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 a truncation of TRk and EGk, with an adaptive level (k). A finite state machine approach was found, that proved to be a very efficient at coding the absolute remaining level. This approach was implemented in hardware.

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

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

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

Preface ................................................................. 1
Acknowledgment .................................................... 2
Summary and Conclusion ............................................ 3

1 Introduction ......................................................... 9
  1.1 Background .................................................. 9
  1.2 Objectives .................................................. 10
  1.3 Limitations ................................................ 10
  1.4 Approach .................................................. 11
  1.5 Features and Contributions ................................ 11
  1.6 Structure of the Report .................................... 12

2 Entropy and Arithmetic Coding .................................. 13
  2.1 Shannon Entropy ........................................... 14
  2.2 Entropy for binary strings ................................ 15
  2.3 Arithmetic Coding ......................................... 16

3 HEVC System and Data Structure ................................. 18
  3.1 Syntax Elements ............................................ 19
  3.2 CABAC Encoding ........................................... 20
  3.3 CABAC Decoding ............................................ 21

4 Binarization ......................................................... 22
  4.1 Binarization Processes .................................... 22
  4.2 Unary, Truncated Unary (TrU) Fixed-Length (FL) ........ 22
  4.3 Truncated Rice (TRk) ....................................... 23
  4.4 Exp-Golomb ............................................... 23
  4.5 Scan Direction ............................................. 24
  4.6 Transform-coefficient level data ......................... 25
  4.7 Coding of First Non Zero Element Coordinate ........... 26
  4.8 Coding of Absolute Level ................................ 27
  4.9 Coding of Sign ............................................ 32
  4.10 Grouping of Bins ......................................... 32
  4.11 Complete Example ......................................... 33

5 Context Modeling .................................................. 34
  5.1 sig_coeff_flag ............................................. 34
  5.2 sig_coeff_greater1_flag and sig_coeff_greater2_flag .... 34
  5.3 Probability model ......................................... 35
  5.4 Initialization ............................................. 35

6 HEVC CABAC Algorithm ............................................ 36
  6.1 Overview .................................................. 36
  6.2 Variable Initialization .................................... 36
  6.3 Encoding a Decision ....................................... 37
  6.4 Renormalization ........................................... 38
## List of Figures

1. Simple Unary coding based compression of byte data. ........................................ 13
2. Encoding of a single symbol. ........................................................................... 16
3. Encoding of a single symbol. ........................................................................... 16
4. Sequential Encoding of Three Symbols ............................................................. 17
5. AVC vs HEVC Picture Partitioning .................................................................... 18
6. Syntax Element Bitrate Distribution ................................................................. 19
7. CABAC encoding process for a $4 \times 4$ Transform Block. ................................ 20
8. CABAC decoding process for a $4 \times 4$ Transform Block. .............................. 21
9. Scan direction for Diagonal scan. ..................................................................... 24
10. Scan direction for vertical and horizontal scans. ............................................. 24
11. Residual data in the Transform Block to be Binarized .................................... 25
12. Binarization of last$_{sig}_{coeff}_{x/y}$-prefix in $4 \times 4$ Transform Blocks. ............ 26
13. Binarization of $Z$ using the SIG, ALG1, ALG2 and ALRem. .......................... 28
14. Binarization of $Z$ using the SIG, ALG1 and ALRem. .................................... 29
15. Binarization of $Z$ using the SIG and ALRem. ............................................... 30
16. ALRem code length for each value of the adaptive variable k. .......................... 31
17. Binarization of SIGN. ..................................................................................... 32
18. Complete Example Binarization of a $4 \times 4$ Transform Block. ....................... 33
19. SIG context assignment for $4 \times 4$ Transform Blocks.[9] ............................... 34
20. FSM based probability estimation in CABAC. .................................................. 35
21. Encoding of a binary decision. ....................................................................... 37
22. Renormalization during encoding of a binary decision. ................................... 38
23. PutBit procedure ............................................................................................ 39
24. Decoding of a binary decision. ....................................................................... 40
25. Software model user interface. ...................................................................... 41
26. Hardware and Software Interfacing .................................................................. 43
27. Planed Hardware Modules ............................................................................. 44
28. Binarization of sig$_{coeff}_{flag}$ ....................................................................... 46
29. Binarization of coeff$_{abs}_{level}_{greater1}_{flag}$ ................................................. 47
30. Binarization of coeff$_{abs}_{level}_{greater2}_{flag}$ ................................................ 48
31. Binarization of coeff$_{abs}_{level}_{sign}_{flag}$ ....................................................... 52
32. Hardware Encoder State Machine Diagram ................................................... 54
33. Asynchronous fifo. ......................................................................................... 63
List of Tables

1. Entropy of Symbols .................................................. 14
2. Entropy of Binary Strings .............................................. 15
3. Unary, Truncated Unary and Fixed Length code. ................. 22
4. Truncated Rice(TRk) Code ............................................. 23
5. Reverse order Exp-Golomb(EGk) code. .......................... 23
6. Selection of scan direction in $4 \times 4$ Transform Blocks. ......... 24
7. Syntax Elements detailed in this project. .......................... 25
8. Initial values for encoding and decoding. ........................ 36
9. Alternative representation of ALRem .............................. 49
10. Alternative representation of ALRem(calculation) ............... 50
11. Example Binarization of ALRem for $Z = 8$ and $k = 0$ .......... 50
12. Example Binarization of ALRem for $Z = 8$ and $k = 2$ .......... 50
13. Hardware Encoder State Sequence ................................. 55
14. Hardware Encoder State Sequence With Redundancies. ........... 55
15. Pipelined Hardware Encoder State Sequence ....................... 55
16. Encoder ports. ......................................................... 56
17. Hardware Encoder Parameters. ..................................... 57
18. Syntax Elements supported in the current context table. ........ 58
19. Hardware Encoder Context Table Structure ....................... 58
20. Range variable precision requirements. .......................... 59
21. Synthesis results ..................................................... 60
22. Synthesis results ..................................................... 60
23. CABAC Encoder maximum frequency for a few select parameters.. 61
24. Performance for bypass encoding. ................................ 62
25. Performance for regular encoding. ................................ 62
Appendixes:

Appendix A: VHDL Binarizer
Appendix B: VHDL Encoder
Appendix C: Verilog FIFO
Appendix D: C# Encoder
Appendix E: C# Decoder
Appendix F: System Setup
Appendix G: Context Table
Appendix H: Alternative Binarization of coeff_abs_level_remaining
Appendix I: Software Encoder and Decoder demonstration
1 Introduction

1.1 Background

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

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

HEVC seems to have lived up to its performance goal, as documented by Netflix in its large-scale study on video codecs published in 2016.[7] Using one of the leading open-source HEVC encoders, x265, and comparing it with the leading open-source AVC encoders, x264, as well as the VP9 reference encoder, libvpx. Netflix showed with their advanced video multiformat 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.](image)

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:

\[
H(X) = \sum_{i=1}^{n} P(x_i) \log_2 P(x_i) = - \sum_{i=1}^{n} P(x_i) \log_2 P(x_i)
\]

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 \( \lceil H(X) \rceil \).

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

<table>
<thead>
<tr>
<th>Symbol string</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>H per symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAABCC</td>
<td>1/2</td>
<td>1/3</td>
<td>1/6</td>
<td>0</td>
<td>0</td>
<td>1.459</td>
</tr>
<tr>
<td>ABCABC</td>
<td>1/4</td>
<td>1/3</td>
<td>1/3</td>
<td>0</td>
<td>0</td>
<td>1.585</td>
</tr>
<tr>
<td>AAAAAA</td>
<td>3/3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ABCD</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>AABB</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ABCDE</td>
<td>1/5</td>
<td>1/5</td>
<td>1/5</td>
<td>1/5</td>
<td>1/5</td>
<td>2.322</td>
</tr>
</tbody>
</table>

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

This also illustrates how entropy calculation is relative to your model. Consider the symbol string \{AAABCC\}. If you could map \{AAA\} \rightarrow \{D\} 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.

<table>
<thead>
<tr>
<th>Bin string</th>
<th>Probability</th>
<th>1</th>
<th>0</th>
<th>H per bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001111</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00110011</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>01010101</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11111110</td>
<td>7/8</td>
<td>1/8</td>
<td>0.543</td>
<td></td>
</tr>
<tr>
<td>00000011</td>
<td>1/4</td>
<td>3/4</td>
<td>0.811</td>
<td></td>
</tr>
<tr>
<td>11011001</td>
<td>5/8</td>
<td>3/8</td>
<td>0.954</td>
<td></td>
</tr>
</tbody>
</table>

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

This simple model works on a per bin basis, while it is easily observable that there exists patterns in the bin strings. \{00001111\} could be reduced down to \{01\} by introducing a dictionary that maps \{0000, 1111\} → \{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.

\[ X = \{A, B, C\} \quad P(X) = \{0.25, 0.25, 0.5\} \]

![Figure 2: Encoding of a single symbol.](image)

Output: 1

\[ X = \{A, B, C\} \quad P(X) = \{0.25, 0.25, 0.5\} \]

![Figure 3: Encoding of a single symbol.](image)

Output: 00

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.

\[ X = \{A, B, C\} \quad P(X) = \{0.25, 0.25, 0.5\} \]

Output: 01001

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 $4 \times 4$ 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 $16 \times 16$ samples. For HEVC the macroblocks has been replaced with the Coding tree unit(CTU), allowing for larger block structures($16 \times 16$, $32 \times 32$ or $64 \times 64$). This innovation is an important part of the coding efficiency improvements HEVC provides. Allowing for large low-complexity areas to be signaled more efficiently. The trade-off being a relative increase in encoding time, but with an added benefit of reduced decoding time.[8]

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

Pictures in HEVC 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 $16 \times 16$, $32 \times 32$ or $64 \times 64$. 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 $4 \times 4$, $8 \times 8$, $16 \times 16$ or $32 \times 32$, but only binarization and coding of TBs of size $4 \times 4$ is detailed here.

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

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

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

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

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

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

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

- **Binarization**: The first stage in the encoding is a pre-processing stage that converts the syntax elements into a binary representation more suitable for Binary Arithmetic Coding. Finished binarized syntax elements are commonly referred to as bins.

- **Context Modeling**: Selection of encoding mode for bins is performed. Bypass coding is selected for bins where the distribution is assumed to be uniform, and Regular is selected for bins where this assumption cannot 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 $4 \times 4$ Transform Block.](image)
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 $4 \times 4$ 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 $4 \times 4$ 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.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Unary(U)</th>
<th>Truncated Unary(TrU)</th>
<th>Fixed-Length(FL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>110</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>1110</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11110</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>111110</td>
<td>111110</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>1111110</td>
<td>1111110</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>11111110</td>
<td>11111110</td>
<td>111</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(cMax=3)</td>
<td>1(cMax=7)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>NA</td>
</tr>
<tr>
<td>5</td>
<td>NA</td>
</tr>
<tr>
<td>6</td>
<td>NA</td>
</tr>
<tr>
<td>7</td>
<td>NA</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>1001</td>
</tr>
<tr>
<td>4</td>
<td>1010</td>
</tr>
<tr>
<td>5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>110000</td>
</tr>
<tr>
<td>7</td>
<td>110001</td>
</tr>
<tr>
<td>8</td>
<td>110010</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Intra Prediction mode</th>
<th>0 to 5</th>
<th>6 to 14</th>
<th>15 to 21</th>
<th>22 to 30</th>
<th>31 to 34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan Direction</td>
<td>Diagonal</td>
<td>Vertical</td>
<td>Diagonal</td>
<td>Horizontal</td>
<td>Diagonal</td>
</tr>
</tbody>
</table>

Table 6: Selection of scan direction in $4 \times 4$ 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.

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 $4 \times 4$ TB simply consists of a $4 \times 4$ 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 ($16 \times 1$) array using the Diagonal Scan Direction. For simplicity, the figures in this chapter utilizes the $16 \times 1$ decimal representation for representing the Transform Block, but the finished binarizations are all in binary.

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.

![Figure 11: Residual data in the Transform Block to be Binarized](Image)

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Abbreviation</th>
<th>Binarization Process</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>last sig. coeff x, prefix</td>
<td>LAST</td>
<td>TR</td>
<td>Regular, Regular, Regular</td>
</tr>
<tr>
<td>last sig. coeff y, prefix</td>
<td>LAST</td>
<td>TR</td>
<td>Regular, Regular, Regular</td>
</tr>
<tr>
<td>sig. coeff flag</td>
<td>SIG</td>
<td>FL</td>
<td>Regular, Regular, Regular</td>
</tr>
<tr>
<td>coeff_abs_level_greater1_flag</td>
<td>ALG1</td>
<td>FL</td>
<td>Regular, Regular, Regular</td>
</tr>
<tr>
<td>coeff_abs_level_greater2_flag</td>
<td>ALG2</td>
<td>FL</td>
<td>Regular, Regular, Regular</td>
</tr>
<tr>
<td>coeff_abs_level_remaining</td>
<td>ALRem</td>
<td>TrU, TR and Egk</td>
<td>Bypass, Bypass, Bypass</td>
</tr>
<tr>
<td>coeff_sign_flag</td>
<td>SIGN</td>
<td>FL</td>
<td>Bypass, Bypass, Bypass</td>
</tr>
</tbody>
</table>

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 $4 \times 4$ a Fixed Length suffix is introduced,[9] but his will not be covered further here.

![Diagram showing binarization of last_sig_coeff_x/y_prefix in 4 x 4 Transform Blocks.](image)

**Figure 12:** Binarization of last_sig_coeff_x/y_prefix in 4 x 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 $4 \times 4$ 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.

$$Z = SIG + ALG1 + ALG2 + ALRem$$ (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 $B_1$. 

127
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.](image)
Figure 14: Binarization of \( Z \) from 0 to 15 using the SIG, ALG1, and ALRem syntax elements(ALG2 absent). ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the state after a first \( Z > 1 \) has occurred, and before 8 total \( Z > 1 \) has occurred. \( Z > 3 \) has also occurred, as shown by the fact that the adaptive variable \( k \) now equals 1.
Figure 15: Binarization of $Z$ from 0 to 15 using the SIG, ALG1, and ALRem syntax elements (ALG2 inferred 0). ALRem coding is distinguished with Orange for TRk and Yellow for EGk. This shows the state after a total of 8 $Z > 1$ has occurred. $Z > 3$ has also occurred, as shown by the fact that the adaptive variable $k$ now equals 1.
The coding efficiency of the ALRem binarization scheme is noteworthy. Figure 16 shows how it is able to change the effective region of lowest possible code length, by simply changing the adaption variable $k$. Note that due to the adaption rule ($k = \min(k+1, 4)$), this adaptive binarization scheme is most efficient when processing data of increasing magnitude. This could have been a motivating factor for implementing the multiple scan directions.

![Figure 16: ALRem code length for each value of the adaptive variable $k$.](image_url)
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.

<table>
<thead>
<tr>
<th>TB</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGN</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $4 \times 4$ Transform block, totaling 16-bits$\times 4 \times 4 = 256$-bits, is now reduced down to 81-bits using the HEVC adaptive binarization scheme. The size is now equal to about 32% of the original size, even before CABAC encoding is performed. Notice also how the symbols 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 probability skew of about 80% for the most probable symbol. These bins do of course not share the same probability models (contexts), but this does give some insight into why this binarization scheme is so efficient when paired with a CABAC encoder.

The Bypass coded bins (SIGN and ALRem) show a more equally probable distribution of symbols, with a count of 31 total ‘1’s and 24 total ‘0’s, and a probability skew of 56% for the most probable symbol. This is an example of why the higher throughput bypass mode can be utilized with an insignificant compression penalty. A major design goal of HEVC was to reduce the amount of regular coded bins, i.e. increase throughput. The adaptive binarization scheme for the transform coefficient level values was an important factor in achieving this goal.

![Figure 18: Complete Example, showing how the Transform Block from Figure 11 is binarized using the Diagonal Scan direction. Threshold variable values for each scan, as well as the condition that triggers a transition is also shown. Note that the red square under SIG is not signaled, but inferred directly from the LAST coordinates.](image-url)
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 substantially more advanced method, where context index calculation is based on a selection of previously processed bins.

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

5.2 sig_coeff_greater1_flag and sig_coeff_greater2_flag

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

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

**ALG1 subblock adaption rules:**

- At the start of a subblock, the context 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 contains 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 initialization/reset of the encoder or decoder should these variables be set to their initial values.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>codIRange</td>
<td>510</td>
</tr>
<tr>
<td>codILow</td>
<td>0</td>
</tr>
<tr>
<td>qCodIRangeidx</td>
<td>0</td>
</tr>
<tr>
<td>CodIRangeLPS</td>
<td>0</td>
</tr>
<tr>
<td>codIoffset</td>
<td>0</td>
</tr>
<tr>
<td>codILow</td>
<td>0</td>
</tr>
<tr>
<td>BitsOutstanding</td>
<td>NA</td>
</tr>
<tr>
<td>firstBitFlag</td>
<td>NA</td>
</tr>
</tbody>
</table>

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 bitstream 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.

![Diagram of PutBit function](image)

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

Figure 24: Decoding of a binary decision.
7 Software Model

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

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

![Software model user interface](image)

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 planned for both the Binarizer/Context Modeller and the Hardware Encoder, but it was only completed for the Hardware Encoder. The asynchronous fifo was only verified using the accompanying testbench.

Figure 26: System structure for interfacing with between hardware and software. Including development platforms. Note that the Binarizer/Context 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 programmable SoC.

8.1 Modules

Figure 27: The planned 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 $4 \times 4$ Transform Blocks. Several methods for binarization of the relevant syntax elements were explored, with the main focus of finding an efficient implementation of coding the ALRem syntax element. This implementation was designed without a predefined interface from the rest of the encoder. Therefore this work should only serve as inspiration for designing a compliant binarizer. Further work should also try to incorporate both binarization and context index calculation into a single module. The examples here 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.

![Diagram of Binarization of sig_coeff_flag](image)

**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 (plus the inferred first ‘1’ seen in red in Fig. 31), but has a maximum of 8 per subblock.

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.

```
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
```

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 coef_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 coef_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.

<table>
<thead>
<tr>
<th>TabIdx</th>
<th>Z_{min}</th>
<th>Z_{max}</th>
<th>Prefix bins</th>
<th>Suffix bins</th>
<th>Prefix Length</th>
<th>Suffix Length</th>
<th>Max k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2^{k-1}</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1 \times 2^k</td>
<td>2 \times 2^k - 1</td>
<td>C</td>
<td>10</td>
<td>2</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2 \times 2^k</td>
<td>3 \times 2^k - 1</td>
<td>C</td>
<td>110</td>
<td>3</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>1110</td>
<td>4</td>
<td>4</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11110</td>
<td>xC</td>
<td>5</td>
<td>1 + k</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>111110</td>
<td>xxC</td>
<td>6</td>
<td>2 + k</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>1111110</td>
<td>xxxC</td>
<td>7</td>
<td>3 + k</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11111110</td>
<td>xxxxC</td>
<td>8</td>
<td>4 + k</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11111110xxxC</td>
<td>9</td>
<td>5 + k</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11111110xxxxxC</td>
<td>10</td>
<td>6 + k</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11111110xxxxxxxC</td>
<td>11</td>
<td>7 + k</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>111111110xxxxxxxxC</td>
<td>12</td>
<td>8 + k</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>1111111110xxxxxxxxC</td>
<td>13</td>
<td>9 + k</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11111111110xxxxxxxC</td>
<td>14</td>
<td>10 + k</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>111111111110xxxxxxxC</td>
<td>15</td>
<td>11 + k</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>1111111111110xxxxxxxC</td>
<td>16</td>
<td>12 + k</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>11111111111110xxxxxxxC</td>
<td>17</td>
<td>13 + k</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2^k \times (2^k+2)</td>
<td>2^k \times (2^k+2) - 1</td>
<td>111111111111110xxxxxxxC</td>
<td>18</td>
<td>14 + k</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>TabIdx</th>
<th>$Z_{\text{min}}$</th>
<th>$Z_{\text{max}}$</th>
<th>Prefix bins</th>
<th>Suffix bins</th>
<th>Prefix Length</th>
<th>Suffix Length</th>
<th>Max k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$K^2 - 1$</td>
<td>0</td>
<td>$Z - Z_{\text{min}}$</td>
<td>1</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>$1 \times 2^k$</td>
<td>$2 \times 2^k - 1$</td>
<td>10</td>
<td>$Z - Z_{\text{min}}$</td>
<td>2</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>$2 \times 2^k$</td>
<td>$3 \times 2^k - 1$</td>
<td>110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>3</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>1110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>4</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>11110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>5</td>
<td>1 + k</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>6</td>
<td>2 + k</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>1111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>7</td>
<td>3 + k</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>11111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>8</td>
<td>4 + k</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>11111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>9</td>
<td>5 + k</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>11111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>10</td>
<td>6 + k</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>11</td>
<td>7 + k</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>12</td>
<td>8 + k</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>13</td>
<td>9 + k</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>14</td>
<td>10 + k</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>15</td>
<td>11 + k</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>16</td>
<td>12 + k</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>17</td>
<td>13 + k</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>$2^k (2^k + 2)$</td>
<td>$2^k (2^k + 2) - 1$</td>
<td>1111111110</td>
<td>$Z - Z_{\text{min}}$</td>
<td>18</td>
<td>14 + k</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>TabIdx</th>
<th>$Z_{\text{min}}$</th>
<th>$Z_{\text{max}}$</th>
<th>Prefix bins</th>
<th>Suffix bins</th>
<th>Prefix Length</th>
<th>Suffix Length</th>
<th>Max k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>11</td>
<td>110</td>
<td>00</td>
<td>3</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>15</td>
<td>110</td>
<td>00</td>
<td>4</td>
<td>k</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>23</td>
<td>11110</td>
<td>00</td>
<td>5</td>
<td>1 + k</td>
<td>4</td>
</tr>
</tbody>
</table>

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’ in sig_coeff_flag) has a value of less than 0. This is a very simple binarization that only requires checking of the sign bit. The only real complexity lies in supporting the optional sign bit hiding technique. Which in practice involves skipping this binarization under certain conditions.

Figure 31: Binarization of coeff_abs_level_sign_flag
8.5 Context Index Calculator

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

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

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

Encoding of termination is implemented, but not included in this diagram. The initial plan was to decouple the Bypass and Regular encoders into two distinct modules. This would allow for implementation of a dispatcher to split the workload whenever a regular and a bypass coded bin are encoded in order. This did however introduce a few complications to the update of the range variables. Bypass encoding was instead incorporated into the regular coder, as a simple state. This allowed for both the Bypass and Regular parts of the encoder to use the same PutBit state, as well as sharing range variables. The main challenge in implementing the hardware encoder lies in correctness of the nested while loops located in the RenormE and PutBit part of the algorithm. These while loops contributed the largest amount of non-trivial bugs during development, and was the main motivator for developing the software model. Another major challenge is optimizing throughput using efficient pipelining, but due to the time constraints and the complexity of such designs, this was not explored further.
The focus of the current implementation was achieving correct output of the encoder. As can be seen by the code, the resemblance to the specification flowcharts is clear. A consequence of this is that a large amount of possible redundant clock cycles, where the whole cycle is spent checking a simple conditional statement. This is a result of the while-loops in the specification. During `enc_bin_r` a conditional check to see if the output is ready to be written or if a renormalization is required. This conditional check could possibly be moved to the end of the renormalization, effectively skipping this cycle. Further the `RenormE` state includes a conditional check that could possibly be moved as a condition for entering this state. The `bitsOutstanding` loop located in the `PutBit` also introduces complications.

<table>
<thead>
<tr>
<th>S.E</th>
<th>Type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Regular r_input r_ctx enc_bin_r RenormE PutBit RenormE <code>enc_bin_r</code> w_output</td>
<td>r_input <code>enc_bin_r</code> PutBit <code>enc_bin_r</code> w_output</td>
</tr>
<tr>
<td>2</td>
<td>Bypass</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Syntax Element Nr.</th>
<th>Type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Regular r_input r_ctx enc_bin_r RenormE PutBit RenormE <code>enc_bin_r</code> w_output</td>
<td>r_input <code>enc_bin_r</code> PutBit <code>enc_bin_r</code> w_output</td>
</tr>
<tr>
<td>2</td>
<td>Bypass</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Syntax Element Nr.</th>
<th>Type</th>
<th>Pipeline Stage</th>
</tr>
</thead>
</table>
| 1                  | Regular r_input r_ctx enc_bin_r RenormE PutBit RenormE `enc_bin_r` w_output | Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall Stall St
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.

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

Table 16: Encoder ports.

Some redundant ports are present. Offsets from SliceQPY and initType could possibly 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.

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

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

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Binarization Process</th>
<th>initType = 0</th>
<th>initType = 1</th>
<th>initType = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>last_sig.coeff_x_prefix</td>
<td>TR</td>
<td>0 - 17</td>
<td>120 - 137</td>
<td>240 - 257</td>
</tr>
<tr>
<td>last_sig.coeff_y_prefix</td>
<td>TR</td>
<td>18 - 35</td>
<td>138 - 155</td>
<td>258 - 175</td>
</tr>
<tr>
<td>sig.coeff_flag</td>
<td>FL</td>
<td>40 - 83</td>
<td>160 - 203</td>
<td>280 - 323</td>
</tr>
<tr>
<td>coeff.abs_level_greater1_flag</td>
<td>FL</td>
<td>84 - 107</td>
<td>204 - 227</td>
<td>324 - 347</td>
</tr>
<tr>
<td>coeff.abs_level_greater2_flag</td>
<td>FL</td>
<td>108 - 113</td>
<td>228 - 133</td>
<td>348 - 353</td>
</tr>
<tr>
<td>coeff.abs_level_remaining</td>
<td>TrU, TRk and EGk</td>
<td>Bypass</td>
<td>Bypass</td>
<td>Bypass</td>
</tr>
<tr>
<td>coeff.sign_flag</td>
<td>FL</td>
<td>Bypass</td>
<td>Bypass</td>
<td>Bypass</td>
</tr>
</tbody>
</table>

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

Context 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.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctxIdxTableInitials</td>
<td>18720 Initialized as read-only memory. Contains the proper initialization variables for all values of SliceQPY and initType.</td>
</tr>
<tr>
<td>ctxIdxTable</td>
<td>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.</td>
</tr>
<tr>
<td>currCtx</td>
<td>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.</td>
</tr>
</tbody>
</table>

| 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 \textit{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 \textit{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 \textit{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\cite{NOTE2} 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 \textit{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.

<table>
<thead>
<tr>
<th>Range Variable</th>
<th>Required Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bypass</td>
</tr>
<tr>
<td>ivlLow</td>
<td>11</td>
</tr>
<tr>
<td>ivlCurrRange</td>
<td>9</td>
</tr>
<tr>
<td>ivlLpsRange</td>
<td>8</td>
</tr>
<tr>
<td>qRangeldx</td>
<td>NA</td>
</tr>
<tr>
<td>pStateldx</td>
<td>NA</td>
</tr>
<tr>
<td>valMps</td>
<td>NA</td>
</tr>
</tbody>
</table>

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, \textit{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.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>2930</td>
<td>53200</td>
<td>5.51</td>
</tr>
<tr>
<td>LUTRAM</td>
<td>20</td>
<td>17400</td>
<td>0.11</td>
</tr>
<tr>
<td>FF</td>
<td>150</td>
<td>106400</td>
<td>0.14</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>2892</td>
<td>53200</td>
<td>5.44</td>
</tr>
<tr>
<td>LUTRAM</td>
<td>20</td>
<td>17400</td>
<td>0.11</td>
</tr>
<tr>
<td>FF</td>
<td>96</td>
<td>106400</td>
<td>0.09</td>
</tr>
</tbody>
</table>

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.

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

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.

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

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.

<table>
<thead>
<tr>
<th>Syntax Element Length</th>
<th>Encoding Time ns</th>
<th>cycles</th>
<th>bins/cycle</th>
<th>Mb/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3901000</td>
<td>462039,5581</td>
<td>0,216431685</td>
<td>25,6344527</td>
</tr>
<tr>
<td>5</td>
<td>1429500</td>
<td>169311,8555</td>
<td>0,295313047</td>
<td>34,9772648</td>
</tr>
<tr>
<td>1</td>
<td>607000</td>
<td>71893,87638</td>
<td>0,139093905</td>
<td>16,4744646</td>
</tr>
</tbody>
</table>

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.

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.

![Figure 33: Asynchronous fifo.](image-url)
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


Appendix A

-- Binarizer for HEVC H.265
-- 01/03/2017
-- Norwegian University of Science and Technology
-- Lars Erik Songe Paulsen
-- TODO LIST:
-- ********************************************************************************

--pragma synthesis_off
-- your simulation-only code
--pragma synthesis_on

library ieee;
library std;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;

library work;
use work.BinarizerParameters.all;

entity Binarizer is

port ( 
  Clk : in std_logic;
  DataIn : in std_logic_vector((16*CoeffWidth)-1 downto 0);
  ScanDir : in std_logic_vector(1 downto 0);
  Resetn : in std_logic;
  StartBinarizer : in std_logic;
  DataOut : out std_logic_vector(OutputWidth-1 downto 0);
  DataLength : out std_logic_vector(OutputWidthLength-1 downto 0);
  PrefixLength : out std_logic_vector(OutputWidthLength-1 downto 0);
  Finished : out std_logic 
);

end Binarizer;

architecture struct of Binarizer is

-- Signal declarations
-- ********************************************************************************

type BinarizeStateType is(
  read_input,
  write_last_sig_coeff_x_prefix,
  write_last_sig_coeff_y_prefix,
  write_sig_coeff_flag,
  write_coeff_abs_level_greater1,
  write_coeff_abs_level_greater2,
  write_coeff_abs_level_remaining,
  write_coeff_sign_flag,
  write_finished
);

type transform_block is array(15 downto 0) of integer range -(2**(CoeffWidth-1)) to (2**((CoeffWidth-1)));

signal BinarizeState : BinarizeStateType;
signal coefficients : Transform_Block;
signal coeff_abs_level_greater1_flag : std_logic_vector(0 to 15);
signal coeff_abs_level_greater2_flag : std_logic_vector(0 to 15);
signal sig_coeff_flag : std_logic_vector(0 to 15);
signal ABS_index : integer range 0 to 15;
signal ABS_started : std_logic;
signal ABS_done : std_logic;
signal ABS_level_writeout : integer range 0 to 32767;
signal k : integer range 0 to 4;

begin
   Next_ABS : process(Clk, DataIn, Resetn, StartBinarizer)
   variable ABS_index_cycler : integer range 0 to 15;
   begin
      if Resetn = '0' then
         ABS_index_cycler := 0;
         ABS_index <= 0;
         ABS_done <= '0';
      elsif rising_edge(Clk) then
         case BinarizeState is
            when sync | write_coeff_abs_level_remaining =>
               find_next_ALG2_loop : for ABS_index_cycler in 0 to 15 loop
                  if (ABS_index_cycler <= ABS_index) then
                     --spin TODO FIX end condition
                  else
                     if(coeff_abs_level_greater1_flag(ABS_index_cycler) = '1') then
                        report "ALG found at ABS_index[" & integer'image(ABS_index_cycler) & "]";
                        ABS_index <= ABS_index_cycler;
                        ABS_level_writeout <= coefficients(ABS_index_cycler);
                        if(ABS_index_cycler = 15) then
                           ABS_done <= '1';
                        else
                           exit find_next_ALG2_loop;
                        end if;
                     end if;
                  end if;
               end loop;
            when others =>
               ABS_index <= 0;
               ABS_done <= '0';
            end case;
         end if;
      end process;

      Binarize : process(Clk, DataIn, Resetn, StartBinarizer)
      Begin
         if Resetn = '0' then
            sig_coeff_flag <= (others => '0');
            sig_coeff_flag_found := '0';
            ALG1_index := 0;
            SIGN_index := 0;
            DataOut <= (others => '0');
            BinarizeState <= read_input;
            Finished <= '1';
            coeff_abs_level_greater1_flag <= (others => '0');
            coeff_abs_level_greater2_flag <= (others => '0');
            k <= 0;
         end if;
      end process;
   end process;
end;
}
PrefixLength <= (others => '0');

elsif rising_edge(Clk) then
  case BinarizeState is
  when read_input =>
    BinarizeState <= write_last_sig_coeff_x_prefix;
    Finished <= '0';
  for i in 0 to 15 loop
    -- Read absolute values.
    coefficients(Scan_Direction(15-i)) <=
      to_integer(abs(signed(DataIn(((CoeffWidth*(i+1))-1) downto (CoeffWidth*i))));
  end loop;
  -- Read sign.
  coeff_sign_flag(Scan_Direction(15-i)) := DataIn((CoeffWidth*(i+1))-1);
  -- Find non-zero data.
  sig_coeff_flag(Scan_Direction(15-i)) <= or_reduce(DataIn(((CoeffWidth*(i+1))-2) downto (CoeffWidth*i)));
  -- Find >1 data and >2 data. Procedure dependent on sign bit.
  case DataIn((CoeffWidth*(i+1))-1) is
    when '0' =>
      coeff_abs_level_greater1_flag(Scan_Direction(15-i)) <=
        or_reduce(DataIn(((CoeffWidth*(i+1))-2) downto (CoeffWidth*i+1)))
        or
        or_reduce(DataIn(((CoeffWidth*(i+1))-2) downto (CoeffWidth*i+2)));
    when '1' =>
      coeff_abs_level_greater1_flag(Scan_Direction(15-i)) <=
        nand_reduce(DataIn(((CoeffWidth*(i+1))-2) downto (CoeffWidth*i+1)))
        or
        nand_reduce(DataIn(((CoeffWidth*(i+1))-2) downto (CoeffWidth*i+2)));
    when others =>
      end case;
  end loop;
  when write_last_sig_coeff_x_prefix =>
  for i in 0 to 15 loop
    report "coefficients(" & integer'image(i) & ")": " &
      integer'image(coefficients(i));
  end loop;
  report "sig_coeff_flag": " & integer'image(conv_integer(sig_coeff_flag));
  report "coeff_abs_level_greater1_flag": " &
    integer'image(conv_integer(coeff_abs_level_greater1_flag));
  report "coeff_abs_level_greater2_flag": " &
    integer'image(conv_integer(coeff_abs_level_greater2_flag));
  report "coeff_sign_flag": " & integer'image(conv_integer(coeff_sign_flag));
  BinarizeState <= write_last_sig_coeff_y_prefix;
  write_last_sig_coeff_x_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));
    end case;
when 1 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 2 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 3 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 4 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 5 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 6 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 7 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(2,OutputWidthLength));

when 8 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 9 => DataOut(OutputWidth-1) <= '0';
DataLength <=
\rightarrow std_logic_vector(to_unsigned(1,OutputWidthLength));

when 10 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));

when 11 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(2,OutputWidthLength));

when 12 => DataOut(OutputWidth-1) <= '0';
DataLength <=
\rightarrow std_logic_vector(to_unsigned(1,OutputWidthLength));

when 13 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
DataLength <=
\rightarrow std_logic_vector(to_unsigned(2,OutputWidthLength));

when 14 => DataOut(OutputWidth-1) <= '0';
DataLength <=
\rightarrow std_logic_vector(to_unsigned(1,OutputWidthLength));

when 15 => DataOut(OutputWidth-1) <= '0';
DataLength <=
\rightarrow 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 <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));
when 1 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
  DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));
when 2 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
  DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));
when 3 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
  DataLength <=
\rightarrow std_logic_vector(to_unsigned(2,OutputWidthLength));
when 4 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
  DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));
when 5 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
  DataLength <=
\rightarrow std_logic_vector(to_unsigned(3,OutputWidthLength));
when 6 => DataOut(OutputWidth-1) <= '0';
  DataLength <=
\rightarrow std_logic_vector(to_unsigned(1,OutputWidthLength));
  end case;
  exit write_last_sig_coeff_y_prefix_loop;
end if;
end loop;
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 7 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
when 8 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
when 9 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "111";
DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
when 10 => DataOut(OutputWidth-1) <= '0';
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 11 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
when 12 => DataOut(OutputWidth-1 downto OutputWidth-3) <= "110";
DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
when 13 => DataOut(OutputWidth-1) <= '0';
DataLength <= std_logic_vector(to_unsigned(15-index,OutputWidthLength));
coefficients(index) <= coefficients(index) - 1;
when 14 => DataOut(OutputWidth-1 downto OutputWidth-2) <= "10";
DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
when 15 => DataOut(OutputWidth-1) <= '0';
end case;
exit write_last_sig_coeff_y_prefix_loop;
end if;
end loop;
when write_sig_coeff_flag =>
BinarizeState <= write_coeff_abs_level_greater1;
write_coeff_abs_level_greater1_loop : for index in 0 to 15 loop
if sig_coeff_flag_found = '1' then
DataOut(sig_coeff_flag_index) <= sig_coeff_flag(index);
sig_coeff_flag_index := sig_coeff_flag_index + 1;
if(sig_coeff_flag(index) = '1') then
coefficients(index) <= coefficients(index) - 1;
end if;
elsif(sig_coeff_flag(index) = '1') then
sig_coeff_flag_found := '1';
DataLength <= std_logic_vector(to_unsigned(15-index,OutputWidthLength));
coefficients(index) <= coefficients(index) - 1;
end if;
end loop;
when write_coeff_abs_level_greater1 =>
BinarizeState <= write_coeff_abs_level_greater2;
write_coeff_abs_level_greater2_loop : for index in 0 to 15 loop
if sig_coeff_flag(index) = '1' and ALG1_index < 8 then
if( coeff_abs_level_greater1_flag(index) = '1') then
DataOut(OutputWidth-1-ALG1_index) <= '1';
report "ALG1:coefficients(" & integer'image(index) & ")": " & integer'image(coefficients(index));
coefficients(index) <= coefficients(index) - 1;
end if;
else
DataOut(OutputWidth-1-ALG1_index) <= '0';
end if;
ALG1_index := ALG1_index + 1;
end if;
end loop;
DataLength <= std_logic_vector(to_unsigned(ALG1_index,OutputWidthLength));
when write_coeff_abs_level_greater2 =>
BinarizeState <= sync;
write_coeff_abs_level_greater_2_loop : for index in 0 to 15 loop
if sig_coeff_flag(index) = '1' then
if (coeff_abs_level_greater2_flag(index) = '1') then
  DataOut(OutputWidth-1) <= '1';
  report "ALG2:coefficients(" & integer'image(index) & ": ") & 
  " integer'image(coefficients(index));
  coefficients(index) <= coefficients(index) - 1;
  report "ALG2:coefficients(" & integer'image(index) & ": ") & 
  " integer'image(coefficients(index));
else
  DataOut(OutputWidth-1) <= '0';
end if;
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
exit write_coeff_abs_level_greater_2_loop;
end if;
end loop;
when sync =>
  BinarizeState <= write_coeff_abs_level_remaining;
for i in 0 to 15 loop
  report "coefficients(" & integer'image(i) & ": ") & 
  " integer'image(coefficients(i));
end loop;
when write_coeff_abs_level_remaining =>
  report "write_coeff_abs_level_remaining: @ k: ") & integer'image(k) & 
  "ABS_level_writeout: ") & integer'image(ABS_level_writeout);
  if(ABS_done = '1') then
    BinarizeState <= write_coeff_sign_flag;
  end if;
case k is
  when 0 =>
    case (ABS_level_writeout) is
    when 0 =>
      DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
      PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
    when 1 =>
      DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
      PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
    when 2 =>
      DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
      PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
    when 3 =>
      DataLength <= std_logic_vector(to_unsigned(0,OutputWidthLength));
      PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
    when 4 to 5 =>
      DataOut(OutputWidth-1 downto OutputWidth-1) <= 
      std_logic_vector(to_unsigned(ABS_level_writeout-4,1));
      DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
      PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
    k <= 1;
      when 6 to 9 =>
        DataOut(OutputWidth-1 downto OutputWidth-2) <= 
        std_logic_vector(to_unsigned(ABS_level_writeout-6,2));
        DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
        PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
      when 10 to 17 =>
        DataOut(OutputWidth-1 downto OutputWidth-3) <= 
        std_logic_vector(to_unsigned(ABS_level_writeout-10,3));
        DataLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
        PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
      when 18 to 33 =>
        DataOut(OutputWidth-1 downto OutputWidth-4) <= 
        std_logic_vector(to_unsigned(ABS_level_writeout-18,4));
        DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
        PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
      when 34 to 65 =>
        DataOut(OutputWidth-1 downto OutputWidth-5) <= 
        std_logic_vector(to_unsigned(ABS_level_writeout-34,5));
        DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
        PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
      when 66 to 129 =>
        DataOut(OutputWidth-1 downto OutputWidth-6) <= 
        std_logic_vector(to_unsigned(ABS_level_writeout-66,6));
DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
when 130 to 257 =>
DataOut(OutputWidth-1 downto OutputWidth-7) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-130,7));
DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
when 258 to 513 =>
DataOut(OutputWidth-1 downto OutputWidth-8) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-258,8));
DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
when 514 to 1025 =>
DataOut(OutputWidth-1 downto OutputWidth-9) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-514,9));
DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
when 1026 to 2049 =>
DataOut(OutputWidth-1 downto OutputWidth-10) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-1026,10));
DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
when 2050 to 4097 =>
DataOut(OutputWidth-1 downto OutputWidth-11) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-2050,11));
DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
when 4098 to 8193 =>
DataOut(OutputWidth-1 downto OutputWidth-12) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-4098,12));
DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
when 8194 to 16385 =>
DataOut(OutputWidth-1 downto OutputWidth-13) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-8194,13));
DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(17,OutputWidthLength));
when 16386 to 32767 =>
DataOut(OutputWidth-1 downto OutputWidth-14) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-16386,14));
DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(18,OutputWidthLength));
when others =>
end case;
when 1 =>
case (ABS_level_writeout) is
when 0 to 1 =>
DataOut(OutputWidth-1 downto OutputWidth-1) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout,1));
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 2 to 3 =>
DataOut(OutputWidth-1 downto OutputWidth-1) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-2,1));
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
when 4 to 5 =>
DataOut(OutputWidth-1 downto OutputWidth-1) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-4,1));
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
when 6 to 7 =>
DataOut(OutputWidth-1 downto OutputWidth-1) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-6,1));
DataLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
when 8 to 11 =>
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(5,OutputWidthLength));
when 12 to 19 =>
DataOut(OutputWidth-1 downto OutputWidth-3) <=
→ std_logic_vector(to_unsigned(ABS_level_writeout-12,3));
when 20 to 35 =>
  DataOut(OutputWidth-1 downto OutputWidth-4) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-20,4));
  DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
when 36 to 67 =>
  DataOut(OutputWidth-1 downto OutputWidth-5) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-36,5));
  DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
when 68 to 131 =>
  DataOut(OutputWidth-1 downto OutputWidth-6) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-68,6));
  DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
when 132 to 259 =>
  DataOut(OutputWidth-1 downto OutputWidth-7) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-132,7));
  DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
when 260 to 515 =>
  DataOut(OutputWidth-1 downto OutputWidth-8) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-260,8));
  DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
when 516 to 1027 =>
  DataOut(OutputWidth-1 downto OutputWidth-9) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-516,9));
  DataLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
when 1028 to 2051 =>
  DataOut(OutputWidth-1 downto OutputWidth-10) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-1028,10));
  DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
when 2052 to 4099 =>
  DataOut(OutputWidth-1 downto OutputWidth-11) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-2052,11));
  DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
when 4100 to 8195 =>
  DataOut(OutputWidth-1 downto OutputWidth-12) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-4100,12));
  DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
when 8196 to 16387 =>
  DataOut(OutputWidth-1 downto OutputWidth-13) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-8196,13));
  DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(15,OutputWidthLength));
when 16388 to 32767 =>
  DataOut(OutputWidth-1 downto OutputWidth-14) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-16388,14));
  DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(16,OutputWidthLength));
when others =>
  DataOut(OutputWidth-1 downto OutputWidth-2) <=
    std_logic_vector(toUnsigned(ABS_level_writeout,2));
  DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 2 to 3 =>
  DataOut(OutputWidth-1 downto OutputWidth-2) <=
    std_logic_vector(toUnsigned(ABS_level_writeout,2));
  DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 4 to 7 =>
  DataOut(OutputWidth-1 downto OutputWidth-2) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-4,2));
  DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 8 to 11 =>
  DataOut(OutputWidth-1 downto OutputWidth-2) <=
    std_logic_vector(toUnsigned(ABS_level_writeout-8,2));
DataLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(5,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;
end 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));
DataLength <= std_logic_vector(to_unsigned(3, OutputWidthLength));
PrefixLength <= std_logic_vector(to_unsigned(1, OutputWidthLength));
when 8 to 15 =>
  DataOut(OutputWidth-1 downto OutputWidth-3) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-8,3));
  DataLength <= std_logic_vector(to_unsigned(3, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(2, 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(3, OutputWidthLength));
when 24 to 31 =>
  DataOut(OutputWidth-1 downto OutputWidth-3) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-24,3));
  DataLength <= std_logic_vector(to_unsigned(4, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(4, OutputWidthLength));
when 32 to 47 =>
  DataOut(OutputWidth-1 downto OutputWidth-4) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-32,4));
  DataLength <= std_logic_vector(to_unsigned(4, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(5, OutputWidthLength));
when 48 to 79 =>
  DataOut(OutputWidth-1 downto OutputWidth-5) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-48,5));
  DataLength <= std_logic_vector(to_unsigned(6, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(6, OutputWidthLength));
when 80 to 143 =>
  DataOut(OutputWidth-1 downto OutputWidth-6) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-80,6));
  DataLength <= std_logic_vector(to_unsigned(6, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(7, OutputWidthLength));
when 144 to 271 =>
  DataOut(OutputWidth-1 downto OutputWidth-7) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-144,7));
  DataLength <= std_logic_vector(to_unsigned(8, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(8, OutputWidthLength));
when 272 to 527 =>
  DataOut(OutputWidth-1 downto OutputWidth-8) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-272,8));
  DataLength <= std_logic_vector(to_unsigned(8, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(9, OutputWidthLength));
when 528 to 1039 =>
  DataOut(OutputWidth-1 downto OutputWidth-9) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-528,9));
  DataLength <= std_logic_vector(to_unsigned(9, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(10, OutputWidthLength));
when 1040 to 2063 =>
  DataOut(OutputWidth-1 downto OutputWidth-10) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-1040,10));
  DataLength <= std_logic_vector(to_unsigned(10, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(11, OutputWidthLength));
when 2064 to 4111 =>
  DataOut(OutputWidth-1 downto OutputWidth-11) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-2064,11));
  DataLength <= std_logic_vector(to_unsigned(11, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(12, OutputWidthLength));
when 4112 to 8207 =>
  DataOut(OutputWidth-1 downto OutputWidth-12) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-4112,12));
  DataLength <= std_logic_vector(to_unsigned(12, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(13, OutputWidthLength));
when 8208 to 16399 =>
  DataOut(OutputWidth-1 downto OutputWidth-13) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-8208,13));
  DataLength <= std_logic_vector(to_unsigned(13, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(14, OutputWidthLength));
when 16400 to 32767 =>
  DataOut(OutputWidth-1 downto OutputWidth-14) <=
  std_logic_vector(to_unsigned(ABS_level_writeout-16400,14));
  DataLength <= std_logic_vector(to_unsigned(14, OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(15, OutputWidthLength));
when others =>
when 10 to 15 =>
  DataOut(OutputWidth-1 downto OutputWidth-4) <=
    std_logic_vector(to_unsigned(ABS_level_writeout,4));
  DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(1,OutputWidthLength));
when 16 to 31 =>
  DataOut(OutputWidth-1 downto OutputWidth-4) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-16,4));
  DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(2,OutputWidthLength));
when 32 to 47 =>
  DataOut(OutputWidth-1 downto OutputWidth-4) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-32,4));
  DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(3,OutputWidthLength));
when 48 to 63 =>
  DataOut(OutputWidth-1 downto OutputWidth-4) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-48,4));
  DataLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(4,OutputWidthLength));
when 64 to 95 =>
  DataOut(OutputWidth-1 downto OutputWidth-5) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-64,5));
  DataLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(5,OutputWidthLength));
when 96 to 159 =>
  DataOut(OutputWidth-1 downto OutputWidth-6) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-96,6));
  DataLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(6,OutputWidthLength));
when 160 to 287 =>
  DataOut(OutputWidth-1 downto OutputWidth-7) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-160,7));
  DataLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(7,OutputWidthLength));
when 288 to 543 =>
  DataOut(OutputWidth-1 downto OutputWidth-8) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-288,8));
  DataLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(8,OutputWidthLength));
when 544 to 1095 =>
  DataOut(OutputWidth-1 downto OutputWidth-9) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-544,9));
  DataLength <= std_logic_vector(to Unsigned(9,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(9,OutputWidthLength));
when 1056 to 2079 =>
  DataOut(OutputWidth-1 downto OutputWidth-10) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-1056,10));
  DataLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(10,OutputWidthLength));
when 2080 to 4127 =>
  DataOut(OutputWidth-1 downto OutputWidth-11) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-2080,11));
  DataLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(11,OutputWidthLength));
when 4128 to 8223 =>
  DataOut(OutputWidth-1 downto OutputWidth-12) <=
    std_logic_vector(to Unsigned(ABS_level_writeout-4128,12));
  DataLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(12,OutputWidthLength));
when 8224 to 16415 =>
  DataOut(OutputWidth-1 downto OutputWidth-13) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-8224,13));
  DataLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(13,OutputWidthLength));
when 16416 to 32767 =>
  DataOut(OutputWidth-1 downto OutputWidth-14) <=
    std_logic_vector(to_unsigned(ABS_level_writeout-16416,14));
  DataLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
  PrefixLength <= std_logic_vector(to_unsigned(14,OutputWidthLength));
when others =>
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;
    else
      DataLength <= std_logic_vector(to_unsigned(SIGN_index,OutputWidthLength));
    end if;
  end loop;
  BinarizeState <= write_finished;
  Finished <= '1';
  DataOut <= (others => '0');
  DataLength <= (others => '0');
end case;
end if;
end process;
end struct;
Appendix B

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;

library std;
use std.textio.all;

library work;
use work.CABAC_EncParameters.all;

entity CABAC_Enc is
port
(
Clk : in std_logic;
Input : in std_logic_vector(InputW-1 downto 0);
InputLen : in std_logic_vector(InputWLen-1 downto 0);
ctxIdx : in std_logic_vector(6 downto 0);
SliceQPY : in std_logic_vector(5 downto 0);
initType : in std_logic_vector(1 downto 0);
Resetn : in std_logic;
Start : in std_logic;
Output : out std_logic_vector(OutputW-1 downto 0);
OutputLen : out std_logic_vector(OutputWLen-1 downto 0);
BypassI : in std_logic;
BypassO : out std_logic;
TermI : in std_logic;
TermO : out std_logic;
Finished : out std_logic
);
end CABAC_Enc;

architecture struct of CABAC_Enc is

-- Type declarations
--
-- States

type CABAC_EncStateType is(
  r_Input,
  init_ctxTbl,
  r_ctx,
  enc_bin_r,
  enc_bin_b,
  RenormE,
  PutBit,
  w_ctx,
  enc_Term,
  w_finished
);

-- Table types

type transIdx_t is array(0 to 63) of std_logic_vector(5 downto 0);

type qRange_t is array(0 to 3) of std_logic_vector(7 downto 0);

type rangeTabLps_t is array(0 to 63) of qRange_t;

type ctxTblInit_t is array(0 to (52*3*ctxIdxRange)-1) of std_logic_vector(6 downto 0);

type ctxTbl_t is array(0 to ctxIdxRange-1) of std_logic_vector(6 downto 0);

-- Functions

function string_to_binary(inp: string) return std_logic_vector is
  variable temp: std_logic_vector(inp'length-1 downto 0) := (others => 'X');
  begin
    for i in inp'range loop
      case inp(i) is
        when '0' => temp(i-1) := '0';
        when '1' => temp(i-1) := '1';
        when others => temp(i-1) := 'X';
      end case;
    end loop;
    return temp;
  end function;  

impure function InitctxTbl (RomFileName : in string) return ctxTblInit_t is
  FILE romfile : text is in RomFileName;
  variable RomFileLine : line;
  variable rom : ctxTblInit_t;
  variable TestString : string(7 downto 1);
  begin
    for i in ctxTblInit_t'range loop
      readline(romfile, RomFileLine);
      read(RomFileLine, TestString);
      rom(i) := string_to_binary(TestString)(6 downto 0);
    end loop;
    return rom;
  end function;

impure function InittransIdx (RomFileName : in string) return transIdx_t is
  FILE romfile : text is in RomFileName;
  variable RomFileLine : line;
  variable rom : transIdx_t;
  variable TestString : string(8 downto 1);
  begin
    for i in 0 to 63 loop
      readline(romfile, RomFileLine);
      read(RomFileLine, TestString);
      rom(i) := string_to_binary(TestString)(5 downto 0);
    end loop;
    return rom;
  end function;

impure function InitrangeTabLps (RomFileName : in string) return rangeTabLps_t is
  FILE romfile : text is in RomFileName;
  variable RomFileLine : line;
  variable rom : rangeTabLps_t;
  variable TestString : string(8 downto 1);
  begin
    for i in 0 to 3 loop
      for j in 0 to 63 loop
        readline(romfile, RomFileLine);
        read(RomFileLine, TestString);
        rom(j)(i) := string_to_binary(TestString)(7 downto 0);
      end loop;
    end loop;
    return rom;
  end function;

-- Signal declarations
-- Signal declarations

-----------------------------------------------------------------------------
signal CABAC_EncState : CABAC_EncStateType;
signal currCtx : std_logic_vector(6 downto 0);
signal initTbl : integer range 0 to ctxIdxRange-1;

-- Tables
signal rangeTabLPS : rangeTabLps_t := InitrangeTabLps("CABAC_Enc_Tables\rangeTabLPS.txt");
signal transIdxLPS : transIdx_t := InittransIdx ("CABAC_Enc_Tables\transIdxLPS.txt");
signal transIdxMPS : transIdx_t := InittransIdx ("CABAC_Enc_Tables\transIdxMPS.txt");
signal ctxIdxTableInitials : ctxTblInit_t := InitctxTbl ("CABAC_Enc_Tables\ctxIdxTableInitials.txt");

begin

---- ------------------------------------------------------------------------
---- Debugger process
---- ------------------------------------------------------------------------
--Debugger : process(Resetn)
--begin
--if rising_edge(Resetn) then
--for i in 0 to 63 loop
--report "index: " & integer'image(i) & " rangeTabLPS: " & integer'image(conv_integer(rangeTabLPS(i)(0))) & " " & integer'image(conv_integer(rangeTabLPS(i)(1))) & " " & integer'image(conv_integer(rangeTabLPS(i)(2))) & " transIdxLPS: " & integer'image(conv_integer(transIdxLPS(i))) & " transIdxMPS: " & integer'image(conv_integer(transIdxMPS(i)))
--end loop;
--for i in 0 to 1739 loop
--report "index: " & integer'image(i) & " ctxIdxTableInitials: " & integer'image(conv_integer(ctxIdxTableInitials(i)))
--end if;
--end process;

-- ctxIdxTable interfacing process

-- ctxIdxTableLookups : process(Clk)
--begin
--if rising_edge(Clk) then
--case CABAC_EncState is
--when init_ctxTbl => -- TODO verify full table is loaded
--report "index: " & integer'image(i) & " ctxIdxTable (initTbl)" & " ctxIdxTableInitials((conv_integer(SliceQPY)*(ctxIdxRange+3))" & "+(conv_integer(initType)+ctxIdxRange)+initTbl);" & " store current context in working register
--when r_ctx =>
--report "index: " & integer'image(i) & " update context from working register" & " ctxIdxTable(conv_integer(ctxIdx)) <= currCtx;" & " when others =>
--end case;
--end if;
end process;

-- CABAC_Enc coding main process

CABAC_Enc : process(Clk, Input, initType, SliceQPY, BypassI, Resetn, Start, ctxIdx, TermI)
begin

-- Variable declarations

-- Encoding vals
variable ivlLow : std_logic_vector(10 downto 0);--unsigned(10 downto 0);
variable ivlCurrRange : std_logic_vector(8 downto 0);--unsigned(8 downto 0);
variable ivlLpsRange : std_logic_vector(7 downto 0);
variable qRangeIdx : std_logic_VECTOR(0 to 1);

-- binVals
variable bins : std_logic_vector(InputW-1 downto 0);
variable binValI : integer range 0 to InputW-1;
variable binsLen : std_logic_vector(InputWLen-1 downto 0);

-- PutBit variables
variable PutBitVal : std_logic;
variable PutBitI : integer range OutputW-1 downto 0;
variable bitsOutstanding : integer range 0 to OutputW-1;
variable firstBitFlag : std_logic;

variable Flushed : std_logic_vector(1 downto 0);
variable InitFlag : std_logic;

begin
if Resetn = '0' then
  currCtx <= (others => '0');
  Output <= (others => '0');
  OutputLen <= (others => '0');
  CABAC_EncState <= r_Input;
  Finished <= '1';
  BypassO <= '0';
  InitFlag := '0';
  Flushed := "00";
  initTbl <= 0;
  ivlLow := (others => '0');
  ivlCurrRange := "111111110";
  firstBitFlag := '1';
  bitsOutstanding := 0;
  qRangeIdx := (others => '0');
  PutBitI := 0;
elsif rising_edge(Clk) then
  case CABAC_EncState is
  when r_Input =>
    if (Start = '1') then
      if (InitFlag = '0') then
        CABAC_EncState <= init_ctxTbl;
      elsif (TermI = '1') then
        CABAC_EncState <= enc_Term;
      else
        if (BypassI = '1') then
          CABAC_EncState <= enc_bin_b;
        else
          CABAC_EncState <= r_ctx;
        end if;
      end if;
    end if;
  when init_ctxTbl =>
    if (initTbl = (ctxIdxRange-1)) then
      if(BypassI = '1') then
        CABAC_EncState <= enc_bin_b;
      else
        CABAC_EncState <= r_ctx;
      end if;
    end if;
  when r_ctx =>
    if (binValI>=InputW) then
      CABAC_EncState <= enc_bin_r;
      currCtx <= ctxIdxTable(conv_integer(ctxIdx));
    when enc_bin_b =>
      if (binValI>=InputW) then
        end if;
  end case;
end if;
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;
elseif (ivlLow<512) then
  PutBitVal := '0';
  CABAC_EncState <= PutBit;
else
  ivlLow := ivlLow - 512;
  bitsOutstanding := bitsOutstanding + 1;
end if;
binValI := binValI - 1;
end if;

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 0);
  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;
  elseif (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;
  elseif (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
Output((OutputW-1)-PutBitI) <= PutBitVal;
PutBitI := PutBitI + 1;
end if;

PutBit_loop : for i in 0 to PutBitLoopLen-1 loop -- TODO: Potential overflow here if bitsOutstanding > PutBitLoopLen
  if (bitsOutstanding > 0) then
    Output((OutputW-1)-PutBitI) <= not PutBitVal;
    bitsOutstanding := bitsOutstanding - 1;
    PutBitI := PutBitI + 1;
  else
    if(Flushed = "10") then
      Output((OutputW-1)-PutBitI downto (OutputW-1)-PutBitI-1) <= ivlLow(8) & "1";
      OutputLen <= std_logic_vector(to_unsigned(PutBitI+2,OutputWLen));
      CABAC_EncState <= w_finished;
      elsif(BypassI = '1') then
        CABAC_EncState <= enc_bin_b;
      else
        ivlCurrRange := ivlCurrRange(7 downto 0) & "0";
        ivlLow := ivlLow(9 downto 0) & "0";
        CABAC_EncState <= RenormE;
      end if;
      exit PutBit_loop;
    end if;
    end loop;
  when w_ctx =>
    CABAC_EncState <= r_Input;
    PutBitI := 0;
    Finished <= '1';
  when enc_Term =>
    ivlCurrRange := ivlCurrRange - 2;
    if (bins(binValI) /= '0') then
      ivlLow := ivlLow + ivlCurrRange;
      ivlCurrRange := "000000010"; --2
      Flushed := "01";
      CABAC_EncState <= RenormE;
    else
      Flushed := "11";
      CABAC_EncState <= RenormE;
    end if;
  when others =>
    CABAC_EncState <= w_finished;
    Finished <= '1';
    TermO <= '1';
end case;
end if;
end process;
end struct;
Appendix C

// Asynchronous fifo

// 08.05.17
// Norwegian University of Science and Technology
// Lars Erik Songe Paulsen

// No "almost full" or "almost empty" signaling logic implemented

// ************************************************************************************
// TODO LIST:
// ************************************************************************************

// 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

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 XilinxSimpleDualPortClockBlockRamExample.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

// data_out must only be sampled when data_out_valid is asserted
assign data_out = Buffer[BufferReadAddress];

// MSB used for checking fifo full condition
// Remainder is actual Buffer address
reg [ADDRESS_WIDTH:0] ExtendedBufferWriteAddress, ExtendedBufferReadAddress;

// Used for addressing memory
wire [ADDRESS_WIDTH-1:0] BufferWriteAddress, BufferReadAddress;

// Binary coded (ADDRESS_WIDTH) bit memory next address
wire [ADDRESS_WIDTH:0] WriteNextAddress, ReadNextAddress;

// Gray coded Pointers for generating full/empty signals
reg [ADDRESS_WIDTH:0] WriteGrayPointer, ReadGrayPointer;

// Gray coded Next Pointers for synchronizing across clock domains
wire [ADDRESS_WIDTH:0] WriteGrayNextPointer, ReadGrayNextPointer;

// Rows to signal fifo status
wire DataInFull, DataOutEmpty;

// Write side logic
// Check full condition
assign DataInFull = (WriteGrayNextPointer ==
          (~ReadGrayPointer2Write2[ADDRESS_WIDTH:ADDRESS_WIDTH-1],
           ReadGrayPointer2Write2[ADDRESS_WIDTH-2:0]));

// Remove MSB before memory indexing
assign BufferWriteAddress = ExtendedBufferWriteAddress[ADDRESS_WIDTH-1:0];

// Increase Write address if conditions are met
assign WriteNextAddress = ExtendedBufferWriteAddress +
          (data_in_valid & ~data_in_full);

// Binary to Gray code conversion
assign WriteGrayNextPointer = (WriteNextAddress>>1) ^ WriteNextAddress;

always @(posedge clock_in or negedge rst_in_n) begin
  if (!rst_in_n) begin
    data_in_full <= 0;
    ExtendedBufferWriteAddress <= 0;
    WriteGrayPointer <= 0;
    WriteGrayPointer2Read1 <= 0;
    WriteGrayPointer2Read2 <= 0;
  end else begin
    // Update data in full register
    data_in_full <= DataInFull;
    ExtendedBufferWriteAddress <= WriteNextAddress;
    // Update current Gray code write pointer
    WriteGrayPointer <= WriteGrayNextPointer;
    // Send previous Gray code write pointer to Read side logic
    WriteGrayPointer2Read1 <= WriteGrayPointer;
    WriteGrayPointer2Read2 <= WriteGrayPointer2Read1;
  end
end

// Read side logic
// Check empty condition
assign DataOutEmpty = (ReadGrayNextPointer==WriteGrayPointer2Read2);

// Remove MSB before memory indexing
assign BufferReadAddress = ExtendedBufferReadAddress[ADDRESS_WIDTH-1:0];

// Increase Read address if conditions are met
assign ReadNextAddress = ExtendedBufferReadAddress + (data_out_ack & data_out_valid);
// Binary to Gray code conversion
assign ReadGrayNextPointer = (ReadNextAddress>>1) ^ ReadNextAddress;

always @(posedge clock_out or negedge rst_out_n) begin
    if (!rst_out_n) begin
        data_out_valid <= 0;
        ExtendedBufferReadAddress <= 0;
        ReadGrayPointer <= 0;
        ReadGrayPointer2Write1 <= 0;
        ReadGrayPointer2Write2 <= 0;
    end
    else begin
        // Update data out valid register
        data_out_valid <= !DataOutEmpty;
        // Update Read address register
        ExtendedBufferReadAddress <= ReadNextAddress;
        // Update current Gray code readpointer
        ReadGrayPointer <= ReadGrayNextPointer;
        // Send previous Gray code readpointer to Write side logic
        ReadGrayPointer2Write1 <= ReadGrayPointer;
        ReadGrayPointer2Write2 <= ReadGrayPointer2Write1;
    end
end
endmodule
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
        end
    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
    end
end

end
data_in <= 1'b1;
end
else begin
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
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;st_out_n = 1'b0;
data_out_ack = 1'b0;
#5;st_out_n = 1'b1;
data_out_ack = 1'b1;
#220;
clock_out = 1'b1;
end
repeat (200) begin
if (data_out_valid) begin
if(clock_out) begin
if (data_out == COMPARE[out_index]) begin
$monitor("data match");
end
else begin
$monitor("data mismatch: out_index: %d: %d != %d", out_index, COMPARE[out_index], data_out);
end
end
else begin
out_index = out_index + 1;
end
end
end
$monitor("data mismatch: out_index: %d: %d != %d", out_index, COMPARE[out_index], data_out);
end
end
clock_out = ~clock_out;
end

repeat (120) begin
#25;
if (data_out_valid) begin
  if(clock_out) begin
    if (data_out == COMPARE[out_index]) begin
      $monitor("data match");
    end
    else begin
      $monitor("data mismatch: out_index: %d: %d != %d", out_index, COMPARE[out_index], data_out);
    end
  end
  else begin
    out_index = out_index + 1;
  end
end
clock_out = ~clock_out;
end

repeat (360) 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 mismatch: out_index: %d: %d != %d", out_index, COMPARE[out_index], data_out);
    end
  end
  else begin
    out_index = out_index + 1;
  end
end
clock_out = ~clock_out;
end
endmodule
Appendix D

```csharp
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);
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }

            return new List<bin>();
        }
    }
}
```
public uint read_bit(out bool bypass)
{
    try
    {
        while (S_E_index < S_E_.Count)
        {
            if (S_E_[S_E_index].currPos < S_E_[S_E_index].Bins.Count)
            {
                bypass = S_E_[S_E_index].bypass;
                return (S_E_[S_E_index].Bins[S_E_[S_E_index].currPos++] == '1') ? (uint)1 : 0;
            }
            else
            {
                S_E_index++;
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
    //MessageBox.Show("read bit called when finished");
    bypass = false;
    return 0;
}

public void ResetCodeVals()
{
    //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;
}

public void EncodeDecision(uint bin)
{
    qCodIRangeidx = (codIRange >> 6) & 3;
    CodIrangeLPS = rangeTabLPS[pStateIdxTable[offset], qCodIRangeidx];
    codIRange = codIRange - CodIrangeLPS;
    if (bin != MPSIdxTable[offset])
    {
        codILow = codILow + codIRange;
        codIRange = CodIrangeLPS;
        if (pStateIdxTable[offset] == 0)
        {
            MPSIdxTable[offset] = 1 - MPSIdxTable[offset];
        }
        pStateIdxTable[offset] = transIdxLPS[pStateIdxTable[offset]];
    }
    else
    {
        pStateIdxTable[offset] = transIdxMPS[pStateIdxTable[offset]];
    }
    RenormE();
}
public void EncodeBypass(uint bin)
{
    try
    {
        codILow = codILow << 1;
        if (bin != 0)
        {
            codILow = codILow + codIRange;
        }
        if (codILow >= 1024)
        {
            PutBit();
            codILow = codILow - 1024;
        }
        else if (codILow < 512)
        {
            PutBit(0);
        }
        else
        {
            codILow = codILow - 512;
            bitsOutstanding++;
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}

public void RenormE()
{
    while (codIRange < 256)
    {
        if (codILow < 256)
        {
            PutBit(0);
        }
        else if (codILow >= 512)
        {
            codILow = codILow - 512;
            PutBit(1);
        }
        else
        {
            codILow = codILow - 256;
            bitsOutstanding = bitsOutstanding + 1;
        }
        codIRange = codIRange << 1;
        codILow = codILow << 1;
    }
}

public void PutBit(uint B)
{
    if (B != 0 && B != 1) { MessageBox.Show("ERROR: PutBit called with argument: " + B.ToString() + "\n Only 0 or 1 is valid arguments"); }
    if (firstBitFlag)
    {
        firstBitFlag = false;
    }
    else
    {
        bin tempBin = new bin();
        tempBin.val = (B == 1) ? '1' : '0';
        Encoded.Add(tempBin);    }
    while (bitsOutstanding > 0)
    {
        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 workaround 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;
}
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 bits Outstanding;
        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());
            }
        }
    }
}
public string Decode(List<bin> bins)
{
    Decoded = new char[65535];
    Decodedindex = 0;
    bins_ = bins;
    bins_index = 0;
    syntax_element_index = 0;
    ResetCodeVals();
    try
    {
        return new string(Decoded).Substring(0, Decodedindex);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
    return "empty";
}

public uint read_bit()
{
    try
    {
        while (bins_index < bins_.Count)
        {
            return (bins_[bin_index++] == '1') ? (uint)1 : 0;
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
    MessageBox.Show("read bit called when finished");
    return 0;
}

private void DecodeDecision()
{
    try
    {
        qCodIRangeidx = (codIRange >> 6) & 3;
        codIRangeLPS = rangeTabLPS[pStateIdxTable[offset], qCodIRangeidx];
        codIRange = codIRange - codIRangeLPS;
        if (codIOffset >= codIRange)
        {
            Decoded[Decodedindex++] = (MPSIdxTable[offset] == 1) ? '0' : '1';
            codIOffset = codIOffset - codIRange;
            codIRange = codIRangeLPS;
            if (pStateIdxTable[offset] == 0)
            {
                MPSIdxTable[offset] = 1 - MPSIdxTable[offset];
            }
            pStateIdxTable[offset] = transIdxMPS[pStateIdxTable[offset]];
        }
        else
        {
            Decoded[Decodedindex++] = ((MPSIdxTable[offset] == 1) ? '1' : '0');
            pStateIdxTable[offset] = transIdxMPS[pStateIdxTable[offset]];
        }
        RenormD();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
private void DecodeBypass()
{
    codIOffset = codIOffset << 1;
    codIOffset = codIOffset | read_bit();
    if (codIOffset >= codIRange)
    {
        Decoded[Decodedindex++] = '1';
        codIOffset = codIOffset - codIRange;
    }
    else
    {
        Decoded[Decodedindex++] = '0';
    }
}

private void RenormD()
{
    try
    {
        while (codIRange < 256)
        {
            codIRange = codIRange << 1;
            codIOffset = codIOffset << 1;
            codIOffset = codIOffset | read_bit();
        }
        if (codIOffset >= codIRange)
        {
            MessageBox.Show("Decoding error:\n The bitstream shall not contain data that result in
" +
                "codIOffset being greater than or equal to codIRange upon completion of
" +
                "this process.");
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}
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:

![Design Sources files](image)

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

![Table Sources files](image)

Then add the following files to simulation sources:

![Simulation Sources files](image)

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

![Simulation Sources files](image)
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 TextFile : TEXT open read_mode is "C:\Users\username\Desktop\CABACFiles\HEVC_CABAC_Verification_Tool\EncoderInput.txt";
-- Include EncoderOutput.txt in Vivado simulation sources.
file TextFileOut : TEXT open write_mode is "C:\Users\username\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 Button.
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.

```csharp
// 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 immediately followed by every initvalue
    // from initType 1, and finally then every initvalue from initType2.
    StreamWriter input = initializeReadFile("initinput.txt");

    // Output file containing Binary(1/0 ASCII chars) formatted 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) formatted output
                Output.WriteLine(String.Format("{0:X}", valMPS) +
                                  Convert.ToString(pStateIdx, 2).PadLeft(6, '0'));
            }
        }
    }
    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;
}
```

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;
}

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;
}
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.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Prefix bins</th>
<th>Suffix bins</th>
<th>Prefix length</th>
<th>Suffix length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>45</td>
<td>8</td>
<td>11</td>
<td>11110</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>15</td>
<td>11110</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>23</td>
<td>11111110</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>39</td>
<td>11111110</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>71</td>
<td>1111111000100</td>
<td>8</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>118</td>
<td>111111110</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>136</td>
<td>263</td>
<td>1111111110</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>264</td>
<td>519</td>
<td>11111111110</td>
<td>11</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>520</td>
<td>1031</td>
<td>111111111110</td>
<td>12</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1032</td>
<td>2055</td>
<td>1111111111110</td>
<td>13</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2056</td>
<td>4103</td>
<td>11111111111110</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4104</td>
<td>8199</td>
<td>111111111111110</td>
<td>15</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>8200</td>
<td>16391</td>
<td>1111111111111110</td>
<td>16</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16392</td>
<td>32775</td>
<td>11111111111111110</td>
<td>17</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>32776</td>
<td>65543</td>
<td>111111111111111110</td>
<td>18</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>65544</td>
<td>131079</td>
<td>1111111111111111110</td>
<td>19</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>131080</td>
<td>262151</td>
<td>11111111111111111110</td>
<td>20</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>262152</td>
<td>524305</td>
<td>111111111111111111110</td>
<td>21</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>524306</td>
<td>1048583</td>
<td>1111111111111111111110</td>
<td>22</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

The interactive excel sheet (CoeffAbsLevelRemaining.xlsx) 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.