Hardware study on the H.264/AVC video stream parser

Michelle M. Brown

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Hardware Study on the H.264/AVC Video Stream Parser

by

Michelle M. Brown

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Kenneth Hsu,
Professor, RIT Department of Computer Engineering
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2008

Approved By:

Dr. Kenneth Hsu
Professor, RIT Department of Computer Engineering
Primary Adviser

Dr. Andreas Savakis
Professor, RIT Department of Computer Engineering

Dr. Dhireesha Kudithipudi
Assistant Professor, RIT Department of Computer Engineering
Dedication

I dedicate this thesis to my supportive family, who have always been there for me through the good, the bad, and when I needed them the most. I would especially like to dedicate this thesis to my mom, who provided me with the strength, courage, and determination to always strive for my goals, and my dad, who encouraged me to pursue a college degree and to study engineering.
I would like to thank my advisors for proving their guidance, knowledge, and time during the work for this thesis.
Abstract

The video standard H.264/AVC is the latest standard jointly developed in 2003 by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). It is an improvement over previous standards, such as MPEG-1 and MPEG-2, as it aims to be efficient for a wide range of applications and resolutions, including high definition broadcast television and video for mobile devices. Due to the standardization of the formatted bit stream and video decoder many more applications can take advantage of the abstraction this standard provides by implementing a desired video encoder and simply adhering to the bit stream constraints. The increase in application flexibility and variable resolution support results in the need for more sophisticated decoder implementations and hardware designs become a necessity.

It is desirable to consider architectures that focus on the first stage of the video decoding process, where all data and parameter information are recovered, to understand how influential the initial step is to the decoding process and how influential various targeting platforms can be. The focus of this thesis is to study the differences between targeting an original video stream parser architecture for a 65nm ASIC (Application Specific Integrated Circuit), as well as an FPGA (Field Programmable Gate Array). Previous works have concentrated on designing parts of the parser and using numerous platforms; however, the comparison of a single architecture targeting different platforms could lead to further insight into the video stream parser.

Overall, the ASIC implementations showed higher performance and lower area than the FPGA, with a 60% increase in performance and 6x decrease in area. The results also show the presented design to be a low power architecture, when compared to other research.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Thesis Objective</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Overview</td>
<td>3</td>
</tr>
<tr>
<td>2 Video Compression</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Compression Techniques</td>
<td>4</td>
</tr>
<tr>
<td>2.2 H.264/AVC Encoder and Decoder</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 Encoding</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2 Decoding</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Existing Hardware Implementations</td>
<td>10</td>
</tr>
<tr>
<td>3 H.264/AVC Video Stream Parser</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Reading NAL Units</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Parsing NAL Units</td>
<td>17</td>
</tr>
<tr>
<td>3.2.1 Basic Coding</td>
<td>18</td>
</tr>
<tr>
<td>3.2.2 Exponential-Golomb Coding</td>
<td>18</td>
</tr>
<tr>
<td>3.2.3 Context-Adaptive Variable Length Coding (CAVLC)</td>
<td>19</td>
</tr>
</tbody>
</table>
# Design and VHDL models

4.1 Reading NAL units ........................................ 22

4.2 Parsing NAL units ........................................ 22

4.2.1 Basic Decoding ........................................ 23

4.2.2 Exponential-Golomb Decoding ......................... 25

4.2.3 Context-Adaptive Variable Length Coding (CA VLC) .... 27

# Implementations and Testing

5.1 ASIC Implementation ...................................... 43

5.1.1 Sub-Design Comparisons ................................ 45

5.2 FPGA Implementation ...................................... 49

5.2.1 Sub-Design Comparisons ................................ 50

5.3 Synthesis Simulations ...................................... 52

5.4 ASIC and FPGA Comparisons .............................. 52

5.5 Comparisons with Existing Works ....................... 54

# Future Work

6 ................................................................. 56

# Conclusion

7 ................................................................. 58

Bibliography ..................................................... 59
List of Figures

1.1 Structure of the H.264/AVC Video Stream Parser .......................... 3
2.1 H.264 and MPEG-2 Comparison [3] .............................................. 6
2.2 Structure of an H.264/AVC encoder [8] ........................................... 6
2.3 Multi-Frame Motion Compensation [11] ......................................... 7
2.4 Integer Transform Matrix used in H.264/AVC [17] .......................... 7
2.5 CA VLC Decoder [18] ................................................................. 10
2.6 Low Power CA VLC Decoder Design [8] ...................................... 11
2.7 Low Cost High Performance CA VLC Decoder Design [6] .............. 12
2.8 Exp-Golomb Decoder [19] ............................................................. 13
3.1 Video Stream Parser ................................................................. 14
3.2 NAL Types [9] ................................................................. 16
3.3 Example of CA VLC Decoding Reverse Zig Zag Scan .................... 20
4.1 Architecture of the Video Stream Parser ....................................... 22
4.2 NAL Unit Format ................................................................. 23
4.3 NAL Unit Parser State Machine ................................................. 24
4.4 Hardware Design of the Exp-Golomb Decoder ............................... 25
4.5 Hardware Design of the First One Detector used by the Exp-Golomb Decoder ................................................. 26
4.6 Architecture of CA VLC Decoder ................................................. 27
4.7 Architecture for parse_coeff_token ............................................. 28
4.8 Neighboring Macroblocks of the Current Macroblock in Frame Mode 28
4.9 Hierarchical Organization of parse_coeff_token ............................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10</td>
<td>State Machine Used By All Components Utilizing the Integer Division</td>
<td>30</td>
</tr>
<tr>
<td>4.11</td>
<td>Neighboring Macroblocks of the Current Macroblock in Field Mode</td>
<td>31</td>
</tr>
<tr>
<td>4.12</td>
<td>mbAddrN Specification with MbaffFrameFlag equal to One [17]</td>
<td>33</td>
</tr>
<tr>
<td>4.13</td>
<td>Architecture of find_luma_neighbors</td>
<td>34</td>
</tr>
<tr>
<td>4.14</td>
<td>State Machine for find_luma_neighbors</td>
<td>35</td>
</tr>
<tr>
<td>4.15</td>
<td>State Machine Modeling the VLC look-up Tables</td>
<td>36</td>
</tr>
<tr>
<td>4.16</td>
<td>Architecture of trailing1s_level_wrapper</td>
<td>37</td>
</tr>
<tr>
<td>4.17</td>
<td>Architecture of totalZeros_runBefore_wrapper</td>
<td>39</td>
</tr>
<tr>
<td>4.18</td>
<td>State Machine Modeling the totalZeros_runBefore_wrapper design</td>
<td>40</td>
</tr>
<tr>
<td>4.19</td>
<td>Hardware Architecture for the Coefficients’ Array Indices</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>16x16 Luma Block (left) and 8x8 Cb or Cr Block (right)</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>The ASIC and FPGA Implementations’ Performance for Various Resolutions</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Basic Decoding ASIC</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Exponential Golomb Decoding ASIC</td>
<td>46</td>
</tr>
<tr>
<td>5.5</td>
<td>CAVLC Decoder ASIC</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>Power, Area, and Performance Comparison of CAVLC Sub-Designs</td>
<td>47</td>
</tr>
<tr>
<td>5.7</td>
<td>parse_coeff_token ASIC</td>
<td>48</td>
</tr>
<tr>
<td>5.8</td>
<td>trailing1s_level_wrapper ASIC</td>
<td>48</td>
</tr>
<tr>
<td>5.9</td>
<td>totalZeros_runBefore_wrapper ASIC</td>
<td>49</td>
</tr>
<tr>
<td>5.10</td>
<td>FPGA Resource Usage of Top Level Designs</td>
<td>50</td>
</tr>
<tr>
<td>5.11</td>
<td>FPGA and ASIC Gate Count Comparison</td>
<td>53</td>
</tr>
<tr>
<td>5.12</td>
<td>FPGA and ASIC Performance Comparison</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Tables

3.1 NAL Unit Format .......................................................... 16
3.2 Entropy Decoder Algorithms ........................................... 17
3.3 Mapping of Exp-Golomb codeNums and Codewords .............. 18
3.4 CAVLC Decoder Syntax Elements [8] ............................... 20

4.1 Neighboring Macroblock Address Calculations for Frame and Field Modes 31
4.2 mbAddrN Specification with MbaffFrameFlag equal to Zero [17] .... 32

5.1 Summary of ASIC and FPGA Performance for Various Resolutions .... 43
1. Introduction

1.1 Background

Video compression is the procedure through which the amount of data representing a video sequence is significantly reduced to allow for a decrease in transmission time and an increase in data storage by removing redundancies within the video. The need for video compression arose with the development of technology and can be traced back to 1964 when AT&T introduced the first Picturephone at the World’s Fair in New York [10]. However, a strong need came with the development of digital technology, such as standard-definition television (SDTV), which was first introduced in 1996. Compared to analog systems, SDTV offers more channels and produces better picture quality. SDTV displays at a resolution of 704 X 480 pixels at 30 frames per second, which requires uncompressed data to be transmitted at 121.7 Mbps. The MPEG-2 video compression standard is commonly used to handle the large amounts of video data the SDTV technology requires to be sent by compressing the bit rate to 3 Mbps. One of the major advancements in television after SDTV is high-definition television (HDTV), which allows viewers to watch television at even greater resolution. However, this implies more data to be transmitted during the same amount of time. HDTV can be viewed at a resolution of 1920 X 1080 pixels, which requires 746.5 Mbps of uncompressed data to be transmitted and about six times more data to be sent than SDTV. The MPEG-2 standard is only able to compress the HDTV bit rate to 32 Mbps for HDTV [4]. As a result, a more advanced algorithm, such as the one specified by the H.264/AVC standard, is desired to achieve an increase in transmission efficiency. With the many techniques the H.264/AVC standard specifies, transmitting the same image quality requires at least half the bit rate over MPEG-2.
1.2 Thesis Objective

The objective of this thesis is to explore the design of the decoder’s first step, the video stream parser, with a focus on providing an architecture to be targeted for two ASICs and an FPGA. While existing implementations have shown to be valuable, the results of targeting these designs for different platforms has yet to be studied. As a result of the thesis work presented here, when an H.264/AVC video parser design is targeted for two ASIC process technologies and an FPGA, insight is gained into the impact this decoder component has across various process technologies and platforms. Since the video parser is composed of numerous algorithms, the resulting architecture consists of original and leveraged designs, where many designs were implemented as a Finite State Machine (FSM) and another used different hardware components than seen in other published works.

The implementation satisfies the Basic H.264/AVC Profile, which can handle tasks such as entropy decoding, macroblock adaptation of frame and field modes, and parsing different slice types. The overall design is shown in Figure 1.1, where the first step is to read the compressed video stream and parse the Network Abstraction Layer (NAL) units. The NAL units, which are used to provide a layer of abstraction over the video data, are parsed individually and various decoding algorithms are invoked, including Basic, Exponential-Golomb (Exp-Golomb), and Context-Adaptive Variable-Length Coding (CA VLC) decoding, to recover all video data.

The VHDL hardware description language was used to describe an implementation of the video stream parser. An existing behavioral model of the H.264/AVC decoder assisted with the creation and validation of the synthesizable VHDL description. Implementing the design allowed for the analysis of the parser with respect to the timing and hardware complexities. The Xilinx ISE application was used to synthesize the FPGA model, while Synopsys’ Design Compiler performed the synthesis of the lower power ASIC model. ModelSim 6.1a SE was used to test the FPGA implementation, with the presumption that both models generate the same functionality.
1.3 Thesis Overview

An overview of video compression is given in Chapter 2, where various video compression techniques, and an overview of the H.264/AVC standard are explored. The details of the H.264/AVC video stream parser are presented in Chapter 3 and the design of the parser, which includes how the NAL units are created and parsed, is presented in Chapter 4. The three parsing methods, Basic, Exp-Golomb, and CAVLC decoding, are also explained in this chapter. The testing strategies and statistics on the two implementations are given in Chapter 5. Finally, the document is completed with a conclusion and future work suggestions in Chapter 6.
2. Video Compression

Video compression standards date back to the 1980s, when the first video codec (encoder/decoder) was standardized as the H.120 standard by the ITU-T. A decade later, the MPEG group made vast improvements on video and audio compression and created the MPEG-1 standard, which also defined the MP3 audio format and was used in Video CD. Two years later, in 1991, MPEG expanded on their previous standard by creating the MPEG-2, which specified the format of broadcast digital signals and stored digital video, and is currently used in DVD standards and SDTV systems. The MPEG-2 was a vast improvement on the MPEG-1 because of its expansion of format specification and support of interlaced video, which allows for the same video to be seen using half the bandwidth. Later, in 1998, MPEG created the MPEG-4 standard, which aimed at the compression of audio and video digital data and is currently used in many areas including web video, video telephone, and broadcast television. There are several standards defined by the MPEG-4, one of which is termed the ISO/IEC MPEG-4 Part 10 standard, or the ITU-T H.264 standard, which is a digital video codec that is an improvement over previous standards by flexibly providing high quality video data at various bit rates and has been around since 2003.

2.1 Compression Techniques

Different video compression techniques are available and can be classified into the following categories based on each technique’s goals: lossless, lossy, interframe, intraframe, object, and transform based.

Lossless compression uses various methods to compress video data without losing any
information and lossy compression discards some information to achieve a higher compression ratio. Interframe compression takes advantage of temporal redundancies by using the similarities between successive frames to reduce the amount of data required to represent the video sequence, while intraframe compresses each frame based solely on the current one. The object-based technique compresses data based on the detection of particular objects between frames. Finally, transform based compression transforms the video data from the spatial to frequency domain to exploit the human eye’s low sensitivity to high frequency change. This is different from object-based because the entire image is divided into blocks and the data is compressed independent of any objects within those blocks. Parts of the MPEG-4 standard utilizes the object based compression, while the MPEG-1, MPEG-2, and H.264/AVC standards use the transform based. The 2-D discrete cosine transform (DCT), discrete wavelet transform (DWT), and the integer transform are some methods used within transform based compression.

### 2.2 H.264/AVC Encoder and Decoder

The main goal of the H.264/AVC standard is to be efficient for a wide range of applications and resolutions, while achieving lower bit rates than previous standards (see Figure 2.1). The increase in application flexibility and compression ratio is enabled by the introduction of many new features, such as context-adaptive variable length coding (CAVLC), weighted prediction, multiple reference picture motion compensation, and an in-the-loop deblocking filter.

#### 2.2.1 Encoding

The H.264/AVC standard can be further explored by analyzing the encoding process presented in Fig. 2.2. The process encompasses four steps (motion estimation, transform, quantization, and entropy coding), where each works on a 16x16 macroblock of video data.
Figure 2.1: H.264 and MPEG-2 Comparison [3]

Figure 2.2: Structure of an H.264/AVC encoder [8]

**Motion Estimation**

Motion estimation uses reference frames to detect change, or motion, between frames to allow for only the residual data to be encoded. The video frames are classified into three types: I, P, and B. An I-frame is encoded using intra-frame prediction, where macroblocks within the frame are referenced, and can be used as a reference picture for subsequent frames. A P-frame uses inter-prediction, where a previous frame is referenced to produce a prediction signal for each block within the frame. The B-frame also uses inter-prediction, but is able to reference two previous frames and take a weighted average of the two prediction signal values [17]. The accuracy of the motion representation has been improved to include quarter-samples, compared to half-sample accuracy in previous standards. Also, the allowance of motion vectors to breech picture boundaries has been added to the H.264/AVC specification. The inclusion of using previously decoded pictures as reference for motion compensation prediction can be seen in Figure 2.3.
Transform and Quantization

The H.264/AVC standard uses an integer transform algorithm on a 4x4 block, instead of using a 4x4 DCT as in previous standards (see Fig. 2.4). The transformed coefficients are then quantized, which is a lossy compression technique that divides the values and rounds them to the nearest integer. This allows for greater compression efficiency because most of the high frequency coefficients become zero [17].

\[
H = \begin{bmatrix}
1 & 1 & 1 & 1 \\
2 & 1 & -1 & -2 \\
1 & -1 & -1 & 1 \\
1 & -2 & 2 & -1
\end{bmatrix}
\]

Figure 2.4: Integer Transform Matrix used in H.264/AVC [17]

Entropy Coding

The output of the motion estimation, transformation, and quantization stages is sent through a decoding feedback loop, where the difference between the incoming uncompressed video and the processed data is recursively used in the encoding flow. The video data and settings used in the previous stages are sent to the entropy encoder to produce more efficient code.
lengths by utilizing Exp-Golomb codes for all syntax elements, except for the quantized transform coefficients, where CAVLC is used. The Exp-Golomb encoding scheme allows for the use of only one look-up table, instead of having a table for each syntax element. The CAVLC encoder is highly efficient and complicated; therefore, it is only used to encode the quantized transform coefficients. There are multiple look-up tables used to encode the various syntax elements associated with this scheme. More detailed information about the Exp-Golomb and CAVLC schemes can be found in Chapter 3 - H.264/AVC Video Stream Parser.

**Video Coding Layer and Network Abstraction Layer**

In the last stage, the Video Coding Layer (VCL) encoder provides a customizable representation of the video data and the Network Abstraction Layer (NAL) encoder adds headers and organizes the data into NAL units. Having an abstraction layer, consisting of the VCL and NAL layers, provides immense freedom for the application, while also adhering to the standard.

**2.2.2 Decoding**

The process of decompression can be viewed as undoing the actions performed during the compression process where similar techniques are used to recover the original video data. Within a video sequence, each picture is divided into macroblocks, which represent a fixed sized area of the picture. The fixed sizes of the macroblocks are 16x16 samples for the luma component and 8x8 samples for the two chroma components. The H.264/AVC standard separates the color representation of video into three components: Y,Cb,Cr, where Y (or luma) refers to the brightness, and the Cb and Cr (or chroma) components refer to the picture color with respect to blue and red. Since the human eye is more sensitive to change in brightness than color, the luma component is represented with four times the amount of samples than the chroma components. Equation 2.1 is used to calculate the Y, Cb, Cr values, where \( K_r = 0.2126 \) and \( K_b = 0.0722 \) [5].
A picture can also be represented as a frame, which embodies two interleaved fields. The top field is made up of all even rows in the frame and the bottom field contains the odd rows. Since moving objects often cause adjacent rows to be independent, compressing them separately can provide greater coding efficiency. Conversely, non-moving objects should be compressed in frame mode, since a dependency is likely to exist between adjacent rows. The H.264/AVC standard supports adaptive field/frame encoding on a pair of macroblocks, which allows for greater efficiency when a frame contains both moving and non-moving areas [17].

The decoding process begins by parsing the incoming compressed stream and is performed by the video stream parser. The decoder receives the data in NAL units, which are packets that contain the encoded data, and are classified by the type of data they contain. These units are parsed by the entropy decoder and depending on the type of NAL unit, a specified entropy decoding algorithm is invoked. The three types of algorithms are based on basic coding, Exponential-Golomb coding, and context-adaptive coding. After the stream parser recovers all the parameter information and residual data, the inverse quantization and transform stages reconstruct the residual data. Then, based on the type of prediction used during encoding, the residual data is used to recreate the original frames. A side effect of operating on blocks within each frame is visually noticeable block edges throughout the frames. To smooth the edges of the blocking effect, H.264/AVC incorporates an in-loop deblocking filter, which adapts its filter strength based on previous syntax elements and parameter information.

\[
Y = K_R \ast R + (1 - K_R - K_B) \ast G + K_B \ast B; \quad Cb = \frac{1}{2} \left( \frac{B - Y}{1 - K_B} \right) \quad Cr = \frac{1}{2} \left( \frac{R - Y}{1 - K_R} \right)
\]

(2.1)
2.3 Existing Hardware Implementations

High performance architectures of the H.264/AVC standard have been a focus of research within many universities and in industry, but there has yet to be published studies focused on a single architecture targeting different platforms. An industry example is one that was a joint effort between Xilinx, Inc. and 4i2i Communication Ltd and is a main/high profile decoder IP core for an FPGA. None of the implementation details have been provided, which prevents others from learning how they accomplished the design; however, high level information has been given, such as the IP core targeting HD applications, the fully pipelined design with multiple configuration options, and an external SRAM memory needed to support the HD video [15].

There has also been some research performed on specific H.264/AVC decoder components, namely the Context-Adaptive Variable-Length Coding (CA VLC) and Exp-Golomb decoders. A proposed architecture, which focused on a generic VLSI architecture of the CA VLC decoder, can be viewed in Figure 2.5.

![Diagram of CA VLC Decoder](image)

Figure 2.5: CA VLC Decoder [18]
The proposed CAVLC design is a pipelined architecture, which is suitable for applications requiring high throughput decoding due to the one cycle recovery time of a single syntax element. The design consists of six components: the controller, input buffer, coeff_token Variable Length Code (VLC) decoder, level VLC decoder, total_zeros VLC decoder, and the run_before VLC decoder. Since the end of each VLC is not known until the previous VLC has been decoded, all actions occur sequentially. The input buffer aligns the input stream so it is possible to decode the next code word, and the coeff_token and level VLC decoders determine which VLC table to use based on neighboring block information. The total_zeros VLC decoder determines the number of zeros preceding the last non-zero level and the run_before VLC decoder determines the number of zeros preceding the last non-zero coefficient [18].

Another CAVLC decoder design is proposed for low power consumption and is targeted as an ASIC using the 0.18um CMOS standard cell-based library (see Fig. 2.6).

The design achieves its lower power consumption by employing various power saving techniques, which include prefix predecoding and table partitioning within most of the components. Another low power technique, which is used in the CAVLC design presented in this thesis, places latches in front of partitioned tables to disable non-used portions of
A low cost and high performance CAVLC decoder was proposed in [6], where various techniques were used to achieve the real-time processing requirement of 1080 HD video decoding (see Fig. 2.7).

To reach the high performance, this architecture contains a many more components as the previous two designs, which include a Flush-unit, parameter interface, prediction data R/W module, and Interleave Double Stacks (IDS). The Flush-unit flushes the previous codeword into the bit stream and aligns the next one. The controller assists in decreasing the computation time and lowering the power consumption by implementing the Zero Codeword Skip (ZCS), which does not decode zero codewords in 4x4 and 2x2 blocks that only contain zeros. Also, placing an enable signal on each component allows for power to be saved by disabling those which are not being used. Within the coeff_token component, hierarchical logic for look-up tables are used, which partitions the tables by frequency of appearance and helps the design achieve its high performance goal. The IDS component handles communication between the the CAVLC decoder and the inverse quantization [6].

A generic VLSI architecture for a Exp-Golomb decoder was proposed in [19] and a modified version is presented in this thesis work. The proposed architecture can be viewed in Figure 2.8.
Figure 2.8: Exp-Golomb Decoder [19]

The barrel shifter, "Shifter0," is used to align the input bit stream for the next decoding cycle and the "First 1 Detector" counts the number of leading zeros. The other barrel shifter, "Shifter1," is used to determine CodeNum + 1, which is used to determine the value of the recovered syntax element. This architecture only requires 3210 gates with a critical path delay of 5.83 ns [19].
3. H.264/AVC Video Stream Parser

A VHDL model of the video stream parsing process has been designed and successfully implemented on two different platforms, a low cost FPGA and a low power ASIC. The design was focused on the use of Finite State Machines and on the use of different hardware components than seen in other published works. Also, low power techniques were implemented to decrease the overall power consumption. Pipelining was not incorporated into the design because of the inherent sequential bit reading of the incoming stream.

There are two main steps in achieving the functionality of the video stream parser: reading NAL units and decoding the NAL units (see Fig. 3.1).

![Figure 3.1: Video Stream Parser](image-url)
The video parsing process consists of reading the compressed bit stream, creating the NAL units, and parsing the NAL units to recover picture information. The parsing process involves the use of three decoding components: Basic, Exp-Golomb, and CAVLC. The Basic and Exp-Golomb decoders are used throughout the video parsing scheme and return a single syntax element, which could represent many slice header, sequence parameter, or picture parameter values. The CAVLC decoder is a much more complex scheme and is used to parse the residual, zig-zag ordered blocks of transform coefficients of each frame to take advantage of the following quantized blocks’ characteristics:

1. Most non-zero coefficients tend to be toward the low frequency end of the zig-zag ordered list. As a result, VLC look-up tables are used to encode the level (magnitude).

2. Most of the values following the non-zero coefficients are (+/-) one; therefore, the amount and sign of the trailing ones are encoded.

3. Each string of zero is encoded using run-level encoding since most of the quantized blocks contain many zeros.

The top-level design consists of reading the data stream, iteratively parsing each unit, and storing the recovered information. Most of the received units contain sampled values of the video picture, while the units received at the start of the stream contain information that could be applied to multiple units. Once all of them have been parsed and the video parser completes, all the recovered data is sent to the next stage of the decoder.

### 3.1 Reading NAL Units

The creation of NAL units is the first step of the video stream parser. The compressed video stream is read and based on the sequence of bits received, NAL units are created. The video data and parameters are organized into units, which are categorized by the type
of data each one contains (see Fig. 3.2). The importance of the NAL is noticeable in its ability to be efficiently customizable for various transport systems.

<table>
<thead>
<tr>
<th>nal_unit_type</th>
<th>Content of NAL unit and RBSP syntax structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unspecified</td>
</tr>
<tr>
<td>1</td>
<td>Coded slice of a non-IDR picture</td>
</tr>
<tr>
<td>2</td>
<td>Coded slice data partition A</td>
</tr>
<tr>
<td>3</td>
<td>Coded slice data partition B</td>
</tr>
<tr>
<td>4</td>
<td>Coded slice data partition C</td>
</tr>
<tr>
<td>5</td>
<td>Coded slice of an IDR picture</td>
</tr>
<tr>
<td>6</td>
<td>Supplemental enhancement information (SEI)</td>
</tr>
<tr>
<td>7</td>
<td>Sequence parameter set</td>
</tr>
<tr>
<td>8</td>
<td>Picture parameter set</td>
</tr>
<tr>
<td>9</td>
<td>Access unit delimiter</td>
</tr>
<tr>
<td>10</td>
<td>End of sequence</td>
</tr>
<tr>
<td>11</td>
<td>End of stream</td>
</tr>
<tr>
<td>12</td>
<td>Filler data</td>
</tr>
<tr>
<td>13..23</td>
<td>Reserved</td>
</tr>
<tr>
<td>24..31</td>
<td>Unspecified</td>
</tr>
</tbody>
</table>

Figure 3.2: NAL Types [9]

<table>
<thead>
<tr>
<th>Header Byte</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forbidden zero bit</td>
<td>Raw Bit Sequence Payload (RBSP)</td>
</tr>
<tr>
<td>NAL reference ID</td>
<td></td>
</tr>
<tr>
<td>NAL unit type</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: NAL Unit Format

The start of each unit is signified by a header byte, which holds various information, as seen in Table 3.1. Following the header byte are payload bytes of the type specified in the header. For systems that require the delivery of units in a byte-stream format, a start code prefix is required to denote the beginning of each unit. Other systems, such as IP/RTP, which is a protocol for delivering audio and video over the Internet, require the delivery of them in packets; therefore, for these systems the use of the start code prefix is not necessary.

The payload can contain Video Coding Layer (VCL) or non-VCL data. The VCL NAL units are defined as those that contain the sampled video data. The non-VCL NAL units
contain parameter set information, which can be applied to multiple units and are values that are not expected to change frequently. These parameter sets can be classified into sequence parameter sets, which apply to a sequence of coded video pictures, or picture parameter sets, which apply to separate coded video pictures.

A sequence of units that define a coded picture is called an access unit. There can also be special NAL units that signify the beginning of an access unit, called an access unit delimiter, and the end of an access unit, called an end of sequence or end of stream NAL unit.

### 3.2 Parsing NAL Units

The task of the NAL parser is to analyze the incoming units to recover the video data, header, and parameters. Based on the type of unit received, certain decoding algorithms are invoked to recover the necessary syntax elements. The types of payloads that are encoded can be categorized into Basic, Exp-Golomb, and context-adaptive syntax elements (see Table 3.2).

<table>
<thead>
<tr>
<th>Coding Algorithm</th>
<th>Payload Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Byte</td>
</tr>
<tr>
<td>Basic</td>
<td>Fixed-pattern n-bit string</td>
</tr>
<tr>
<td>Basic</td>
<td>Signed n-bit integer</td>
</tr>
<tr>
<td>Basic</td>
<td>Unsigned n-bit integer</td>
</tr>
<tr>
<td>Exp-Golomb</td>
<td>Mapped Exp-Golomb-coded syntax element</td>
</tr>
<tr>
<td>Exp-Golomb</td>
<td>Truncated Exp-Golomb-coded syntax element</td>
</tr>
<tr>
<td>Exp-Golomb</td>
<td>Signed integer Exp-Golomb-coded syntax element</td>
</tr>
<tr>
<td>Exp-Golomb</td>
<td>Unsigned integer Exp-Golomb-coded syntax element</td>
</tr>
<tr>
<td>Context-Adaptive</td>
<td>Context-adaptive arithmetic entropy-coded syntax elements</td>
</tr>
<tr>
<td>Context-Adaptive</td>
<td>Context-adaptive variable-length entropy-coded syntax element</td>
</tr>
</tbody>
</table>

Table 3.2: Entropy Decoder Algorithms
3.2.1 Basic Coding

The Basic decoding technique involves direct interpretation of each syntax element as the type of element it was encoded as. For example, if an element was encoded as an unsigned integer, then it is decoded as an unsigned integer. This technique handles the interpretation of signed and unsigned integers, bytes, and fixed-pattern strings (see Table 3.2).

3.2.2 Exponential-Golomb Coding

The Exp-Golomb decoding algorithm is slightly more complex and uses a single code-word look-up table (VLC table). Variable length coding uses smaller code word lengths for frequently occurring data and larger codeword lengths for less frequently occurrences. As a result, the average codeword length is reduced and higher compression is achieved. Within the Exp-Golomb algorithm, the variable length codewords are defined as: [M zeros][1][INFO], where M denotes the number of leading zeros and INFO denotes an M-bit field of information. A codeNum value would have been mapped to its corresponding codeword during the encoding stage.

<table>
<thead>
<tr>
<th>codeNum</th>
<th>codeWord</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>00100</td>
</tr>
<tr>
<td>4</td>
<td>00101</td>
</tr>
<tr>
<td>5</td>
<td>00110</td>
</tr>
<tr>
<td>6</td>
<td>00111</td>
</tr>
<tr>
<td>7</td>
<td>0001000</td>
</tr>
<tr>
<td>8</td>
<td>0001001</td>
</tr>
<tr>
<td>9</td>
<td>0001010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.3: Mapping of Exp-Golomb codeNums and Codewords

Depending on the type of NAL unit received, one of the four Exp-Golomb decoding
algorithms might be used (see Table 3.2). Each decoding algorithm determines the code-
Num value by using the equation codeNum = codeWord - 1. Then, based on the codeNum
calculated and decoding algorithm used, a corresponding element value is provided. These
element values are used to define certain video parameters and are passed to the remainder
of the decoder for further processing.

3.2.3 Context-Adaptive Variable Length Coding (CA VLC)

Context-Adaptive Variable Length Coding (CA VLC) decoding is a type of run length de-
coding, where the number of zeros to be transmitted is reduced. As a result of the al-
gorithm’s increased complexity and efficiency, it is only used when quantized transform
coefficients are transmitted. During video compression, many video coefficients become
zero after the quantization step occurs, which is termed a run of zeros. Instead of encoding
each zero into the video compression stream, run length compression is used, where the
run length of the zeros is encoded to increase the overall compression efficiency. CA VLC
decoding also uses the probability of occurring symbols to further increase the compression
ratio.

The CA VLC decoding algorithm receives the quantized coefficients within a mac-
roblock in zig-zag order, starting at the top left of the block (see Fig. 3.3). The low fre-
quency values are located in the top left and tend to have larger values than those at higher
frequencies. These values become less dense as the bottom right corner of the block is
approached.

The next step requires the decoding of five syntax elements from the received coef-
ficients: coeff_token, sign of trailing ones (T1s), level, total_zeros, and run_before (see
Table 3.4).

The sign of the T1s and the level can be arithmetically decoded, while the other syntax
elements need to be decoded using look-up tables. There are two types of VLC tables
used: (1) for the number of non-zero coefficients and (2) for the level of the non-zero
coefficients. Since these values are correlated between neighboring blocks, the VLC table
Figure 3.3: Example of CAVLC Decoding Reverse Zig Zag Scan

<table>
<thead>
<tr>
<th>Syntax Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coeff_token</td>
<td>the number of all non-zero coefficients (total_coeff) and the number of trailing ones (T1s) are encoded by this syntax element.</td>
</tr>
<tr>
<td>Sign of T1s</td>
<td>the sign bit of each T1 is reverse zig-zag scan order is encoded by this syntax element.</td>
</tr>
<tr>
<td>Level</td>
<td>The value of each non-zero coefficient (except for T1s) is encoded by this syntax element.</td>
</tr>
<tr>
<td>total_zeros</td>
<td>The total number of zero coefficients preceding the last non-zero coefficients in zig-zag order is encoded by this syntax element.</td>
</tr>
<tr>
<td>run_before</td>
<td>The number of successive zero coefficients following the non-zero coefficients in reverse zig-zag order.</td>
</tr>
</tbody>
</table>

Table 3.4: CAVLC Decoder Syntax Elements [8]

Choice is based on the values obtained from these blocks. Once a compressed bit stream has been decoded, the pixel values of a 4x4 block can be recovered. When the final list of coefficients have been compiled, they are passed onto the transform unit for further processing.
4. Design and VHDL models

The video stream parser consists of three main functionalities: reading NAL units, parsing NAL units, and a memory component (see Fig. 4.1). During the reading process, test data is compiled into numerous NAL units and later parsed after all have been formed. The recovery of all the picture parameter and data information occurs within the parsing component, where the Basic, Exp-Golomb, and CAVLC decoders are utilized. The Exp-Golomb and Basic decoding models are used by many other components because of their ability to recover single syntax elements, which could represent the slice header, sequence, or picture data, and are used as parameters by the remainder of the H.264/AVC decoder. The majority of the video stream parser effort is located in recovering the slice data, which represents the actual picture information, and where the CAVLC decoder is utilized. The memory component is accessed to store and read the recovered data throughout the parsing process. Also, the remaining stages of the H.264/AVC decoder will be able to read all of the desired elements from this component.

The stream parser is modeled as a finite state machine to control the flow of data (see Fig. 4.2). Reading of the NAL units begins when the stream parser is enabled and after all units are read, the iterative parsing of each one commences. As each NAL is parsed the recovered data is stored in a global memory component and the updated information is available to all subsequent NAL units. After all parsing completes and the information is saved, the design returns to its default state. By the end of this design, all appropriate data has been recovered and gradually written to memory.
4.1 Reading NAL units

Reading in the NAL units is accomplished by recursively reading a line from a hexadecimal file until the end of the file is reached. Each line represents a byte of data, which is added to the input buffer. The end of NAL units are detected by a delimiter, which is defined as three consecutive bytes of zero followed by a fourth byte equal to one. When a delimiter is detected, it is discarded and the NAL unit is created. At the end of this design, all NAL units have been read in and are ready to be parsed.

4.2 Parsing NAL units

The goal of this design is to recover all necessary data from the NAL units. As each parser algorithm is invoked, more information is found and could represent the slice header, data, or parameter information. The design consists of much control logic for utilization of the three parsing algorithms and for saving the syntax elements they produce. A state machine
is used to manage the flow of the design and is a vital part in the organization and control of data. There are a total of eighty-six states within this design and a simplified version can be viewed in Figure 4.3.

It can be noticed that the Basic and Exp-Golomb parsers are used by every state, which signifies their importance in recovering single syntax elements throughout the video parsing process. Even though the CA VLC parser is only used by one state, it constitutes the most computational complexity and time consumption than the other two parsers. Its complexity derives from the intensive algorithms it must endure to produce multiple coefficient values.

### 4.2.1 Basic Decoding

While the H.264/AVC standard specifies the Basic decoding scheme to decode signed and unsigned integers, bytes, and strings, this design only has the capability to decode unsigned integers and bytes. Since the CABAC decoding algorithm was not supported in this implementation, the other two data types were not required to be decoded. As a result of the algorithm’s simplicity, hardware was not required to be used; however, it was implemented in this design to remain consistent with the rest of the video parser implementation. The design of the basic decoding scheme takes in as input the current NAL payload and the size of the desired syntax element, which could range from 1-bit to 8-bits. Since the range is fixed, a case statement based on the syntax element size is used to find the integer valued
element. Within the branches representing sizes 1 through 7 lies another case statement, which provides all the bit configurations of the NAL payload for the particular syntax element size. Choosing this configuration allows for simple hardware representation by the use of multiplexers to model the case statements. The 8-bit case is implemented using the "CONV_INTEGER" function provided in the IEEE library since this function is more efficient than using a 256-branched case statement for the 256 possibilities. The decoded element is the integer equivalent to the bit configuration within the matched case statement branch.
4.2.2 Exponential-Golomb Decoding

A structural approach is taken with the Exp-Golomb design and a modified version of [19] is implemented, where a 32-bit accumulator and a 32-bit shift register are removed. The removal of the unnecessary components provides an increase in performance due to the Exp-Golomb decoding mechanism performing only one syntax element recovery at a time. As a result, both an accumulator to track what bits of the input buffer have been consumed and a register to shift the data in preparation for subsequent parsing are not needed. The resulting implementation has a decrease in complexity and power consumption. The hardware design can be viewed in Figure 4.4 and consists of five components: first-one detector, two bit shifters, an adder, and a post-processing module.

![Diagram of Exp-Golomb Decoder]

Figure 4.4: Hardware Design of the Exp-Golomb Decoder

An Exp-Golomb encoded codeword is formatted as \([M \text{ zeros}] [1] [M-\text{bits of information}]\). Given the maximum codeword length is 32-bits and the format of an Exp-Golomb encoded codeword, it is a guarantee that the first 15 bits of data will contain a one.

The goal of the first-one detector (see Fig. 4.5) is to find the location of the "1" located
The output of the first-one detector is sent to a shifter and adder to produce the code length, which is defined as $2^M + 1$. A modified version of this value ($32 - \text{code length}$) is used by a 32-bit shifter to shift the input and produce $(\text{codeNum} + 1)$, which is used by the post-processing module to recover the syntax element. The final stage of the Exp-Golomb...
parser is controlled using a multiplexer that chooses the type of parsing to perform. If an unsigned syntax element needs to be recovered, then the output is simply codeNum, and when a mapped element is parsed a look-up is performed. When a signed or truncated element is desired Eq. 4.1 or Eq. 4.2 are used, respectively.

\[
syntaxelement = (-1)^{(\text{codeNum}+1)} \times (\text{codeNum} + 1) \div 2
\]  
\[\text{(4.1)}\]

\[
syntaxelement = (\text{codeNum} + 1) \% 2
\]  
\[\text{(4.2)}\]

### 4.2.3 Context-Adaptive Variable Length Coding (CA VLC)

![Architecture of CA VLC Decoder](image)

Figure 4.6: Architecture of CA VLC Decoder

Out of three decoding algorithms implemented, the CA VLC is the most complex. This design consists of thirteen hardware components, where the highest level is designed using a large state machine to manage the data flow. There are three main components, which assist in the completion of parsing CA VLC coded information:
1. parse the coeff_token value to recover the amount of trailing ones and total coefficients (parse_coeff_token)

2. parse the number of trailing ones and level values for all non-zero coefficients (trailing1s_level_wrapper)

3. parse the total amount of zeros and the location of each zero within the coefficient array (totalZeros_runBefore_wrapper)

**parse_coeff_token**

![Diagram of parse_coeff_token](image)

*Figure 4.7: Architecture for parse_coeff_token*

This component is an original FSM-based (Finite State Machine) design to handle the computational complexity and provide data flow management. The goal of this component is to parse the coeff_token codeword, which results in the production of two values: the number of non-zero coefficients and trailing ones. These values are found via a VLC look-up table, where the choice of table is dependent on the previously decoded macroblocks. Figure 4.8 shows the naming conventions for neighboring macroblocks, where each ones address and index in the macroblock array are found to help determine which look-up table to use.

<table>
<thead>
<tr>
<th>mbAddrD</th>
<th>mbAddrB</th>
<th>mbAddrC</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbAddrA</td>
<td>CurrMbAddr</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4.8: Neighboring Macroblocks of the Current Macroblock in Frame Mode*
As a result of the computational complexity inherent in the parsing of the codeword, FSM-based designs are used throughout the process to help control the massive amount of data flow and use of many utility components. Figure 4.9 displays the components that make up `parse_coeff_token` in a hierarchical organization. When the CAVLC component is enabled, the 64-bit input buffer is filled from the incoming data stream to allow for faster data access throughout this parsing procedure. The purpose of gathering the neighbor information is to assist in determining which VLC table to use to find the total number of coefficients and number of trailing ones.

![Hierarchical Organization of parse_coeff_token](image)

**Figure 4.9: Hierarchical Organization of parse_coeff_token**

- **Division** is used throughout the CAVLC decoding process. Even though division by two can be executed as performing a bitwise shift right, there are many cases where a divisor of two is not used. As a result, it is necessary to implement a integer division design that can be used in the CAVLC process. The hardware components and organization are derived from [13] and consists of four basic components: a multiplexer, register, down-counter, and right-to-left shift register.

The usage of the division function is controlled by a state machine, shown in Figure 4.10, and is a part of all components that need to perform integer division. Once the division is enabled, the numerator and denominator are assigned values, the load/enable signals for the division component’s internal registers are set, and
these values are held at the input for two clock cycles to ensure proper signal assignment within the block. When the division completes, the necessary output signals are assigned to internal signals and during the final state they are latched into registers.

Figure 4.10: State Machine Used By All Components Utilizing the Integer Division Function

- **mb_is_available** determines if a macroblock is available and is accomplished by a simple comparison with its address, the value of zero, and the current macroblock’s address. The macroblock is available if it has a valid address, it is greater than zero, and it is greater than the current one’s address, which would imply it has not been analyzed.

- **getMacroblockIndex** determines a macroblock’s index in the macroblock array, which is performed using a simple look-up into the array using the known address.

- **get_neighbor_mb_address** finds all four neighboring macroblocks and returns their addresses, if they exist. It supports macroblocks that are encoded in frame or field mode, which could have been done independently on vertical pairs of luma macroblocks and is denoted by the MbaffFrameFlag signal. The MbaffFrameFlag being
set denotes that the pair of macroblocks are coded in frame mode, otherwise they are coded in field mode (see Figure 4.11). Table 4.1 shows how each neighboring address is calculated depending on the value of MbaffFrameFlag, if they exist.

<table>
<thead>
<tr>
<th>Neighbor Address</th>
<th>Frame Mode</th>
<th>Field Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbAddrA</td>
<td>CurrMbAddr - 1</td>
<td>2 * (CurrMbAddr/2 - 1)</td>
</tr>
<tr>
<td>mbAddrB</td>
<td>CurrMbAddr - PicWidthInMbs</td>
<td>2 * (CurrMbAddr/2 - PicWidthInMbs)</td>
</tr>
<tr>
<td>mbAddrC</td>
<td>CurrMbAddr - PicWidthInMbs+1</td>
<td>2 * (CurrMbAddr/2 - PicWidthInMbs+1)</td>
</tr>
<tr>
<td>mbAddrD</td>
<td>CurrMbAddr - PicWidthInMbs-1</td>
<td>2 * (CurrMbAddr/2 - PicWidthInMbs-1)</td>
</tr>
</tbody>
</table>

Table 4.1: Neighboring Macroblock Address Calculations for Frame and Field Modes

- **get_4x4_luma_scan** returns a block index when given a luma or chroma location (xW,yW) by using Eq. 4.3. The division function previously discussed is instantiated four times to handle the use of division used within this equation.

\[
luma4x4BlkIdx = 4\times(xW/8) + 8\times(yW/8) + 1\times((xW\%8)/4) + 2\times((yW\%8)/4) \tag{4.3}
\]

- **get_neighbor_location** performs the functionality of finding a neighbor’s location relative to the upper left corner of the returned address. A neighboring macroblock could contain luma or chroma type coefficients, where the size is expressed in terms of the number of coefficients, which are 16x16 and 8x8, respectively. Given a luma
or chroma location, type of block, the MbaffFrameFlag signal, and the current macroblock address, this component is able to produce a macroblock address where the given location resides as well as a new location expressed relative to the upper left corner of the found address. This entity is implemented using six processes, where the first one, depending on the value of MbaffFrameFlag, finds the macroblock address, mbAddrN, or sets necessary flags for future use. Table 4.2 and Figure 4.12 show how the signal mbAddrN is assigned when given the luma or chroma location (xN, yN). When the address is found, the location of the neighboring luma location (xW, yW) is calculated relative to the upper left corner of mbAddrN using the following equations:

\[
    xW = (xN + maxWH) \div maxWH \\
    yW = (yN + maxWH) \div maxWH
\]  

<table>
<thead>
<tr>
<th>xN</th>
<th>yN</th>
<th>mbAddrN</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than 0</td>
<td>less than 0</td>
<td>mbAddrD</td>
</tr>
<tr>
<td>less than 0</td>
<td>0 ... maxWH-1</td>
<td>mbAddrA</td>
</tr>
<tr>
<td>0 ... maxWH-1</td>
<td>less than 0</td>
<td>mbAddrB</td>
</tr>
<tr>
<td>0 ... maxWH-1</td>
<td>0 ... maxWH-1</td>
<td>CurMbAddr</td>
</tr>
<tr>
<td>less than maxWH-1</td>
<td>less than 0</td>
<td>mbAddrC</td>
</tr>
<tr>
<td>less than maxWH-1</td>
<td>0 ... maxWH-1</td>
<td>not available</td>
</tr>
<tr>
<td>any value</td>
<td>less than maxWH-1</td>
<td>not available</td>
</tr>
</tbody>
</table>

Table 4.2: mbAddrN Specification with MbaffFrameFlag equal to Zero [17]

- **find_luma_neighbors** finds the neighboring luma or chroma addresses and indices, if they exist. Since the AVC Standard specifies the same algorithm for finding neighbors of luma and chroma macroblocks, this component has the ability to find either type of neighbor.

The data flow of the find_luma_neighbors component is shown in Figure 4.13. The first step is to find the (x,y) location of the upper-left luma sample for the given 4x4 block index and Eqs. 4.6 and 4.7 are used to perform the computation.
<table>
<thead>
<tr>
<th>xN</th>
<th>yN</th>
<th>currMbFrameFlag</th>
<th>mbAddrX</th>
<th>mbAddrY</th>
<th>mbAddrN</th>
<th>yM</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>1</td>
<td>mbAddrD</td>
<td>1</td>
<td>mbAddrD + 1</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>0</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrD</td>
<td>1</td>
<td>mbAddrD + 1</td>
<td>2^yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrD</td>
<td>0</td>
<td>mbAddrD</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrD</td>
<td>1</td>
<td>mbAddrD + 1</td>
<td>yN</td>
</tr>
<tr>
<td>&lt; 0</td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>1</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>0</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>1</td>
<td>mbAddrA + 1</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>0</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>1</td>
<td>mbAddrA + 1</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>0</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>1</td>
<td>mbAddrA + 1</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>0</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>1</td>
<td>mbAddrA + 1</td>
<td>yN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>mbAddrA</td>
<td>0</td>
<td>mbAddrA</td>
<td>yN</td>
</tr>
</tbody>
</table>

Figure 4.12: mbAddrN Specification with MbaffFrameFlag equal to One [17]
Figure 4.13: Architecture of find_luma_neighbors

\[ x = \text{InverseRasterScan}( \text{luma}4x4\text{BlkIdx} / 4,8,8,16,0 ) \]

\[ + \text{InverseRasterScan}(\text{luma}4x4\text{BlkIdx}mod4,4,4,8,0) \]  \hspace{1cm} (4.6)

\[ x = \text{InverseRasterScan}( \text{luma}4x4\text{BlkIdx} / 4,8,8,16,1 ) \]

\[ + \text{InverseRasterScan}(\text{luma}4x4\text{BlkIdx}mod4,4,4,8,1) \]  \hspace{1cm} (4.7)

Once a location is calculated, it is modified slightly so it would be possible to calculate the location of the neighbors to the left and above the current macroblock. The corresponding neighbor’s macroblock addresses and locations, relative to the upper left corner of their containing macroblock, are then found by utilizing the get_neighbor_location component previously discussed. Once the neighbor’s addresses (mbAddrA, mbAddrB) and corresponding locations ((xA,yA), (xB,yB)) are found, the 4x4 luma or 4x4 chroma block index relative to the upper left corner of the found macroblock is calculated using the get_4x4_luma_scan component. The resulting information is the two neighbor addresses and corresponding block indexes.
This entity is designed as a ten-state state machine (see Fig. 4.14), where two components previously described, get_neighbor_location and get_4x4_luma_scan, are enabled and disabled as needed. The division function is also used to perform the necessary division and modulation calculation to find the (x, y) location. If any of the neighbors do not exist, their index is not attempted to be found. The found index(es) are saved to local signals and the state transitions to done_state, where the final outputs are latched.

![State Machine Diagram](image)

**Figure 4.14: State Machine for find_luma_neighbors**

- **coeff_array** models the coeff token VLC look-up tables and is implemented as a state machine to manage the data flow (see Fig. 4.15). When this component is enabled, the table to be used is determined based on an identification number passed in. The parsing begins in the parse entry 0 state with a comparison between the first entry in the table and the input data. The next state, parse entry 1, determines if the end of
current table entry has been reached, if a full match has been found, if more comparisons are necessary, or if the end of the table has been reached. The comparison between the current table entry and the input data is performed one bit at a time, where an index value is used to determine where in the entry the comparison is occurring. When the table entry and input data no longer match (NO LUT entry match), the table entry value is incremented and the index value is reset to allow for the next table entry to be examined for a match. Otherwise, if the table entry and input data does match up to the current index value (LUT entry match), then the index is simply incremented to continue the comparison between the two values. The end of a table entry is denoted when the index value equals the entry size or when the maximum
table entry size is reached. A successful full match is found when either of these conditions are encountered during a comparison. Even though it is expected to always find a match, the search would end if the table entry value exceeded 62 since there are only 62 entries in the table.

**trailing1s_level_wrapper**

After the number of total coefficients (TotalCoeff) and amount of trailing ones (TrailingOnes) are found using parseCoeff_token, the signs of the TrailingOnes and values of the coefficients are calculated (see Fig. 4.16). This design is also original, where it consists of three processes:

1. control the use of the level parser, latch the recovered level value, calculate the suffix length based on that value

2. shift the data buffer after each level parser completion to allow for the next level parser to retrieve data from the first data index

3. compile the final level values, which are based on the values of the TotalCoeff and TrailingOnes signals, into an array

![Diagram](image)

Figure 4.16: Architecture of trailing1s_level_wrapper
The desired number of data bits is read from the input buffer to determine the sign of all TrailingOne bits; this number is equivalent to TrailingOnes. The sign of the one is negative when the data bit is zero and positive otherwise. Next, the values of the coefficients are found by finding the level prefix and suffix values and respective lengths. With these values the following equations are used to find the levels:

\[
\text{levelCode} = (\text{levelPrefix} \times (2^{\text{suffixLength}})) + \text{levelSuffix}
\]  

(4.8)

For even-valued levelCode:

\[
\text{level} = (\text{levelCode} + 2) \div 2
\]  

(4.9)

For odd-valued levelCode:

\[
\text{level} = (-1 \times \text{levelCode} - 1) \div 2
\]  

(4.10)

Once the levels are compiled, they are serially written out to a memory element that holds the recovered values and are used by the next design, totalZeros_runBefore_wrapper.

**totalZeros_runBefore_wrapper**

The final stage in performing the CA VLC decoding scheme involves two steps: (1) recovering the total amount of zeros in the coefficient array and (2) determining the runs of zeros between the already found level values. This original design encapsulates both algorithms and controls their utilization with a state machine, whose diagram is shown in Figure 4.18.

The number of zeros are found by enabling parse\_total\_zeros and the runs of zeros are found by enabling parse\_run\_before for each desired run value. Once all the necessary runs are recovered, the final run value is assigned the remaining amount zeros. Based on the runs of zeros found, the locations of the coefficients within an array are calculated with the use of fifteen adders and multiplexers. The modeled architecture can be seen in Figure 4.19 and derives from the standardized algorithm, where the subsequent coefficient locations are
dependent on the previous. Even though the range of TotalCoeff is fixed, the architecture accounts for its dynamic value and is achieved by placing the multiplexers before the input of one adder operand, where the previous coefficient location value could be used, if it existed. The level values recovered by the trailing1s_level_wrapper component are then placed where appropriate within the final coefficient array.

- **parse_total_zeros and parse_run_before**: Determining how many coefficients are zeros and the location of the runs of zeros consists of enabling a look-up table and registering the results upon completion. A control signal is used to determine which type of table to use: (1) finding the total amount of zeros for luma or chroma type of neighbors, or (2) finding the runs of zeros. The total number of coefficients (TotalCoeff) is used to choose a specific table to use. The actual table look-up process is controlled using a state machine (see Fig. 4.11), where each table entry is read and compared per bit to the data stream. Once a complete match is found, the corresponding zeros or run of zeros result is registered for later use.
Figure 4.18: State Machine Modeling the totalZeros_runBefore_wrapper design
Figure 4.19: Hardware Architecture for the Coefficients’ Array Indices
5. Implementations and Testing

The designed architecture was targeted for a low power ASIC and an FPGA, where the simulated ASIC implementation out performed the FPGA. For both platform targets, it was noticed that the CAVLC decoder was significantly more influential on the power consumption and performance, which is due to its computational complexity and use of VLC look-up tables. Moreover, the necessary use of the tables warranted the most power consumption and limited the performance of the entire implementation. This is due to the table sizes and the look-up algorithm implemented.

In the area of H.264/AVC video parsing, academic research has been mostly focused on the CAVLC design because of its characteristics and global impact on the decoding process. As a result, the comparisons presented are mostly based on the CAVLC implementation. It should also be noted that the other aspects of the video parser have shown little impact in the overall power consumption and performance, when compared to the CAVLC decoding; therefore, the comparisons can be justifiably extended to the entire video parser.

![Diagram of 16x16 Luma Block and 8x8 Cb or Cr Block](image)

**Figure 5.1:** 16x16 Luma Block (left) and 8x8 Cb or Cr Block (right)

A single macroblock is represented by a 16x16 luma, an 8x8 Cb, and an 8x8 Cr array (see Fig. 5.1). Since each iteration through the CAVLC design recovers a 4x4 block of
coefficients, the component must be invoked 24 times to recover an entire macroblock. Also, the number of clock cycles the CAVLC consumes greatly depends on the time spent performing the VLC table look-ups. As a result, one macroblock can be recovered in as little as 1,320 cycles or as many as 8,184 cycles. The low limit represents all VLC look-ups matching in the first entry and the upper limit represents every VLC look-up resulting in a match in the last entry. For example, a single 720 HD frame (1280x720 pixels) is made up of 3600 macroblocks, which results in the CAVLC cycles to produce one frame to range from 4,752,000 to 29,462,400 cycles.

<table>
<thead>
<tr>
<th>Video Resolutions</th>
<th># MBs</th>
<th>65nm ASIC (fps)</th>
<th>65nm Virtex 5 FPGA (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QCIF (176x144)</td>
<td>99</td>
<td>740</td>
<td>285</td>
</tr>
<tr>
<td>CIF (352x288)</td>
<td>396</td>
<td>185</td>
<td>71</td>
</tr>
<tr>
<td>NTSC (720x480)</td>
<td>1350</td>
<td>54.3</td>
<td>20.9</td>
</tr>
<tr>
<td>720 HD (1280x720)</td>
<td>3600</td>
<td>20.5</td>
<td>7.9</td>
</tr>
<tr>
<td>1080 HD (1920x1080)</td>
<td>8100</td>
<td>9.1</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of ASIC and FPGA Performance for Various Resolutions

The highest operating frequency of the 65nm worst case ASIC design is 167 MHz (6 ns), which results in a frame throughput of about 54.3 fps for NTSC frames. The highest operating frequency of the FPGA design is 64.23 MHz (15.57 ns), which results in a the lowest throughput of about 20.9 fps for NTSC frames. A performance summary of the ASIC and FPGA implementations can be seen in Table 5.1 and Fig. 5.2, where various resolution capabilities are listed. It is noticed the ASIC design out performed the FPGA implementation, while being able to handle up to NTSC quality video at real-time speeds. Further analysis of the different implementations is presented in the following sections.

5.1 ASIC Implementation

Synopsys Design Compiler and VHDL were used to synthesize and describe the design while targeting a low power 65nm ASIC (TCBN65LPBC). Worst case components were used for synthesis, with a temperature of 125 degrees Celsius, voltage supply of 1.08V,
and varying amount of wire load, which was dependent on the design size. It is noticed that decreasing the clock period constraint forced the design to use more power in order to properly perform under the high frequency. Conversely, increasing the clock period showed the design was given more time to perform its computations, which decreased the power usage.

The results show the video parser design is low power and has enough performance to handle NTSC frames at real-time speeds. Using the 65nm technology, the parser consumed 5.462 mW of dynamic and 0.066 mW of leakage power, while operating at 6 ns (166 MHz) and taking about 7.9 us to recover one macroblock.

Figure 5.2: The ASIC and FPGA Implementations’ Performance for Various Resolutions and Frame Formats
5.1.1 Sub-Design Comparisons

Basic and Exp-Golomb Decoding

The implementation of the Basic decoding algorithm had the least impact on the overall design, with respect to area, power, and performance. As a result of the algorithm’s simplicity, only combinational logic was used and consisted of mostly multiplexers, which chose the final syntax element based on the size required and data bit pattern (see Figure 5.3).

![Figure 5.3: Basic Decoding ASIC](image)

The implementation of the Exp-Golomb decoding algorithm had a slightly greater impact on the design, which was expected because of its more complex algorithm. The resulting design was consistent with the architecture presented in Figure 4.4. Figure 5.4 shows the resulting ASIC implementation.
CAVLC Decoding

As expected, the CAVLC implementation had the greatest impact on the overall design. Its use of VLC look-up tables and computational complexity caused the architecture to constitute most of the video parsing efforts. While all individual designs were analyzed, only the top three sub-designs will be explored due to the low impact the other ones presented. For the 65nm technology, the entire decoder reported a dynamic power usage of 0.988 mW, frequency of 125 MHz, and area of 207,000 gates. Figure 5.5 shows the ASIC implementation with the sub-designs not expanded.

The original FSM-based design (parseCoeff_token) and the trailing1s_level_wrapper design presented in this thesis performed relatively well, when compared to the other
CA VLC sub-design, while contributing relatively low area and power numbers. The contributions from all three components are further analyzed in the following sections.

![CA VLC Sub-Design Comparisons](image)

**Figure 5.6: Power, Area, and Performance Comparison of CAVLC Sub-Designs**

1. parse_coeff_token was a completely original design and consisted of mostly sequential logic, which represented the many state machines that were used in the modeling of this algorithm. The extensive use of state machines throughout the design allowed for predictable synthesis results as well as data flow management. Its model was the most complex out of the other two sub-designs due to the two recovered values, TotalCeoff and TrailingOnes, requiring a lot of memory look-ups and computational complexity. However, since there was only one small VLC look-up table used, the design exhibited higher performance and lower power usage than the subsequent designs. The reported dynamic power usage was 0.296 mW, the frequency was 364 MHz, and area was 29,000 gates. Figure 5.7 shows the resulting ASIC implementation.

2. The trailing1s_level_wrapper design was also completely original. The implementation used no look-up tables to find the values of the coefficients (levels), rather
calculations were performed based on the incoming data and previously calculated values. The necessary computations resulted in a bigger performance impact on the overall design with a decrease of 56%, when compared to parse_coeff_token. However, the power consumption showed a 23% decrease and is due to the computational differences. Recovering the TrailingOnes’ sign did not noticeably contribute to either metric because the input data was directly interpreted to find the values. The reported dynamic power usage was 0.227 mW, the frequency was 162 MHz, and area was 26,000 gates.

3. The sub-designs of totalZeros_runBefore_wrapper are featured in [8] and aim to be low power implementations; however, because of their use of look-up tables to recover both the total amount of zeros and each run of zero it uses more power and causes a slight decrease in performance. The wrapper around the two parsers was
implemented as a state machine, which controlled the use of the designs and processed their outputs. When the necessary information was recovered by the parsers, they were sent through a sequential ordering of adders and multiplexers to compile the final list of coefficient values (see Fig. 4.19). The reported dynamic power usage was 0.464 mW, the frequency was 140 MHz, and area was 223,000 gates. The design constitutes 47% of the CA VLC’s power usage and 79% of its area.

5.2 FPGA Implementation

The video parser ASIC design showed lower gate usage and higher performance compared to the FPGA implementation. The observed differences are due to the quality of the synthesis results as well as the actual targeting platform. Xilinx ISE was used to synthesize and verify the design while targeting the 65nm Virtex 5 LX FPGA. The resulting implementation showed a high usage of physical resources, with the CA VLC decoder having the largest impact on the design. The parser used 50,285 total slices and 7,148 total registers, with the corresponding frequency reaching 64.23 MHz (see Fig. 5.10). Since over 90% of the FPGA resources were consumed for the video parser, it is necessary for the large main memory component to be implemented using off chip memory to allow for the remaining decoder components to fit on the FPGA.
5.2.1 Sub-Design Comparisons

The Virtex-5 FPGA contains 25,920 configurable logic blocks (CLBs), where each CLB has access to a switch matrix for routing purposes and consists of two slices. All slices have at least four logic generators (implemented as look-up tables) and four flip-flops, with some slices containing shifters and distributed RAM. It was noticed that the gate count equivalence provided by Xilinx ISE for all designs was consistently more than the ASIC implementation. This could be attributed the compilation efficiency of Synopsys’ Design Compiler, the high performance constraint placed on the FPGA implementation, and/or the accuracy of the gate estimation algorithm.
Basic Decoding

Due to the simplicity of the algorithm, the Basic decoder showed minimal slice usage of 11 slices, where only the look-up tables (LUTs) were used to implement the required logic. The resulting gate equivalence was given to be 77 logic gates, which is a 77% increase over the ASIC design.

Exp-Golomb Decoding

As expected, the design of the Exp-Golomb decoder showed a larger impact than the Basic decoding, but did not greatly contribute to the overall video parser implementation. There were a total of 154 slices occupied, with 135 LUTs and 68 registers being used. The higher use of logic generators was due to the increase in computations performed. The estimated gate equivalence was reported to be 4,436 gates, which is a 48% increase over the ASIC implementation.

CAVLC Decoding

As with the ASIC implementation, the FPGA-targeted CAVLC decoder showed the greatest impact by constituting 77% of the overall video stream parser implementation. A total of 38,742 slices were used, with 51,492 LUTs and 3,774 registers and a gate equivalence of 397,252 gates, which is a 52% increase compared to the ASIC implementation. The usage of most of the logical resources can be seen in its sub-designs, where the recovery of the total_zeros and run_before values utilizes most of the resources.

1. parse_coeff_token used 1,005 slices, with 1,584 registers and 2,886 LUTs. The gate equivalence was given as 23,772 gates, which is a significant increase from the ASIC design (13,821). The high register usage is attributed to the extensive use of state machines through the developed architecture. Also, the high usage of logic generators is due to the computationally complexity of recovering the TotalCoeff and TrailingOne values.
2. trailingOnes_level_wrapper used 375 slices, with 371 registers and 2,061 LUTs. The gate equivalence was given as 10,283 gates, which is a 55% increase of gates over the ASIC implementation. The low usage of state machines and little amount of computations in the design caused a relatively low amount of registers and LUTs to be used.

3. totalZeros_runBefore_wrapper used 34,395 slices, 1,740 registers, 46,604 LUTs, and had a gate equivalence of 339,223 gates, which is an increase of 66% when compared to the ASIC implementation. The use of two large VLC look-up tables to recover the total_zeros and run_before values and the compilation of the final coefficient array caused the resource usage to be significantly higher than the other CAVLC components.

5.3 Synthesis Simulations

Each component in the video stream parser was synthesized and verified separately to allow for easier debugging and gradual verification. The output of each stage in the process was compared to the calculated results, as well as the functional results from the existing behavioral model presented in [16]. The simulations were performed using ModelSim SE 6.1a and two video files, used in [16], were used as input to the parser. Both files conformed to the Baseline H.264/AVC Profile, where one file contained information representing a single I-type QCIF frame and the other an I-type and P-type QCIF frame. The variations in the two files allowed for multiple functionalities to be tested, as well as provide greater verification between the implemented and behavioral parser.

5.4 ASIC and FPGA Comparisons

Targeting both platforms resulted in very different implementations and resource usage for the same architecture. The ASIC platform utilized a low power and high performance
technology as well as a highly efficient compiler, Synopsys’ Design Compiler Ultra. The FPGA platform utilized the Xilinx synthesizer with the highest speed grade constraint for the Virtex 5 LX FPGA. It was noticed that the video parser ASIC design showed lower gate usage and higher performance compared to the FPGA implementation, as seen in Figures 5.11 and 5.12. The differences between the two implementations is due to the quality of the synthesis results, as well as the actual targeting platform. In terms of gate usage, power, and performance, an ASIC targeted design is the most logical choice for efficiency with regards to those benchmarks.

![FPGA and ASIC Gate Equivalence](image_url)

Figure 5.11: FPGA and ASIC Gate Count Comparison
5.5 Comparisons with Existing Works

Since all the research papers found were fabricated using older process technologies (130nm and higher), the design was targeted as a 130nm ASIC (CB13FS120_TSMC_MAX) for proper comparison. Those results show a dynamic power consumption of 1.14 mW, an 8ns operating frequency (125 MHz), a throughput of about 40.7 fps, and taking about 10.6 us to recover one macroblock.

Within the area of H.264/AVC decoding there are many published research papers available; however, few report detailed information about process technology, area, speed, and power usages. Compared to some existing implementations studied, the architecture presented in this thesis shows lower power and smaller area usages, but lower performance.
A design presented in [7] was designed with a scalable bus architecture, dual memory controller, and used an ARM core as the main controller. It was fabricated using a 130nm process technology and showed to be very high performance with a throughput of 4.1 us/MB (1080 HD); however, compared to the design presented here its size was 312% bigger and showed a 48500% increase in power consumption, with a gate count of 910,000 and power usage of 554 mW.

A high performance multiple-symbol parallel CAVLC decoder was presented in [21] and capitalized on the parallelism found within the decoder’s algorithms. It reported a power usage of 16.8701 mW and throughput of 6.72 us for the recovery of one macroblock, using a 180nm process technology. While the design presented in this thesis falls short of their high speed design by 3.88 us, it saves over 10 mW of power using a smaller technology. Since power is linearly dependent on frequency, 64% of the power consumption increase is due to the high speed of their design, while the rest is attributed to the architecture differences and feature sizes.

A real-time baseline profile decoder was implemented using software on an Intel Pentium 4 platform [20]. This design was studied to provide a pure performance comparison with the architecture presented here. Their design showed a 65% decrease in performance, with a reported average decoding speed of 65 fps for CIF sequences, when using a 2.0 GHz Intel Pentium 4 processor.

Overall, when compared to similar hardware implementations, the presented architecture uses fewer gates and is a very low power design.
6. Future Work

This thesis provided a comparison of one architecture targeting multiple platforms and can be used as a basis for future work. It is possible to combine more advanced algorithms with the implementation presented to achieve desired design targets, such as lower power or higher performance.

A specific component that could be optimized in future work is the CAVLC sub-design which recovers the total_zeros and run_before values. As a result of the extensive use of VLC tables within this component, it contributes the highest percentage to the CAVLC decoder’s overall area, power consumption, and performance constraint. A design which capitalizes on the architecture associated with VLC table look-ups could produce better area, power, and performance contributions. A way to increase the look-up performance is to read all the table entries in parallel and merge the results. This would cause an increase in power and area; however, the time to find an entry would be reduced to one look-up and a read merge.

Since the stream parser is heavily dependent on knowing what bits will be used by each component, it is necessary for prediction algorithms or preemptive bit reading to be performed in order for a decoder step to be successfully performed before the completion of the preceding component. A possible extension to the presented design is to preemptively read bits from the incoming stream and discard all obtained results which are later found to be invalid. This could be performed at various levels of the design hierarchy; however, trade-offs need to be made based on given power and area constraints. It would be desirable to focus improvements on the CAVLC design because it has the most impact on the video stream parser. A possible implementation would be to place the level of abstraction with the sub-designs of the CAVLC decoder, which would provide performance improvement without a large increase in power and area. The preemptive bit reading would allow the
second sub-design to read and process the bit stream starting at all possible bits, based on the range the first sub-design could read. Once the first one completed, the correct result from the second decoder would be chosen. This process would allow for the first and second decoder components to operate in parallel and as a result, increase the overall performance of the CAVLC decoder.
7. Conclusion

A video parser architecture was designed, implemented, and targeted for three different platforms, an FPGA and two low power ASICs. The resulting implementation was a combination of original and leveraged designs, where state machines and low power techniques were explored. While the state machines provided good data flow management and expected synthesis results, it added unnecessary overhead to the implementation and hurt the overall performance. However, placing an enabled latch in front of the partitioned tables within the design resulted in less power consumption and noticeably affected the overall low power consumption of the video stream parser design. Also, enabling the Exp-Golomb decoder only when a syntax element was being recovered assisted with the low power usage.

It was observed that the 65nm ASIC platform served as an acceptable target to achieve a low power design, while performing better than the FPGA targeted implementation, with a 60% increase in performance and 6x increase in area.
Bibliography


