EFFICIENT SOFTWARE AND HARDWARE IMPLEMENTATIONS OF THE H.264 ENTROPY ENCODERS

Thesis
Submitted to
The School of Engineering of the UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for
The Degree of
Master of Science in Electrical Engineering

By

Marc P. Hoffman, B.S.

UNIVERSITY OF DAYTON
Dayton, Ohio
December, 2011
EFFICIENT SOFTWARE AND HARDWARE

IMPLEMENTATIONS OF THE H.264 ENTROPY ENCODERS

Name: Hoffman, Marc Philip

APPROVED BY:

Eric Balster, Ph.D.
Advisory Committee Chairman
Assistant Professor
Electrical & Computer Engineering

Frank Scarpino, Ph.D.
Committee Member
Professor Emeritus
Electrical & Computer Engineering

Vijayan Asari, Ph.D.
Committee Member
Professor
Electrical & Computer Engineering

John Weber, Ph.D.
Associate Dean
School of Engineering

Tony Saliba, Ph.D.
Dean, School of Engineering
& Wilke Distinguished Professor
ABSTRACT

EFFICIENT SOFTWARE AND HARDWARE IMPLEMENTATIONS OF THE H.264 ENTROPY ENCODERS

Name: Hoffman, Marc Philip
University of Dayton

Advisor: Dr. Eric Balster

Part 10 of MPEG-4 describes the Advanced Video Coding (AVC) method widely known as H.264. H.264 is the product of a collaborative effort known as the Joint Video Team (JVT). The final draft of the standard was completed in May of 2003, and since then H.264 has become one of the most commonly used formats for compression of high definition video [9]. The entire H.264/AVC encoder is inherently a sequential process, which typically lends itself to a software solution. Within the H.264 Standard, two entropy decoders are discussed. These two lossless encoding methods are known as Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC). CAVLC offers the most basic solution, while CABAC provides increased compression rates at a cost in algorithm complexity. For fast encoding of H.264 bit streams, three solutions are presented in this thesis. Two implementations of CAVLC are discussed, including a software and a hardware solution. Finally, a simple implementation of CABAC is proposed.
ACKNOWLEDGMENTS

I would like to thank everyone who constantly encouraged me during my time as a student at UD as a Graduate Student. They helped make the experience so much more rewarding, especially the following people:

- **Dr. Eric Balster**: For being my advisor and laying out the foundation for my future career.

- **Dr. Frank Scarpino**: For teaching me what it means to be an Engineer and showing me the rewards of hard work.

- **Kerry Hill, Al Scarpelli, and the Air Force Research Laboratory at Wright-Patterson Air Force Base**: For enabling this experience.

- **Family and Friends**: For encouraging me and always being there when I needed support.
TABLE OF CONTENTS

ABSTRACT ................................................................. iii

ACKNOWLEDGMENTS ...................................................... iv

LIST OF FIGURES ........................................................ vii

LIST OF TABLES .......................................................... ix

CHAPTER:

1. INTRODUCTION ......................................................... 1
   1.1 H.264 Overview ................................................... 2
   1.2 Prediction .......................................................... 4
       1.2.1 Intra Prediction ............................................. 4
       1.2.2 Inter Prediction .............................................. 6
   1.3 Transform and Quantization ..................................... 7
       1.3.1 Hadamard Transform ....................................... 9
       1.3.2 Quantization ................................................ 10
   1.4 Entropy Coding .................................................. 11
       1.4.1 Exp-Golomb Coding ....................................... 13
       1.4.2 CAVLC ....................................................... 14
       1.4.3 CABAC ....................................................... 16
   1.5 Motivation and Organization ................................... 17

2. CAVLC IN SOFTWARE ................................................... 18
   2.1 CAVLC Parsing .................................................. 18
   2.2 Extract Block and Zig Zag Scan ................................. 19
       2.2.1 Calculate nC ............................................... 22
   2.3 Trailing Ones .................................................... 23
   2.4 Coefficient Token ............................................... 23
   2.5 Levels ............................................................ 23
       2.5.1 The Level Prefix .......................................... 24
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>H.264 Encoder Block Diagram</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Inter and Intra Prediction Methods</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Spatial Prediction Data Usage</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>16x16 Intra Prediction Methods</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>4x4 Intra Prediction Methods</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Extracting DC Coefficients</td>
<td>9</td>
</tr>
<tr>
<td>1.7</td>
<td>H.264 Syntax</td>
<td>12</td>
</tr>
<tr>
<td>1.8</td>
<td>CAVLC Block Diagram</td>
<td>14</td>
</tr>
<tr>
<td>1.9</td>
<td>CABAC Block Diagram</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>CAVLC in Software Block Diagram</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Luma 16x16 Extraction Order</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Zig Zag Scan</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Non-Zero Vector Creation Loop</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Raster Order Example</td>
<td>22</td>
</tr>
<tr>
<td>2.6</td>
<td>Level Encoding Block Diagram</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>Run before pseudo-code</td>
<td>30</td>
</tr>
<tr>
<td>2.8</td>
<td>Method Speed Comparisons - Foreman QCIF</td>
<td>31</td>
</tr>
<tr>
<td>2.9</td>
<td>Method Speed Comparisons - Flower CIF</td>
<td>31</td>
</tr>
<tr>
<td>2.10</td>
<td>Method Speed Comparisons - Park Run 720p</td>
<td>32</td>
</tr>
<tr>
<td>Table</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Exp-Golomb codeNum Ranges</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Level VLC LUT</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Coefficient Pairs</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Test Video Specifications</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Cycles per Macroblock</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Comparisons to High Performance Designs</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Set of Unary Binarized Codewords</td>
<td>46</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

In May of 2003 [9], the final draft for the first version of the H.264 Advance Video Coding (AVC) method was completed. This method offered several improvements over its predecessor H.263, mainly the ability to achieve higher compression rates without reducing quality. H.264 has since been updated several times to handle the ever increasing needs of the digital video market, including High Definition and 3D Stereoscopic data [1]. Since its inception, H.264 has become one the most widely used video compression techniques adopted into the digital world [9].

One of best features of H.264 is its flexibility, coming with several tools for reducing redundant video data. This allows H.264 to vary from being highly complex to a rather simple algorithm, depending on the quality requirements. In addition to these tools, the H.264 standard defines 17 sets of capabilities, or profiles, that allow for this flexibility [9]. Each profile is targeted for a specific application, ranging from high quality 3D stereoscopic video compression to a relatively low quality 2D video streaming.

Even with all this flexibility, the basic H.264 encoding method remains consistent at its core. The block diagram shown in Figure 1.1 is typical for all the H.264 profiles. The subsequent sections provides an overview of H.264 and it’s method for data reduction and compression.
1.1 H.264 Overview

H.264 compresses video data in three stages, prediction, transform and quantization, and finally entropy encoding. The first stage is the prediction stage. For every 16x16 block of data, called a macroblock, H.264 forms a 16x16 prediction block. These prediction blocks are created using previously encoded data. The previously encoded data can either be from previous “reference” frames, called Inter Prediction, or data from the current frame, which is called Intra Prediction. After the prediction for the current macroblock is created, it is subtracted from the raw data to form a 16x16 block of residual data. The idea behind this process is that residual data contains less information to encode than the original raw data. The prediction stage is seen in the H.264 block diagram shown in Figure 1.1.

![Figure 1.1: H.264 Encoder Block Diagram.](image)

The next step H.264 takes to compress the data is the transform and quantization stage. Figure 1.1 shows this stage as blocks T and Q. This process is performed on 4x4 blocks of data, which forces each macroblock to be divided further into sixteen 4x4 blocks of data. The transformation is similar to a 4x4 Discrete Cosine Transform (DCT), except for it uses integers instead of cosines [9]. This allows for the process
to be completed using only addition and bit shifting, a concept that is very desirable for hardware design. The block transformation converts the data into a frequency representation, in which the low frequency information is stored near the top left corner of a macroblock, while the high frequency information is stored in the bottom right corner of each block.

After the data is transformed, it is quantized. Quantization is the main source of loss and compression gain. Each 4x4 block of data is scaled based on the quantization parameter (QP). This reduces the magnitude and dynamic range of the coefficients in each 4x4 block data, resulting in better compression rates when using an entropy encoder, which is the final stage of the H.264 encoder.

H.264 encoders must also perform an inverse transformation and quantization so that they can reuse the data for prediction. This is shown as $T^1$ and $Q^1$ in Figure 1.1. In this way, the encoder simulates the work of an H.264 decoder. The inverse transformed blocks are used to create a residual macroblock, which is then added to its original prediction to recreate the current frame. This is stored in a buffer so that the encoder can use the “decoded” frame for inter and intra prediction.

The third and final stage of the H.264 encoding process is the entropy encoder. The H.264 standard specifies two entropy encoding methods, Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC). While CABAC offers increased compression performance, it is much more complex and is not a requirement of all H.264 decoders. CAVLC, on the other hand, is required of all H.264 decoders and is used in the most simplest profile in H.264, the Constrained Baseline Profile (CBP).
1.2 Prediction

The first step that H.264 takes for video compression is through the spatial and temporal prediction of frame data. Video files tend to contain high amounts of redundant data which is carried over from frame to frame. Another characteristic of video files (and stand alone images) is spatial correlation, specifically that neighboring pixels tend to be very similar in value. H.264 exploits these two attributes of video signals through its two prediction methods, intra prediction and inter prediction. Both methods requiring dividing up a frame into 16x16 blocks of data called macroblocks. H.264 attempts to predict the pixel content of each macroblock in a frame in raster order. These macroblocks will be encoded using either spatial methods (intra) or temporal methods (inter). Figure 1.2 represents the two different prediction methods. If predicting using intra prediction, H.264 will attempt to recreate the current macroblock using data from previously encoded macroblocks in the current frame. Since frames are encoded in raster order, previously encoded data will either be from above the current macroblock, or from the left. If using inter prediction, H.264 will create the prediction using pixel data from previously encoded frames. This data is often spatially near the current macroblock.

1.2.1 Intra Prediction

Intra prediction is based on the idea that neighboring pixels tend to be highly correlated. When encoding a frame using intra prediction, H.264 attempts to use pixels from adjacent macroblocks to form the prediction of the current macroblock. As shown in Figure 1.3, the current macroblock X will use data from macroblocks A,B,C and D if the data is available.
There are four methods for predicting a 16x16 block of data, shown in Figure 1.4. These methods include vertical, horizontal, DC (mean) and planar. Once a 16x16 block of predicted data is formed, it subtracted from the raw data to form a block of residual data. Outputs of the intra prediction process include the method used to form the predicted macroblock, and the residual data.

If the prediction method does not meet the user’s specification in terms of quality, a macroblock can be subdivided into sixteen 4x4 blocks of data. There
are nine prediction methods for encoding 4x4 blocks, shown in Figure 1.5. The 4x4 methods include horizontal, vertical and DC, in addition to 6 planar methods. These additional techniques allow for better compression but increase the computational complexity of the intra prediction process.

1.2.2 Inter Prediction

Inter prediction will use data from previously encoded frames to predict the pixel data in the current macroblock. Like intra prediction, inter prediction is not limited to 16x16 blocks. If using a 16x16 block of data from previous frames does produce the desired results, H.264 can divide a macroblock further into some combination of 16x8, 8x16, 8x8, 4x8, 8x4, and 4x4 blocks of data, called submacroblocks. Once the desired reference data is found, H.264 stores the difference between the spatial location of the current macroblock (or submacroblock) and the predicted block from the reference frame. These stored coordinate differences are known as motion
vectors. Outputs from the inter prediction process include motion vectors, along with the residual data that is obtained by subtracting the predicted data from the reference data.

1.3 Transform and Quantization

H.264 uses different transforms depending on the type of residual data. Every 4x4 block of data in a residual macroblock uses the Integer Transform that is based on the Discrete Cosine Transform (DCT). If the residual data is intra predicted using 16x16 blocks, an additional transform, called the Hadamard Transform is performed on the DC coefficients of the current macroblock. After the data is transformed it is quantized to reduce the dynamic range of the data. It is this transformed and quantized residual data that is encoded.

Discrete Cosine Transform (DCT) based Integer Transform

To ease the memory requirement of H.264, the Integer Transform based on the DCT was developed such that there would be a zero mismatch between the forward
and inverse transforms [9]. Also, because it is an integer transform, there is no loss of decoding accuracy due to rounding. The 4x4 DCT that the integer transform is given by

\[ Y = AXA^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ -a & -a & a & a \\ c & -b & b & -c \end{bmatrix} \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ -a & -c & -a & b \\ a & -b & a & -c \end{bmatrix} \begin{bmatrix} X \end{bmatrix} \]

(1.1)

\[ a = \frac{1}{2}, \quad b = \sqrt{\frac{1}{2} \cos \left(\frac{\pi}{8}\right)}, \quad c = \sqrt{\frac{1}{2} \cos \left(\frac{3\pi}{8}\right)} \]

The DCT shown in 1.1 can be factorized to form the H.264 Integer Transform given below.

\[ Y = CXC^T \otimes E = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ \frac{ab}{2} & \frac{ab}{2} & \frac{a^2}{2} & \frac{ab}{2} \\ \frac{ab}{2} & \frac{ab}{2} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix} \]

(1.2)

The multiplication performed at the end of the Integer Transform is absorbed into the quantization process. Thus the core portion of the Integer Transform can be performed using merely addition, subtraction and shifts. This is the same for the inverse transformation process, shown as 1.3.

\[ Y^+ = C_i^T (Y \otimes E_i)C_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -\frac{1}{2} & 2 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -\frac{1}{2} & 1 & -\frac{1}{2} \end{bmatrix} \]

(1.3)
1.3.1 Hadamard Transform

The Hadamard Transform is performed on the DC coefficients of a macroblock only if the current residual data was obtained using a 16x16 intra prediction method or if the data represents chrominance [1]. If working in 4:2:0 format, chroma macroblocks are only 8x8 in size. The Hadamard Transform is performed on the DC coefficients of a macroblock, which are extracted and stored as a separate 4x4 block of data if working with a 16x16 luma macroblock, as shown in Figure 1.6. In the case of chroma data, a 2x2 Hadamard Transform is performed on the 2x2 block of chroma DC coefficients.

The 4x4 block of DC coefficients ($X_{DC}$) are then transformed using Equation 1.4. If working with a 2x2 block of data, the coefficients are transformed using Equation 1.5. The inverse transform does not divide by two when working with 4x4 blocks, but is otherwise identical for both cases. The Hadamard Transform provides
additional data reduction in cases where prediction methods create an abundance of DC data.

\[
Y_{DC} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & -1 & -1 \\
1 & -1 & 1 & 1 \\
1 & -1 & -1 & -1 \\
\end{bmatrix}
\begin{bmatrix} X_{DC} \end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & -1 & -1 \\
1 & -1 & 1 & 1 \\
1 & -1 & -1 & -1 \\
\end{bmatrix}
\quad (1.4)
\]

\[
Y_{DC} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}
\begin{bmatrix} X_{DC} \end{bmatrix}
\begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}
\quad (1.5)
\]

### 1.3.2 Quantization

The quantization process performs both the scaling required by the Integer Transform and the actual data quantization. H.264 uses a scalar quantizer, which is also integer based [6]. The basic forward quantization process is shown in Equation 1.6.

\[
Z_{ij} = \text{round} \left( \frac{Y_{ij}}{Q_{step}} \right)
\quad (1.6)
\]

The H.264 standard supports up to 52 values of the quantization step \(Q_{step}\) sizes, which are derived directly from the Quality Parameter (QP). \(Q_{step}\) is determined by

\[
Q_{step}(QP) = Q_{step}(QP \% 6) \times 2^{\text{floor}(\frac{QP}{6})}
\quad (1.7)
\]

in which \% is the modulus operator. Every value of \(Q_{step}\) can be determined knowing the first five.

Since determining \(Q_{step}\) is a recursive process, H.264 utilizes a look up table to ease computational complexity. The actual values of \(Q_{step}\) are not in the standard,
rather a scalar matrix that has pre-calculated the quantization multipliers is specified as a function of QP. Both the forward and inverse quantization processes have their own unique look up tables, which are used to create the scalar matrices $S_F$ and $S_I$. The scalar matrices not only handles the quantization scalars, but also includes the scaling matrix shown in 1.2. The complete 4x4 forward transform is given by:

$$Q_F = \text{round} \left( (Y \otimes S_F) \cdot \frac{1}{2^{15 + \lfloor QP \rfloor}} \right)$$  \hspace{1cm} (1.8)

in which $Y$ is the transformed matrix. The inverse transform is similar, where $S_i$ integrates the de-quantizing in addition to the scalar matrix shown in 1.3.

$$Y_I = \text{round} \left( [C_I^T] \cdot [Q_F \otimes S_I \cdot 2^{\lfloor QP \rfloor}] \cdot [C_I] \cdot \frac{1}{2^6} \right)$$  \hspace{1cm} (1.9)

The data resulting from the forward quantization is sent to the entropy encoder, while the inverse quantized data is added to the prediction data and stored in the reference buffer.

### 1.4 Entropy Coding

The entropy coding of the portion of the H.264 encoder is where all the data needed to recreate the video file is converted to 1s and 0s. An H.264 video file is organized into syntax layers, shown below in Figure 1.7. For this conversion, there are three tools available. These include Exp-Golomb coding, Context-based Adaptive Variable Length Coding (CAVLC) and Context-based Adaptive Binary Arithmetic Coding (CABAC). In addition to some straight forward fixed-length decimal to binary conversions, H.264 primarily utilizes Exp-Golomb coding for generating the binary data seen in the Sequence Parameter Set (SPS), Picture Parameter Set (PPS) and Slice and Macroblock Header syntax [1]. The SPS contains
the data concerning the type of prediction methods used, the order the frames are encoded, and the size of frame slices. The PPS contains frame information, including frame size and the QP value used for quantization. To avoid the propagation of prediction errors, H.264 utilizes slices. Slices are portions of a frame, encoded separately from the other slices in the frame, so that errors in one slice will not be able to cross into the other. The number of slices in a frame is determined by the user, and is only limited by the number of macroblocks that appear in a frame. Macroblock data is contained within the slice syntax, and includes a header along with the residual pixel data. The header includes information such as the prediction method used to form the residual data, and the location of any all zero quadrants in the macroblock. The header data is either encoded using Exp-Golomb or CABAC, while the residual data is encoded using CAVLC or CABAC. These three primary methods, Exp-Golomb, CAVLC, and CABAC are discussed in Sections 1.4.1-1.4.3.

![Diagram of H.264 Syntax](image)

**Figure 1.7: H.264 Syntax**
1.4.1 Exp-Golomb Coding

Exp-Golomb coding is utilized for most of the syntax elements outside of residual pixel data. An Exp-Golomb codeword consists of a prefix of 0s followed by a stop bit '1' and a suffix containing a binary number related to the value being encoded, \textit{codeNum}. The length of the suffix is identical to the number of zeros found in the prefix. The \textit{codeNum} derived from the suffix is interpreted differently based on the type of Exp-Golomb code used. A table of Exp-Golomb codewords is shown in Table 1.1.

<table>
<thead>
<tr>
<th>Bit String</th>
<th>codeNum Range</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 1 (x_0)</td>
<td>1-2</td>
<td></td>
</tr>
<tr>
<td>0 0 1 (x_0 \ x_1)</td>
<td>3-6</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 (x_0 \ x_1 \ x_2)</td>
<td>7-14</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1 (x_0 \ x_1 \ x_2 \ x_3)</td>
<td>15-30</td>
<td></td>
</tr>
</tbody>
</table>

The variable \textit{codeNum} can either directly represent the syntax element to be encoded, or used in a mapping process defined in [1]. The reason multiple methods are needed is Exp-Golomb by itself cannot support signed values. Another reason is a syntax element may belong to a set of elements, and the number of elements in that set is less than the value of the syntax element. Exp-Golomb then maps that element to an index in the set, thereby reducing the length of the codeword and increasing the compression ratio.
1.4.2 CAVLC

One of the two methods for parsing residual data is known as CAVLC. The ability to decode CAVLC is required of all H.264 decoders. A block diagram of the CAVLC encoder is shown in Figure 1.8.

CAVLC takes every 4x4 and 2x2 block of transformed and quantized coefficients and reorders the data into a vector using the zig zag scan, shown in Figure 2.3. When encoding intra predicted and chroma macroblocks, H.264 encodes the DC coefficients (the top left most coefficient of a 4x4 block) of a macroblock separately from the rest of the coefficients.

The coefficient vectors are then analyzed and converted into a binary bit stream by the CAVLC encoder. CAVLC is optimized for encoding the transformed and quantized residual data. The binary bit streams contain 5 sections, shown in Figure 1.8 and listed below.
1. **Coefficient Token:** The coefficient token is a VLC that contains information concerning the total number of non-zero coefficients in the vector, in addition to the number of “Trailing Ones”. The token is determined using a LUT.

2. **Trailing Ones:** A string of positive and negative ones often occur at the end of the coefficient vector. Since they are so common, CAVLC makes a special case for up to three trailing ones. A single bit for each trailing one is used to encode the sign of that trailing one, (1 for negative, 0 for positive) in reverse zig zag scan order.

3. **Levels:** The value of each of the remaining non-zero coefficients in reverse order are found in the levels portion of the CAVLC bit stream. Each level can be encoded using one of 7 LUTs, or through stream processing. The length of each level increases as the absolute value of the coefficients increase.

4. **Total Zeros:** This portion of the bit stream contains information regarding the number of zeros that are embedded within the non-zero coefficients. A VLC is determined using an LUT and based on the number of non-zero coefficients and embedded zeros.

5. **Run Before:** For every non-zero coefficient (until all the embedded zeros have been accounted for) a few bits are appended to the bit stream that tell the decoder how many zeros “run before” the current coefficient.

CAVLC is optimized for encoding AC coefficients, but struggles to achieve good compression gain with DC coefficients. This is because DC coefficients tend to not vary as much, and it takes CAVLC a few passes before it is optimized for encoding.
coefficients with relatively higher magnitudes. CAVLC also struggles with lower QP values, which typically results in an increased number of non-zero values.

The benefits of CAVLC is such that it is the least complex of the two residual data coding methods, and is therefore encodes data faster. Also, at higher values of QP, the compression performance is similar to CABAC. Therefore CAVLC is sometimes preferred when speed is required, or if the solution is required to be small.

1.4.3 CABAC

CABAC provides an alternative to CAVLC, but is not required. CABAC offers improved compression performance, at dramatic increase in complexity. If an encoder chooses to utilize CABAC, all the syntax in the macroblock layer is written using the binary arithmetic coding method instead of Exp-Golomb and CAVLC. The basic block diagram of CABAC is shown below in Figure 1.9.

![CABAC Block Diagram](image)

Figure 1.9: CABAC Block Diagram

The first step of CABAC is binarization. Binarization assigns a binary code word to a syntax element. The method CABAC uses to generate the binary code word ranges from the simple use of a LUT to unary coding. All the syntax elements
in the macroblock layer are given unique binary codewords. After the binarization process is finished, CABAC chooses between the regular coding engine, or the bypass coding engine, solely on user preference. If the regular coding method is selected, CABAC determines the context of the syntax element. This context determines the probability that the binary symbol in the code word is equal to 1 or 0. Whenever the most probable symbol (MPS) is equal to the actual symbol, CABAC is able to avoid having to encode the value. However if the next binary value is not equal to the MPS, the context for that syntax is updated such that the probability for the least probable symbol (LPS) is increased. If this occurs in enough successive coding passes, the value of the MPS is swapped with the value of the LPS. In this way CABAC is able to achieve significant compression gain. A simpler form of CABAC is available in the form of the bypass coding engine. When using bypass coding, CABAC assumes that the probability of a 1 or 0 is equal, and encodes the data as such. Bypass coding ignores the context of a syntax element, CABAC still achieves relatively high compression ratios, while eliminating much of the complexity.

1.5 Motivation and Organization

This thesis is divided into four additional chapters. Chapter 2 discusses a software solution for performing CAVLC, Chapter 3 presents a high performance CAVLC architecture created using VHDL, Chapter 4 details a CABAC software solution, and compares the compression rates to CAVLC. Finally Chapter 5 provides details on possible future work and conclusions concerning the implementations and performance of the H.264 entropy encoders.
CHAPTER 2

CA VLC IN SOFTWARE

CA VLC is employed by H.264 for lossless encoding of predicted and quantized frames for compression gain. One of the most common methods for implementing CA VLC involves heavy reliance on look up tables (LUTs) to acquire a context adapted VLC [3, 8, 11]. However some solutions implement Arithmetic Table Elimination (ATE) methods to reduce the number of reads required in software [4, 7]. In this chapter, two simple realizations of CA VLC in software are investigated and implemented for verification and validation testing.

2.1 CA VLC Parsing

The CA VLC parsing process provides a fast and simple method that is optimized for encoding the transformed and quantized coefficients provided by H.264’s encoding process. In addition to the current macroblock, as shown in the simple block diagram of the CA VLC software encoder (Figure 2.1), CA VLC receives the variables $nA$ and $nB$, which are arrays detailing the number of non-zero coefficients contained in each 4x4 block that make up the neighboring macroblocks. These arrays are used to predict the number of non-zero coefficients in the current 4x4 block.
The output of CAVLC is inserted into the bit stream, starting with the Coefficient Token module output, and ending with the Run Before bits. Each of the modules that are shown in Figure 2.1 are discussed in more detail in Sections 2.2 - 2.7.

2.2 Extract Block and Zig Zag Scan

The Extract Block and Zig Zag Scan module determines which 4x4 block from the current macroblock is to be encoded next and then reorders that macroblock using a zig zag scan. The order in which sixteen 4x4 blocks are extracted from a luma 16x16 block are shown in Figure 2.2. Anytime the Hadamard Transform is used, CAVLC must encode the DC coefficients separately. In this case the DCs are encoded as one 4x4 block before the remaining sixteen 4x4 AC blocks are encoded. Chroma macroblocks, when encoding the standard 4:2:0 yuv file, are 8x8 in size, and therefore only contain four 4x4 blocks. Since all chroma blocks utilize the Hadamard transform, the DC coefficients are always extracted as a 2x2 block and encoded first.

As seen from Figure 2.2, the 4x4 blocks are extracted in raster order from each of the four quadrants that make up a macroblock. H.264 keeps track of whether or
not non zero coefficients exist in each quadrant. If there are no non zero coefficients in a given quadrant, CAVLC will not be called to encode that quadrant.

Every 4x4 and 2x2 block of data is reordered using the zig zag scan, shown in Figure 2.3. The purpose of the zig zag scan is to order the 4x4 block of data into a vector that is ordered from low frequency to high. This is because the magnitude of the low frequency coefficients tends to be greater than the magnitude of the high frequency coefficients. By encoding this vector in reverse order, CAVLC can minimize the number of required bits.

After the coefficient vector is created (coeffs), the software once again loops through the vectors to extract the non zero coefficients. These coefficients are placed in a second vector (nonZeros). Also during this loop, the number of non zero coefficients (γ) is calculated. The procedure for this task is shown as Figure 2.4. The original vector, the nonzero vector, and the total number of coefficients are the outputs from this portion of the CAVLC encoding algorithm.
\[ \gamma = 0 \]
\[ \text{for } k = 0 \rightarrow 15 \text{ do} \]
\[ \text{if } \text{coeffs}[k] \neq 0 \text{ then} \]
\[ \text{nonZeros}[\gamma] = \text{coeffs}[k] \]
\[ \gamma = \gamma + 1; \]
\[ \text{end if} \]
\[ \text{end for} \]

Figure 2.3: Zig Zag Scan

Figure 2.4: Non-Zero Vector Creation Loop
2.2.1 Calculate \( nC \)

Macroblocks are encoded in raster order, as shown in Figure 2.5. When encoding macroblocks, CAVLC needs the \( nC \) data from the neighboring macroblocks A and B shown in Figure 2.5, which are labeled \( nA \) and \( nB \) in Figure 2.1. Each macroblock contains sixteen \( nC \) values, one for each 4x4 block.

![Figure 2.5: Raster Order Example](image)

The variable \( nC \) represents the predicted number of non zero coefficients in the current macroblock. To calculate \( nC \), CAVLC must know the number of non zero coefficients both above (\( nB \)) and to the left (\( nA \)) of the current 4x4 block. This is why the \( nC \) data from the neighboring macroblocks is needed. If the 4x4 block in the current macroblock is bordering another macroblock, the \( nC \) from that macroblock will be used if it is available. An example of an unavailable 4x4 block is when the neighboring macroblock is from a different slice, or was skipped. Once all the neighboring \( nC \) values are determined, the value of \( nC \) is then calculated with Equation 2.1.
\[ nC = \begin{cases} \frac{nA+nB}{2} & \text{if A and B are available} \\ nA & \text{if only A is available} \\ nB & \text{if only B is available} \\ 0 & \text{otherwise} \end{cases} \] 

(2.1)

After CAVLC receives the coefficient vector and \( nC \) it is ready to parse the data.

CAVLC encodes each vector into the following five sections of data: coefficient token, trailing ones, levels, total zeros, and run before.

### 2.3 Trailing Ones

The Trailing Ones (T1s) section of the code stream represents the number of \( \pm 1 \)s that commonly appear at the end of a transformed and quantized coefficient vector. CAVLC encodes up to three T1s uniquely (0 for positive, 1 for negative). The software algorithm calculates the T1 codeword first because in order to calculate the Coefficient Token, the encoder needs to first know the number of T1s (\( \varphi \)).

### 2.4 Coefficient Token

The coefficient token is determined using a look up table (LUT) that is detailed in [1]. To determine which VLC to use from the LUT, CAVLC requires the variables \( nC, \gamma \) (the total number of coefficients in the current 4x4 block), and \( \varphi \). The \( nC \) value determines the column of the LUT that contains the coefficient token, and \( \varphi \) and \( \gamma \) determine the column.

### 2.5 Levels

CAVLC encodes the remaining non-zero coefficients, if any, that are found in every 4x4 and 2x2 block using the Levels module. The non-zero coefficient vector often starts with those coefficients that are highest in magnitude and the proceeding
coefficients tend to be smaller in magnitude. The CAVLC process reads this vector in reverse zigzag scan order, thereby starting with what is most often a coefficient that is relatively small in magnitude. The length of each codeword is adapted to the probability of the next coefficient being within a pre-defined range of magnitudes. Each level codeword contains a prefix and a suffix. The block diagram for how a level codeword is generated is shown in Figure 2.6.

![Figure 2.6: Level Encoding Block Diagram](image)

### 2.5.1 The Level Prefix

The prefix is made up of a string of zeros and followed by a stop bit ‘1’. When reading the prefix, H.264 decoders count the number of zeros and then assigns this value to $\zeta$. When operating under CBP, the variable $\zeta$ can never be greater than 15. The variable $\zeta$ is determined by how much of the binary form of the coefficient can be contained in the level suffix, described in Section 2.5.2. If the coefficient is above the predefined threshold, $\zeta$ is set to 15, or possibly 14 if the current coefficient is
the first non-zero coefficient. The decoder is aware of these special cases, described in Section 2.5.3 and adapts accordingly.

2.5.2 The Level Suffix

All of non-zero coefficients, except for possibly the first, are encoded with a suffix of magnitude $\chi$ and length $\lambda$. In case of the first non-zero coefficient, CAVLC does have a suffix and the prefix is Unary encoded. However if the magnitude of the first coefficient is above a certain threshold, or if there are more than 10 non-zero coefficients and less than 3 Trailing Ones, the first coefficient will have a suffix with $\lambda$ equal to 1.

For the rest of the non-zero coefficients, there is always a level suffix. The length $\lambda$ increases by 1 bit every time the current coefficient is above an adapted threshold, up to 6. The last bit in the suffix is the sign bit. Any remaining bits in the suffix will represent bits of the binary version of the current coefficient. If $\lambda$ is large enough, it is possible that it will contain all the bits (right aligned) required to encode the current coefficient. Otherwise, if the suffix does not contain enough bits, only the least significant bits (LSBs) are written in $\chi$. Also written in $\chi$ is the sign bit, which is appended to the end of the binary value. The remaining bits that cannot be expressed in the suffix represent $\zeta$ described in Section 2.5.1.

2.5.3 Special Cases

Sometimes the magnitude of the coefficient is too great to store within the constraints of the current $\lambda$. For example the maximum magnitude of a coefficient that can be written with $\lambda = 3$ is 60. If the current coefficient is too large, CAVLC sets $\zeta = 15$ and $\lambda = 12$. The value of the coefficient is stored entirely in $\chi$. Since there are eleven bits in the suffix that can store a coefficient’s value, and one sign
bit, the maximum value that CAVLC can encode, within the constraints of the CBP, is 2048.

2.5.4 Level Coding Using Look up Tables

After initializing \( \zeta \) and \( \chi \) to 0, the first step is to determine the initial \( \lambda \). If \( \gamma > 10 \) and \( \varphi < 3 \), \( \lambda \) is initialized to 1, otherwise it begins at 0. After all the variables are initialized, the encoder begins a loop that continues until all the non-zero coefficients, known as levels, have been accounted for. Each level’s magnitude minus one (or two), \( a \), is compared to an adaptive threshold \( T \), defined by:

\[
T = \begin{cases} 
7 & \text{if } \lambda = 0 \\
15 \times 2^{\lambda-1} & \text{otherwise}
\end{cases} \quad (2.2)
\]

If \( a \geq T \) then the codeword is determined arithmetically. The first step is derive the variable \( \text{lev} \_\text{cod} \) using Equation 2.3.

\[
\text{level} \_\text{cod} = \begin{cases} 
2 \times |\text{level}| - 1 & \text{if } \text{level} < 0 \\
2 \times |\text{level}| - 2 & \text{otherwise}
\end{cases} \quad (2.3)
\]

Depending on the current \( \lambda \), \( a \), and \( \text{lev} \_\text{cod} \), the values of \( \zeta \), \( \Lambda \) and \( \chi \) are set using equations 2.4, 2.5 and 2.6 respectively. Once the three variables \( \zeta \), \( \Lambda \) and \( \chi \) are set the level codeword can be determined.

\[
\zeta = \begin{cases} 
14 & \text{if } \lambda = 0, a < 16 \\
15 & \text{otherwise}
\end{cases} \quad (2.4)
\]

\[
\Lambda = \begin{cases} 
4 & \text{if } \lambda = 0, a < 16 \\
12 & \text{otherwise}
\end{cases} \quad (2.5)
\]

\[
\chi = \begin{cases} 
\text{level} \_\text{cod} - \zeta & \text{if } \lambda = 0, a < 16 \\
\text{level} \_\text{cod} - 15 - \zeta \times 2^{\lambda} & \text{otherwise}
\end{cases} \quad (2.6)
\]
Table 2.1: Level VLC LUT

<table>
<thead>
<tr>
<th>VLC0</th>
<th>VLC2</th>
<th>VLC4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>01</td>
<td>101</td>
<td>1001</td>
</tr>
<tr>
<td>001</td>
<td>110</td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td>00000000000000111110</td>
</tr>
<tr>
<td>000000000000000000000000000000100000</td>
<td>0000000000000000000000111111</td>
<td></td>
</tr>
<tr>
<td>VLC1</td>
<td>....</td>
<td>VLC3</td>
</tr>
<tr>
<td></td>
<td>0000000000000000000101</td>
<td>100000</td>
</tr>
<tr>
<td></td>
<td>0000000000000000000110</td>
<td>100001</td>
</tr>
<tr>
<td>010</td>
<td>0000000000000000000111</td>
<td>....</td>
</tr>
<tr>
<td>011</td>
<td>VLC5</td>
<td>000000000000000000011110</td>
</tr>
<tr>
<td>0010</td>
<td>1000</td>
<td>000000000000000000011111</td>
</tr>
<tr>
<td>0011</td>
<td>1001</td>
<td>VLC6</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td>00000000000000000010</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>00000000000000000011</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td>0000000000000000001110</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>0000000000000000001111</td>
</tr>
</tbody>
</table>

If \( a < T \), then the level codeword is set using one of 6 LUTs, parts of which are shown in Table 2.1. The single dimensional VLC table containing the current codeword is set equal to \( \lambda \). The index of the table is equal to the variable \( \text{lev.cod} \).

The final step is to determine the next \( \lambda \). If the current \( \lambda \) is equal to 0, then the variable \( \lambda \) is first set to 1. Next the current level’s magnitude is compared to another threshold

\[
T_{\text{inc}} = 3 \times 2^{\lambda-1} \tag{2.7}
\]

such that if the magnitude of the current level is greater than \( T_{\text{inc}} \), \( \lambda \) is incremented by one. The maximum value of \( \lambda \) is limited to 6.

2.5.5 Level Coding Using Arithmetic Table Elimination (ATE)

The H.264 standard defines a decoding algorithm for the level information in subclause 7.3.5.3.2 [1]. The proposed level encoder using ATE is the inverse of this process. This method removes the need for storing the 6 VLC LUTs shown in Table 2.1. Instead each level codeword is calculated algorithmically. Similar to the LUT
method, if \( a \geq T \), then \( \zeta, \Lambda, \) and \( \chi \) are calculated using equations 2.4, 2.5 and 2.6. If \( a < T \), then these variables are assigned using

\[
\begin{align*}
\zeta &= \text{floor} \left( \text{lev} \times 2^{\frac{1}{\lambda}} \right) \\
\Lambda &= \lambda \\
\chi &= \text{lev} \times \text{floor} \left( \zeta \times 2^\lambda \right)
\end{align*}
\]  

(2.8)

Once \( \zeta, \Lambda, \) and \( \chi \) have been defined, the level codewords are determined. As described in Section 2.5.1, the number of zeros that are written for the prefix is equal to the value stored in \( \zeta \). These zeros are followed by a 1. Then the value of \( \chi \) is converted to its binary equivalent of length \( \Lambda \) and appended to the bit stream.

### 2.6 Total Zeros

The Total Zeros section details the number of embedded zeros (\( \psi \)) within a coefficient vector. Embedded zeros represent all the zeros in a coefficient vector that appear before the last non-zero coefficient. The Total Zeros codeword is also encoded using a LUT. The column that the total zero code word lies in is determined simply by the total number of coefficients. There are only fifteen columns in this look up table, because fifteen is the highest \( \gamma \) possible with a minimum \( \psi \) of one.

To determine which column of the look up table to use, the algorithm must simply calculate \( \psi \). To do this, the algorithm loops through the coefficient vector in reverse order and searches for the first non zero value. Once this coefficient is found, the algorithm then begins incrementing \( \psi \) for each zero in the remaining indeces. The value of \( \psi \) is used as row index in the Total Zero LUT.

For DC chrominance vectors a separate LUT is used, but the method for calculating the row and column is the same. The \( \psi \) codewords listed in this table are optimized for the probability of the \( \psi \) when only 4 coefficients are present.
2.7 Run Before

The final section of the CAVLC bit stream contains the run before data. The Run Before section contains information on the location of the embedded zeros with the coefficient vector. CAVLC begins looping through each non zero coefficient in the vector and determines a codeword based on the number of zeros that appear before that coefficient, until all zeros have been accounted for. To determine which codeword to use, two variables are needed: $\beta$, which represents the number of zeros remaining in the current vector, and $\delta$, which represents the number of zeros that come before the current coefficient. These values are used to determine the row and column of the LUT that contains the codeword needed.

The algorithm for obtaining run before data is developed as an inverse to the algorithm for decoding run before data detailed in [1]. The algorithm loops through the coefficient vector in reverse order. For each non zero coefficient, the algorithm calculates how many zero coefficients appear before that coefficient. This value is represented by the variable $\delta$. Once another non zero is found, $\delta$ is subtracted from the variable $\beta$. The variable $\beta$ is initialized to $\psi$, and is continually updated until it is set to zero, or until all the coefficients have been scanned.

The run before codeword is also determined using a LUT, and each non zero coefficients codeword is determined using the variables $\delta$ and $\beta$ as the row and column respectively. The pseudo-algorithm for this process is shown as Figure 2.7.
for \( i = \text{firstNonZeroIndex} \) to \( \text{lastNonZeroIndex} \) do
\[
\delta \leftarrow 0 \\
\text{while } \text{coeffs}[i-1] = 0 \text{ do} \\
\quad \delta \leftarrow \delta + 1 \\
\quad i \leftarrow i - 1 \\
\quad \text{if } i = 0 \text{ then} \\
\qquad \text{End Loops} \\
\text{end if} \\
\text{end while} \\
\text{codeword} \leftarrow LUT[\delta][\beta] \\
\beta \leftarrow \beta - \delta \\
\text{if } \beta = 0 \text{ then} \\
\qquad \text{End Loop} \\
\text{end if} \\
\text{end for}
\]

Figure 2.7: Run before pseudo-code

2.7.1 Algorithm Comparisons

To compare the algorithms a timing analysis is performed that calculates the total amount of time the encoder spent converting the level information into binary bit streams. Only the level information is timed because this is the only case where the two algorithms are different. For each test, the GOP length is set to 10, and 50 frames are encoded with several different quality parameter (QP) values. The three video clips encoded include Foreman (QCIF), Flower (CIF), and Park Run (720p)[2]. Initial tests indicate that the LUT and ATE algorithms perform at a similar speeds. Figures 2.8, 2.9, and 2.10 shows the results.

2.8 Software Conclusions

In this chapter two software solutions were implemented and tested for performance in terms of speed. These algorithms are mostly identical except for the method in
Figure 2.8: Method Speed Comparisons - Foreman QCIF

Figure 2.9: Method Speed Comparisons - Flower CIF
which the Level portion of the CAVLC bit stream is encoded. From the tests performed on the software, it appears that both solution offer similar speeds. The main difference in the two algorithms is the complexity. In solutions where memory usage is not an issue, the LUT solution is easier to implement, and may be preferred, but in all other cases the ATE method is preferable.
CHAPTER 3

CAVLC IN HARDWARE

To increase the speed of the H.264 encoder, a high throughput architecture for CAVLC encoding is introduced in this chapter. Hardware design is complex in that instead of sequentially processing each line of code, such as in software, hardware processes each line of code in parallel. A hardware solution requires significant changes and additions to the software algorithm. However by implementing CAVLC in HDL, the rate at which H.264 performs increases significantly over the software solution proposed in Chapter 2, and allows for latest high definition video sequences to be encoded in real time.

3.1 Hardware Overview

The five sections that make up the CAVLC encoder, discussed in Section 1.1, can be performed in parallel with some pre-processing. This pre-processing is performed in the scanning phase of the CAVLC architecture, shown in Figure 3.1. The scanning phase is the most complex, as it performs all the necessary calculations needed to encode the bit stream at a constant rate. Following the scanning phase is the encoding phase, during which all the CAVLC codewords are created. The scanning and encoding phases are discussed in Sections 3.2 and 3.3 respectively.
These phases are also make up two of the states in the architecture’s state machine. The other two states are idle and waiting. The idle state is the default state, and its only purpose is to read in the coefficient vector and wait for the software to signal that the architecture can begin the scanning phase. Once the scanning phase is complete, the architecture signals that the data is ready for encoding. Once finished encoding, the system enters the final state, waiting. The waiting state makes no changes to any registered values, rather it is simply waiting for the architecture to reset before once again entering the idle state. Because of the cyclical nature of the architecture, this design can be pipelined to increase performance. Pipelining is discussed in Section 3.5. The state machine is visualized in Figure 3.2.
3.2 The Scanning Phase

Many CAVLC solutions divide the processing into scanning and encoding phases [4, 5, 7, 12]. The job of the scanning process is to analyze the coefficient vector to determine the characteristics of the vector needed to encode the residual coefficients. To reduce the number of clock cycles required to scan the vector, a modified method of the dual coefficient scanning operation in [7] is proposed here. The proposed scanning operation encounters 1 of the 14 unique pairings of coefficients that are shown in Table 3.1 during each of its 8 cycles. The data needed to encode each pair is saved so that the number of cycles required for the encoding phase is minimized.

Once every coefficient pair from the vector array has been scanned, the scanning phase is complete. The architecture then signals that all the registered values are ready for the encoding phase. All outputs of the scanning phase are latched while the encoding phase is active.
Table 3.1: Coefficient Pairs

<table>
<thead>
<tr>
<th>Output</th>
<th>Coefficient Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(0,0)</td>
</tr>
<tr>
<td>level</td>
<td>-</td>
</tr>
<tr>
<td>data</td>
<td>-</td>
</tr>
<tr>
<td>zero</td>
<td>store</td>
</tr>
<tr>
<td>data</td>
<td>zeros</td>
</tr>
<tr>
<td>t1_dat</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td>γ</td>
<td>-</td>
</tr>
<tr>
<td>ψ</td>
<td>+2</td>
</tr>
<tr>
<td>ϕ</td>
<td>-</td>
</tr>
</tbody>
</table>

3.3 The Encoding Phase

The Coefficient Token module, discussed in Section 2.4, reads in the 5 bit vectors $\gamma$ and $nC$, and the 2 bit vector $\varphi$. The coefficient token module takes 2 clock cycles to perform. On the first clock cycle, it calculates the row and column in the LUT that contains the token representing the current vector. On the second clock cycle it outputs the token.

The vector $t1.dat$ is determined during the scanning phase. The number of bits to use from $t1.dat$ is stored in the $\varphi$ output. Therefore no further processing is required for this section of the CAVLC bit stream.

The Encode Levels module reads in the data bus containing the $\lambda$ and level values for each non-zero coefficient. Having previously calculated these two values for each non-zero coefficient allows every level to be coded in parallel. To minimize resource usage, levels are encoded algorithmically. The codeword is calculated as $\chi$.

The Encode Levels modules also calculates each level codewords length ($code_{len}$) by

$$code_{len} = \zeta + \lambda + 1$$  \hspace{1cm} (3.1)
which is necessary for the software to create the bit stream. This is done for each
level found in the coefficient vector. Since all the coefficients are encoded in parallel,
the encode levels module can be completed in two clock cycles.

The final module, the Encode Zeros module, performs two tasks. First, it encodes
\( \psi \) described in Section 2.6. This is performed in 1 clock cycle. In addition, it encodes
the run before section of the CAVLC bit stream, which is described in Section 2.7.
The run before data buffer is an array of 16 logic vectors that are 8 bits in length.
The value of \( \delta \) for the current non-zero coefficient is stored in the last 4 bits of each
vector. The first 4 bits contain the total number of zeros (\( \kappa \)) found in the vector at
the time the current non-zero is scanned. Calculating the row and column of the
look up table that contains the run before codeword is done using Equation 3.2.

\[
\begin{align*}
\text{row} &= \psi - \kappa \\
\text{column} &= \delta
\end{align*}
\]  

(3.2)

The bit packer receives all the codewords generated in the Encoding Phase,
and when enabled, creates the bit stream. All of the outputs from the Encoding
Phase include a codeword and its length. Whenever the length is longer than the
codeword, zeros are inserted in the MSBs until the length is correct. Codewords are
placed in the order specified in [1] after the encoding phase is complete. Once all
the codewords are ready, the bits are appended to the H.264 File.

3.4 Implementation and Results

The architecture discussed in Section 3.1 is implemented targeting a Stratix
III FPGA, resulting in a maximum clock frequency of 188 MHz, and a Stratix IV
FPGA, resulting in a maximum clock frequency of 200 MHz. When testing three
separate raw video files are used. The data format of these video files, obtained from [2], are listed in Table 3.2.

Table 3.2: Test Video Specifications

<table>
<thead>
<tr>
<th>Video Sequence</th>
<th>Format</th>
<th>Frame Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreman.yuv</td>
<td>4:2:0</td>
<td>QCIF 176 x 144</td>
</tr>
<tr>
<td>Flower.yuv</td>
<td>4:2:0</td>
<td>CIF 288 x 352</td>
</tr>
<tr>
<td>Stockholm.yuv</td>
<td>4:2:0</td>
<td>720p 720 x 1280</td>
</tr>
</tbody>
</table>

To test the performance of the design, the average number of cycles needed to encode each macroblock are calculated for each video file. To calculate the average, 60 frames of a video file are encoded with QP varying from 4 to 50 as shown in Figure 3.3. Table 3.3 shows the cycles per macroblock of low QP values (5 and 10), since these QP values tend to result in the most data encoded. Also calculated for Table 3.3 is the worst case scenario, which is a constant 285 cycles. The worst case is constant because each phase in the architecture is constant.

Table 3.3: Cycles per Macroblock

<table>
<thead>
<tr>
<th>Video Sequence</th>
<th>QP Values</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Foreman</td>
<td>236</td>
<td>235</td>
</tr>
<tr>
<td>Flower</td>
<td>198</td>
<td>195</td>
</tr>
<tr>
<td>Stockholm</td>
<td>283</td>
<td>281</td>
</tr>
<tr>
<td>Average</td>
<td>238</td>
<td>234</td>
</tr>
</tbody>
</table>
The only reason the design would take less than 285 cycles to complete is if the coefficient block being encoded is a part of an 8x8 block (quadrant) of coefficients that are known to be all zero. In this event, no data is encoded for that quadrant. Figure 3.4 shows the number of skipped quadrants for each video sequence.

Figures 3.3 and 3.4 show a direct relation to the number of skipped macroblock quadrants and average number of cycles per macroblock. However at the worst case scenario, 285 cycles, this architecture is still capable of encoding 1080p video sequences at 60 frames per second (fps) with a minimum clock speed of 140 MHz. At the maximum 188 MHz, this design can encode 1080p videos at 81 fps.
3.5 Pipelining

The proposed design facilitates pipelining between the scanning and encoding phases. Figure 3.5 represents how pipelining the architecture could be performed. By scanning the next coefficient vector while the current vector is being encoded, the number of clock cycles are reduced significantly. In theory, a majority of the 4x4 blocks would only require 8 cycles to encode, reducing the worst case scenario for encoding a macroblock to 209 cycles. Considering the fact that the average cycles per macroblock without pipelining is about 82% of the worst case scenario, then the typical encoding rate of a pipelined design would theoretically be 170 macroblocks per second.

Pipelining the design results in a significant increase in throughput, but the change in combinatorial logic is unknown. Therefore the maximum clock speed is unknown. However if the average cycles needed to encode a macroblock each second is indeed 170, then the pipelined architecture would match the throughput
Figure 3.5: Pipeline Timing

of the standard architecture with a clock speed of 152 MHZ, which should be easily obtainable.

3.6 Comparisons

The architecture proposed by [4] does not perform parallel scanning, but does encode two coefficients in parallel. Both [5] and [7] offer solutions that do do parallel scanning, in addition to parallel encoding. Not one of these architectures perform the additional tasks required to encode the level and run before data in a constant amount of cycles, as does the proposed design. A comparison of encoder designs is shown in Table 3.4, from which we can see that the proposed design is faster on average than other high performance architectures. The average number of cycles needed to encode a macroblock is a very difficult value to determine, as the correlation between QP and cycles/MB is not linear. It is important to note that the average values here are calculated differently depending on the author. They
represent the results from different test sequences and broader ranges of QP values. Section 3.4 shows how large an impact encoding techniques have on the resulting macroblock encoding rate. To help better compare the results, the worst case rates are also shown in Table 3.4.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg.</td>
<td>298</td>
<td>244</td>
<td>254</td>
<td>238</td>
<td>238</td>
</tr>
<tr>
<td>Worst</td>
<td>414</td>
<td>280</td>
<td>323</td>
<td>285</td>
<td>285</td>
</tr>
<tr>
<td>Freq. (MHz)</td>
<td>125</td>
<td>180</td>
<td>200</td>
<td>188</td>
<td>200</td>
</tr>
<tr>
<td>MB/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg.</td>
<td>419K</td>
<td>73K</td>
<td>782K</td>
<td>789K</td>
<td>840K</td>
</tr>
<tr>
<td>Worst</td>
<td>302K</td>
<td>643K</td>
<td>619K</td>
<td>659K</td>
<td>702K</td>
</tr>
<tr>
<td>Logic (Gates)</td>
<td>9724</td>
<td>19K</td>
<td>66K</td>
<td>15K</td>
<td>12.5K</td>
</tr>
<tr>
<td>Memory (Kb)</td>
<td>39</td>
<td>48</td>
<td>189</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>Tech (um)</td>
<td>0.18</td>
<td>Virtex</td>
<td>3</td>
<td>Stratix</td>
<td>Stratix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>III</td>
<td>IV</td>
</tr>
</tbody>
</table>

### 3.7 Hardware Conclusions

In this chapter, a high performance H.264 CAVLC encoder is proposed. This design employs a modified parallel coefficient scanning technique that allows for the encoding of any level and run before data in just one clock cycle. With this new feature, the proposed CAVLC encoder has the highest throughput rate of any architecture found in literature, at a worst case 285 cycles per macroblock. The architecture meets the real-time processing demand by encoding 1080p at 86 fps, and allows for simple integration into any H.264 architecture.
CHAPTER 4

CABAC IN SOFTWARE

CABAC provides an alternative to the CAVLC parsing method. Although CABAC offers increased compression performance over CAVLC, it is not required of any H.264 encoding profiles. There are three steps for the regular CABAC coding scheme, and only two for the bypass coding scheme. These are shown in Figure 1.9. While Figure 1.9 provides a simple diagram of how CABAC is performed, a more complete flowchart is shown in Figure 4.1. Both diagrams contain the three primary steps taken by CABAC to transform syntax and residual data into 1’s and 0’s. These steps include binarization, updating the context model, and coding. However before any of these steps can be performed, CABAC must go through an initialization process. This initialization process is discussed in Section 4.1, while the algorithm for performing the three steps of CABAC encoding are described in Sections 4.2 to 4.4.

4.1 CABAC Initialization

Figure 4.1 shows that if a syntax element to be encoded is the first in the slice, initialization of the context model and encoding variables is required. The initialization process for the context model variables requires updating the $pStateIdx$ and $valMPS$ arrays. Each syntax element contains a set of their own $StateIdx$ and
Figure 4.1: CABAC Encoding Flow Chart
for $k = 0$ to 399 do
    $preCtxState = \text{Clip}(1, 126, ((m \ast \text{Clip}(0, 51, \text{SliceQP}_Y)) \gg 4) + n)$
    if $preCtxState \leq 63$ then
        $pStateIdx[k] \leftarrow 63 - preCtxState$
        $valMPS[k] \leftarrow 0$
    else
        $pStateIdx[k] \leftarrow preCtxState - 64$
        $valMPS[k] \leftarrow 1$
    end if
end for

Figure 4.2: CABAC Initialization Equation

$valMPS$ variables, of which there are currently 1024 unique sets of context models [1]. For the Main Profile, only 399 instances of $pStateIdx$ and $valMPS$ are required. These two variables are initialized for each instance as shown in Figure 4.2.

The initialization process also involves resetting the encoding engine, which requires initializing the variable $range$ to 510 and the variables $low$, $bitOutstanding$ and $bitsInNal$ to 0. These four variables are key to the binary arithmetic coding process and will be discussed in more detail in Section 4.4.

### 4.2 Binarization Algorithm

The binarization process involves converting a syntax element or residual data into a binary codeword. These binarization methods include Unary, Truncated Unary, Concatenated Unary/k-th order Exp-Golomb (UEGk), Fixed Length and VLC LUTs. Each syntax element is assigned one or two binarization methods. Many syntax elements are encoded as a prefix and suffix, each of which is binarized separately [1].
Table 4.1: Set of Unary Binarized Codewords

<table>
<thead>
<tr>
<th>synElVal</th>
<th>Bit String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 - - - -</td>
</tr>
<tr>
<td>1</td>
<td>1 0 - - -</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 - -</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1 0 -</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 1 0</td>
</tr>
</tbody>
</table>

| Binary Index | 0 | 1 | 2 | 3 | 4 |

### 4.2.1 Unary Binarization

The binary codeword created using the Unary binarization method consists of a string of 1s followed by a single 0 stop bit. The length of the string is determined by the value of the syntax element $synElVal$ to be encoded, where the length is equal to the syntax elements magnitude plus one. A table of Unary codewords is shown in Table 4.1.

### 4.2.2 Truncated Unary

The Truncated Unary (TU) binarization process is nearly identical to the Unary process. The difference being that the truncated method has a maximum length $cMax$ such that if $synElVal$ is equal to $cMax$, then the binarized form of the codeword will be a string of 1s that is $cMax$ in length. No accuracy is lost in the conversion, since $synElVal$ is never greater than $cMax$ [1].

### 4.2.3 Concatenated Unary/k-th Order Exp-Golomb (UEGk)

The UEGk method of binarization involves a concatenation of a prefix bit string to a suffix bit string. The prefix bit string is derived using a TU binarization. The
value that is binarized depends on the magnitude of \textit{synElVal} and \textit{uCoff}. Which ever value is smaller will be TU binarized. The value \textit{cMax} is set equal to \textit{uCoff}.

If \textit{uCoff} is less than \textit{synElVal}, or \textit{synElVal} is a signed value and not equal to 0, then there will be a suffix [1]. The suffix is determined using the equation in Figure 4.3. The length of the suffix is determined by the variable \textit{k}, which is initialized based on the type of syntax element.

\[
\text{if } (\text{Abs}(\text{synElVal}) \geq \text{uCoff}) \text{ then}
\]
\[
\text{suf} \leftarrow \text{Abs}(\text{synElVal}) - \text{uCoff}
\]
\[
\text{repeat}
\]
\[
\text{put}(1)
\]
\[
\text{sufS} = \text{sufS} - (1 << k)
\]
\[
k \leftarrow k + 1
\]
\[
\text{until } \text{sufS} \geq (1 << k)
\]
\[
\text{put}(0)
\]
\[
\text{while } k \neq 0 \text{ do}
\]
\[
\text{put}((\text{sufS} >> k) \& 1)
\]
\[
\text{end while}
\]
\[
\text{end if}
\]
\[
\text{if signedValFlag and synElVal }\neq 0 \text{ then}
\]
\[
\text{if synElVal} > 0 \text{ then}
\]
\[
\text{put}(0)
\]
\[
\text{else}
\]
\[
\text{put}(1)
\]
\[
\text{end if}
\]
\[
\text{end if}
\]

Figure 4.3: UEGk Suffix Equation

4.2.4 Fixed Length Binarization

The fixed length binarization method is the simple conversion of \textit{synElVal} to a fixed length binary number. The length of the binary number is determined by
$cMax$. In the main profile the only possible values of $cMax$ are 1, 3 and 15 [1]. These three values translate to fixed lengths of 1, 2 and 4 respectively.

### 4.2.5 Variable Length Code Look Up Tables

The final method for binarization is through VLC LUTs. When this method is called, a LUT is selected based on the syntax element that is being encoded. VLC LUTs are used when encoding the syntax element macroblock type, and differ depending on the slice type that the current macroblock belongs to.

### 4.3 Retrieving the Context Model

Every syntax element is assigned one or more context models, depending on the length of the binarization of the syntax element. CABAC encodes each bit from the binarization separately, and typically each bit will have its own context models. In some cases, bits will be associated with multiple context models. This typically occurs when the neighboring relevant syntax elements are available. This is similar to CAVLC’s attempts at predicting the value $nC$, only that CABAC attempts to predict many more of the syntax elements when compression gain is a result.

Each individual context model is represented by two variables, $valMPS$ and $pStateIdx$. The first variable, $valMPS$ represents the value of the most probable symbol. The symbol being a 1 or a 0. The variable $pStateIdx$ is a integer value between 0 and 63, and is representative of the probability of the current symbol being equal $valMPS$.

Determining which model to use is the most complex task when updating the context model. Not only is each syntax element represented by multiple context models, but there are several methods for determining which model to use.
To implement this module, a case statement is used that determines the index of the context model \((ctxIdx)\) based on the syntax element. In the main profile, there are 22 possible syntax elements \([1]\). There are two base equations used when deriving \(ctxIdx\). The offset variables \(ctxIdxOffset\) and \(ctxIdxBlockCatOffset\) are determined directly by the syntax element using LUTs. The increment variable \(ctxIdxInc\) is derived uniquely for each syntax element, and is often dependent on the context of the current syntax element, or the bit of the binarized form of \(synElVal\) that is being encoded.

\[
ctxIdx = ctxIdxOffset + ctxIdxInc \tag{4.1}
\]

\[
ctxIdx = ctxIdxOffset + ctxIdxInc + ctxIdxBlockCatOffset \tag{4.2}
\]

As an example, when calculating the \(ctxIdx\) of the luma portion of the syntax element coded block pattern (cbp), the binarized bits of luma cbp are mapped to each 8x8 quadrant in a macroblock. The context model is chosen based on the neighboring quadrants (A and B) cbp value. Figure 4.4 shows that if the cbp is equal to 0110, and the fourth bit is currently being encoded, then \(ctxIdxInc\) is set equal to 0.

Once \(ctxIdx\) is determined the context model associated with the current bit of the binarization of the current syntax element can be retrieved. Following determining the context model, the coding engine is enabled.

### 4.4 The Coding Engine

The coding engine updates the current context model, and when necessary, appends bits to the bit stream. The purpose of the coding engine is to efficiently
implement a binary arithmetic encoding (BAC) scheme that is unique to H.264. The
BAC method used by H.264 is a recursive procedure for coding interval subdivision
and selection, as shown in Figure 4.5 [10].

Unlike similar encoders, CABAC does not use a constant value for range. The
range is approximated to 1 in the similar QM and MQ coders to eliminate multiplication
[10]. CABAC avoids multiplication as well by using a LUTs that contain the
necessary pre-calculated values of $range_{MPS}$ and $range_{LPS}$. This limits the precision of the product, but it improves accuracy of the subdivision in addition to reducing complexity of H.264’s BAC [10]. The coding engine typically consists of two steps. The first is the encoding decision, and the second is the renormalization process.

### 4.4.1 Encoding Decisions

There are three methods for encoding a bit; regular, termination and bypass coding. The regular method is visualized in Figure 4.5. Whenever a bit is encoded using the regular method, its value is compared to $val_{MPS}$ associated with that bits context model. If they are equal, the range is set to the subdivision.

Once the ranges are determined, the current bit is compared to the $val_{MPS}$ assigned to that bits context. If they are equal, the range is updated to be equal to $range_{MPS}$, and the current $pStateIdx$ is updated such that the probability of the MPS occurring is increased. Otherwise if the current bit is not equal to $val_{MPS}$, the range is set equal to $range_{LPS}$ and the lower bound of the range $low$ is set equal to $low + range_{MPS}$. The current $pStateIdx$ is such that the probability of the bit not being equal to $val_{MPS}$ is more likely. If $pStateIdx$ is equal to zero, $val_{MPS}$ is switched such that the former LPS is now the MPS.

While each bit of a syntax element may make use of its own context model, the coding engine only has one instance of the variables $low$ and $range$. The algorithm for performing the regular coding method is presented as Figure 4.6.

For encoding the end of slice syntax element, or the LPCM macroblock type, the termination coding method is used. Both of these syntax elements use the context model associated with $ctxIdx$ equal to 276. The termination coding method does not compare the current bit to its associated $val_{MPS}$, rather it compares the current
rangeIdx ← range(7 → 6)
rangelPS ← rangeLUT[pStateIdx[ctxIdx]][rangeIdx]
rangeMPS ← range − rangelPS
if (bit = valMPS[ctxIdx]) then
    range ← rangeMPS
    pStateIdx[ctxIdx] = pStateLUT[pStateIdx[ctxIdx]][0]
else
    range ← rangelPS
    low ← low + rangeMPS
    if (pStateIdx[ctxIdx] = 0) then
        valMPS[ctxIdx] = 1 − valMPS[ctxIdx]
    end if
    pStateIdx[ctxIdx] ← pStateLUT[pStateIdx[ctxIdx]][1]
end if
Renorm(range, low, bitsOutstanding)

Figure 4.6: Regular BAC Equation

bit to 1. A 1 bit indicates that CABAC encoding is terminated and a flushing procedure is applied. In either case the renormalization procedure is applied. The equation for the termination coding method is shown in Figure 4.7

range ← range − 2
if (bit = 1) then
    low ← low + range
    range ← 2
    Renorm(range, low, bitsOutstanding)
    FlushCode(low)
else
    Renorm(range, low, bitsOutstanding)
end if

Figure 4.7: Termination BAC Equation
For some syntax elements CABAC makes use of the bypass encoding method. The bypass method is used when the syntax element or parts thereof is assumed to have a uniform probability distribution [1]. Since the probability model for a uniform distribution is not as complex, the bypass method has a simpler renormalization process. This is discussed in Section 4.4.2. The equation for the bypass coding method is shown in Figure 4.8.

\[
\text{low} \leftarrow \text{low} << 1 \\
\text{if } (\text{bit} = 1) \text{ then} \\
\quad \text{low} \leftarrow \text{low} + \text{range} \\
\text{end if} \\
\text{Renorm}_\text{bypass}(\text{range}, \text{low}, \text{bitsOutstanding})
\]

Figure 4.8: Bypass BAC Equation

4.4.2 Renormalization

The renormalization procedure occurs after one of the three BAC methods is complete. In [1], there are two methods for renormalization. The standard method is used for regular or termination coding, and maintains a minimum range size to prevent a loss of precision.

The equation for the regular renormalization procedure is shown in Figure 4.9. The procedure is an iterative process that continues until the value of range is greater than 25% of it’s maximum value (1024). At the end of each loop range and low are doubled. During the loop the most significant bit of low is appended to the bit stream.
while (range < 256) do
  if (low ≥ 256) then
    if low ≥ 512 then
      low ← low − 512
      PutBit(1)
    else
      low ← low − 256
      bitsOutstanding ← bitsOutstanding + 1
    end if
  else
    PutBit(0)
  end if
  range ← range << 1
  low ← low << 1
end while

Figure 4.9: Regular Renormalization

This renormalization process is further visualized in Figure 4.10. Here we see that if the current value of low lies between 256 and 512, bitOutstanding is incremented by 1. Otherwise, if low is less than 256, or greater than 512, a 0 or 1 respectively is output to the bit stream, followed by a string of 1s or 0s (respectively) that is \textit{bitOutstanding} in length. This process is performed by the \textit{PutBit} function.

The other renormalization process utilized by CABAC is performed after the bypass coding method. This process is similar to the one shown in Figure 4.9 with a few key differences. First, the process is always performed, and it is only performed once, rather than looped until certain conditions are met. Second the comparisons of low against 256 and 512 are replaced with comparisons to 512 to 1024, with similar results. The equation for bypass renormalization is shown in Figure 4.11.
if \( (\text{low} < 1024) \) then
if \( \text{low} < 512 \) then
\( \text{PutBit}(0) \)
else
\( \text{low} \leftarrow \text{low} - 512 \)
\( \text{bitsOutstanding} \leftarrow \text{bitsOutstanding} + 1 \)
end if
else
\( \text{PutBit}(1) \)
\( \text{low} \leftarrow \text{low} - 1024 \)
end if

Figure 4.11: Bypass Renormalization
4.5 CABAC vs. CAVLC

To test the compression gain when using CABAC instead of CAVLC, the same three video samples listed in Table 3.2 are encoded using both encoding methods. The compression rates of both CAVLC and CABAC at varying values of QP are compared in the Figures 4.5, 4.5 and 4.5.

![Figure 4.12: Foreman Compression Rates](image)

The improvement in compression that CABAC offers for these tests ranges from 17 percent to 3 percent. CABAC and CAVLC have similar performance when at low QP values. When utilizing low QP values, there is less quantization and therefore more residual data to encode. Using these results as a guide, it can be argued that CAVLC would be the better choice when encoding video sequences real time, since CABAC is much more computationally expensive, and does not offer significant improvement in compression rates. However if time is not a factor, and high QP values are used, then CABAC definitely offers the better compression performance.
Figure 4.13: Flower Compression Rates

Figure 4.14: Stockholm Compression Rates
CHAPTER 5

CONCLUSIONS

5.1 Conclusions

The entropy encoders of H.264 allow for a versatility not seen in previous video encoding methods. If a simple and fast platform is desired, then H.264 offers CAVLC. CAVLC offers good compression performance in addition to a high speed algorithm. CAVLC is very suitable for hardware design, allowing for the real time encoding of high definition video. If a user desires the best compression performance, then H.264 offers CABAC. CABAC is the latest binary arithmetic coding method to be adopted by mainstream media, offering up to a 17% increase in compression over CAVLC.

5.2 Future Work

To further increase the throughput of the CAVLC hardware architecture discussed in Chapter 2, pipelining will implemented. Implementing a pipelined encoder will allow this architecture to easily outpace rival high performance designs currently in literature. Another future project includes implementing CABAC in HDL. This would allow for a high throughput CAVLC architecture in addition to increased compression performance.
BIBLIOGRAPHY


