_level_greater1_flag[ ]	9-30	12	139	96	36	153	216	60	153	336
coeff_abs_level_greater1_flag[ ]	9-30	13	107	97	37	121	217	61	121	337
coeff_abs_level_greater1_flag[ ]	9-30	14	122	98	38	136	218	62	136	338
coeff_abs_level_greater1_flag[ ]	9-30	15	152	99	39	137	219	63	122	339
coeff_abs_level_greater1_flag[ ]	9-30	16	140	100	40	169	220	64	169	340
coeff_abs_level_greater1_flag[ ]	9-30	17	179	101	41	194	221	65	208	341
coeff_abs_level_greater1_flag[ ]	9-30	18	166	102	42	166	222	66	166	342
coeff_abs_level_greater1_flag[ ]	9-30	19	182	103	43	167	223	67	167	343
coeff_abs_level_greater1_flag[ ]	9-30	20	140	104	44	154	224	68	154	344
coeff_abs_level_greater1_flag[ ]	9-30	21	227	105	45	167	225	69	152	345
coeff_abs_level_greater1_flag[ ]	9-30	22	122	106	46	137	226	70	167	346
coeff_abs_level_greater1_flag[ ]	9-31	23	197	107	47	182	227	71	182	347
coeff_abs_level_greater2_flag[ ]	9-31	0	138	108	6	107	228	12	107	348
coeff_abs_level_greater2_flag[ ]	9-31	1	153	109	7	167	229	13	167	349
coeff_abs_level_greater2_flag[ ]	9-31	2	136	110	8	91	230	14	91	350
coeff_abs_level_greater2_flag[ ]	9-31	3	167	111	9	122	231	15	107	351
coeff_abs_level_greater2_flag[ ]	9-31	4	152	112	10	107	232	16	107	352
coeff_abs_level_greater2_flag[ ]	9-31	5	152	113	11	167	233	17	167	353
transform_skip_flag[ ][ ][0]	9-25	0	139	114	1	154	234	2	139	354
transform_skip_flag[ ][ ][1] transform_skip_flag[ ][ ][2]	9-25	3	139	115	4	154	235	5	139	355
explicit_rdpcm_flag[ ][ ][0]	9-32			116	0	139	236	1	139	356
explicit_rdpcm_flag[ ][ ][1] explicit_rdpcm_flag[ ][ ][2]	9-32			117	2	139	237	3	139	357
explicit_rdpcm_dir_flag[ ][ ][0]	9-33			118	0	139	238	1	139	358
explicit_rdpcm_dir_flag[ ][ ][1]explicit_rdpcm_dir_flag[ ][ ][2]	9-33			119	2	139	239	3	139	359

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.

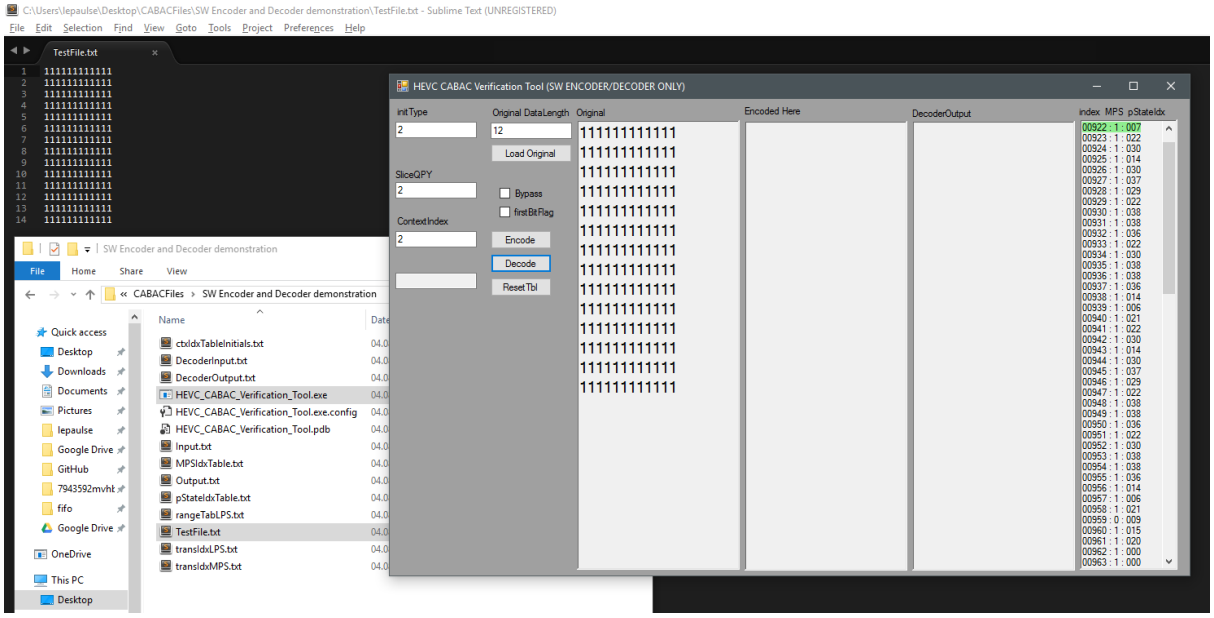
	A	B	C	D	E	F	G
1	K	N_min	N_max	Prefix bins	Suffix bins	Prefix length	Suffix length
2	2	0	3	0	-	1	2
3	Z	4	7	10	-	2	2
4	44	8	11	110	-	3	2
5		12	15	1110	-	4	2
6		16	23	11110	-	5	3
7		24	39	111110	-	6	4
8		40	71	1111110	00100	7	5
9		72	135	11111110	-	8	6
10		136	263	111111110	-	9	7
11		264	519	1111111110	-	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.

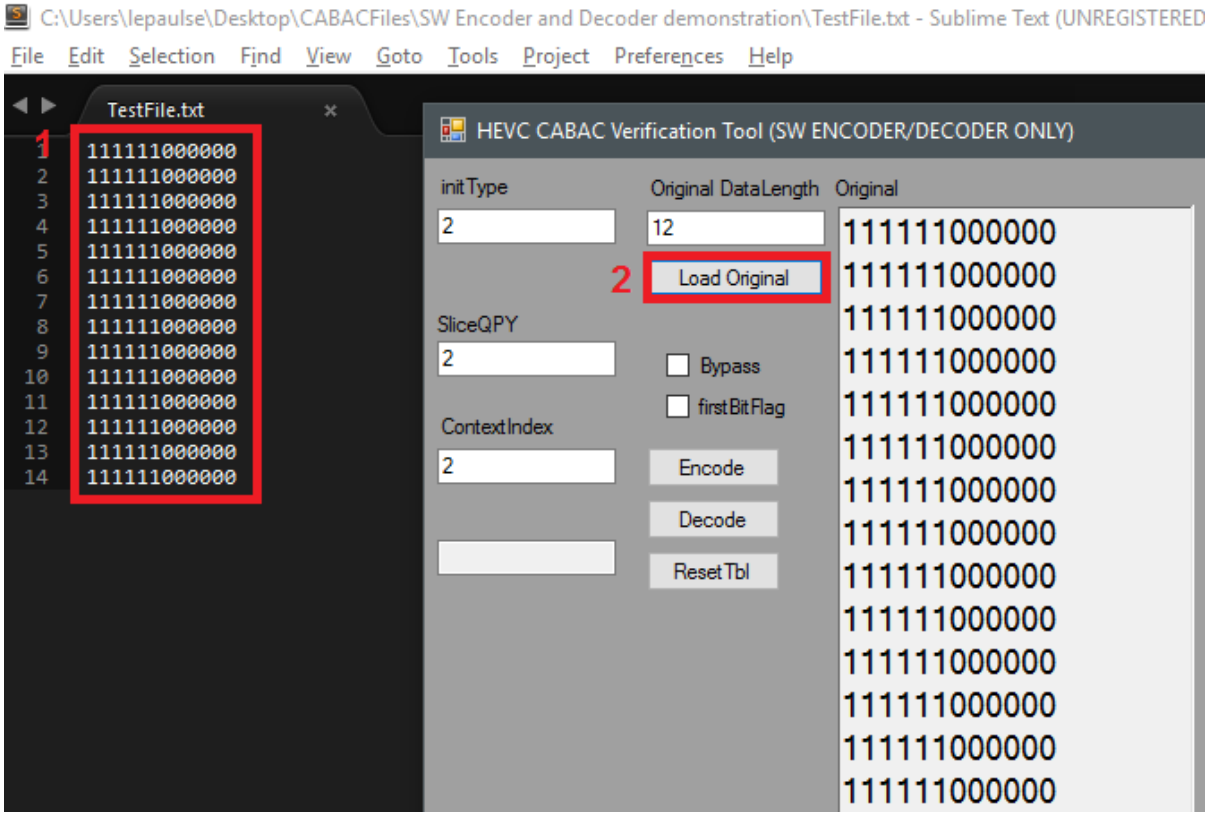
Note that due to the use of an extended DECTOBIN function(vDecimalToBinary), you might need to enable content to use the excel sheet. Because the vDecimalToBinary function only supports output of 16 or less binary symbols, Binarization of suffixes longer than 16-bits will not work properly.

Appendix I

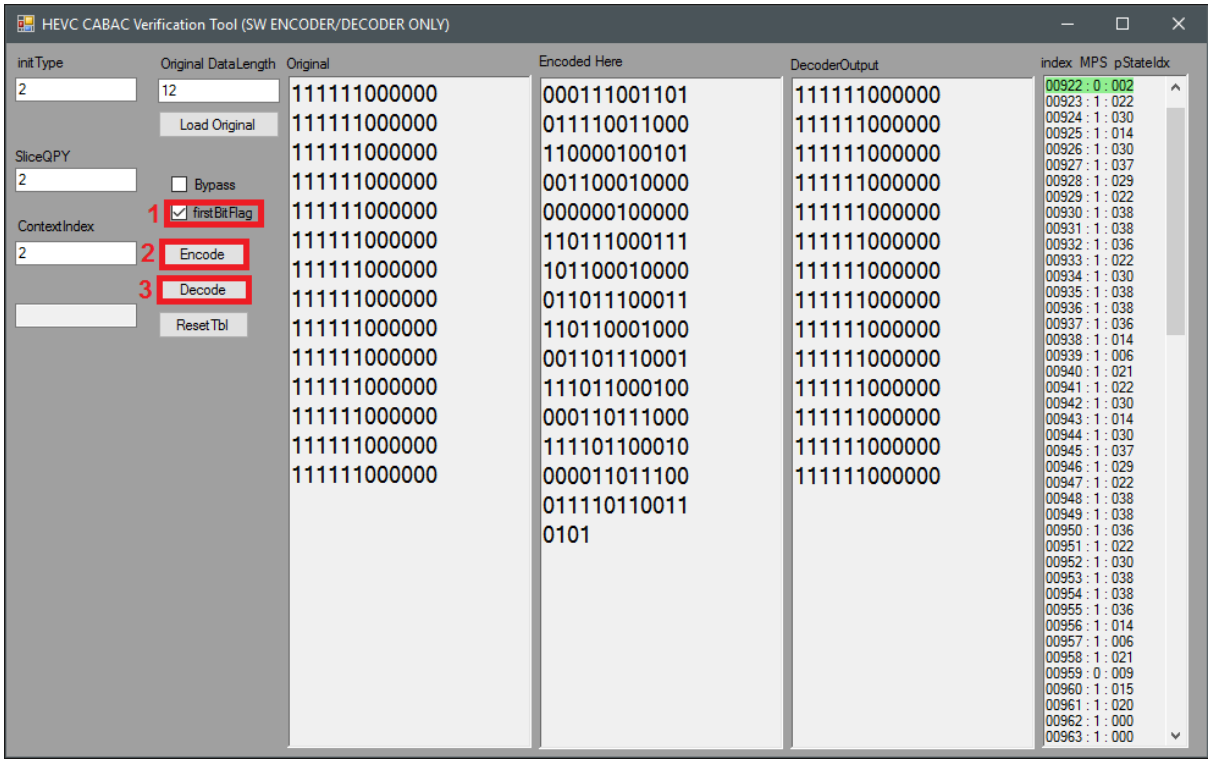
This Appendix shows how a simple demonstration of Software CABAC encoding/decoding using the C# software model. Relevant files is included in the delivery folder.
Navigate to the "SW Encoder and Decoder demonstration" folder and open "TestFile.txt" and the HEVC_CABAC_Verification_Tool as seen below.



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.

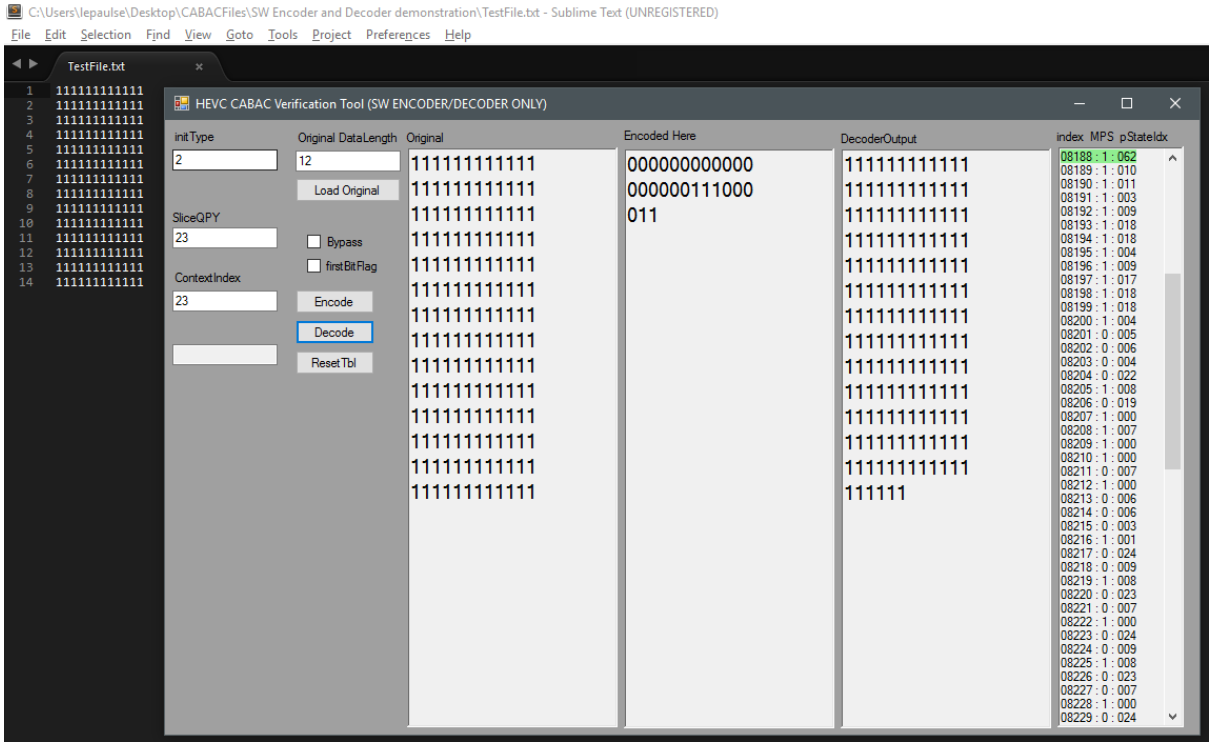


It is now possible to encode and decode. **1** Check the firstBitFlag checkbox(important). This will make sure the first bit is skipped in PutBit, and is required for the decoder to work **2** Press the Encode button. **3** Press the Decode button.



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.

Compression performance is heavily reliant on input data, as well as the initial value of the probability model at the selected context index.



It is possible to switch to bypass coding by checking the Bypass checkbox.