A Novel 5.46 mW H.264/AVC video stream parser IC

Michelle Brown
Kenneth Hsu

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

Recommended Citation
ABSTRACT
This paper presents a 5.46 mW H.264/AVC Video Stream Parser implemented in 65nm. The differences between targeting a video stream parser architecture for a 65nm CMOS ASIC and a Virtex 5 FPGA are also compared. Overall, the ASIC implementations showed higher performance and lower area than an FPGA, with a 60% increase in performance and 6x decrease in area.

I. INTRODUCTION
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. The flexibility provided with the H.264/AVC video coding standard is enabled by the standardization of the formatted bit stream and video decoder. As a result, 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 various resolution support results in the need for more sophisticated decoder implementations; therefore, hardware designs become a necessity.

In the first stage of the video decoding process, a compressed bit stream is received by the video parser, where all data and parameter information is recovered. By focusing on the initial decoding step and targeting various platforms, further insight into the video stream parser is gained.

II. BACKGROUND
The implementation presented here 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 Fig. 1, where the compressed video stream is read and the Network Abstraction Layer (NAL) units, which were used to provide a layer of abstraction over the data, are individually parsed. Various decoding algorithms are invoked, including Basic, Exponential-Golomb (Exp-Golomb), and Context-Adaptive Variable-Length Coding (CAVLC) decoding, to recover the information within the units. When the parsing completes and all the sequence parameters have been recovered, the remaining steps in the H.264/AVC decoding process are able to commence.

Figure 1: Overall design of the implemented video stream parser.

A. Basic Decoding
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.

B. Exp-Golomb Decoding
The Exp-Golomb decoding algorithm is slightly more complex and uses a single codeword look-up table (VLC table). Variable Length Coding uses smaller codeword lengths for frequently occurring data and larger codeword lengths for less frequent 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. Depending on the type of NAL unit received, one of the four Exp-Golomb decoding algorithms might be used. Each decoding algorithm determines the codeNum value by using the equation codeNum = (2M + INFO - 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.
C. CAVLC Decoding

Context-Adaptive Variable Length Coding (CA VLC) decoding is a type of run length decoding, where the number of zeros transmitted is reduced. As a result of the algorithm’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 number of 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 macroblock in zig-zag order, starting at the top left of the block. The low frequency values are located in the top left and tend to have larger values than those at higher frequencies.

The overall algorithm encompasses the decoding of five syntax elements from the received coefficients: coeff token, sign of trailing ones (T 1 s), level, total zeros, and run before (see Fig. 2).

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coeff</td>
<td>the number of all non-zero coefficients (total_coeff) and the number of trailing ones (T 1 s) are encoded by this syntax element.</td>
</tr>
<tr>
<td>T 1 s</td>
<td>the sign of each T 1 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 T 1 s is encoded by this syntax element.</td>
</tr>
<tr>
<td>tot_zeros</td>
<td>the total number of zero coefficients preceding the previous non-zero coefficient in the reverse zig-zag order is encoded by this syntax element.</td>
</tr>
<tr>
<td>run_before</td>
<td>the number of successive zero coefficients preceding each non-zero coefficient in reverse zig-zag order.</td>
</tr>
</tbody>
</table>

Figure 2: The CAVLC syntax elements recovered during the execution of the CAVLC decoder. [1]

The sign of the T 1 s 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 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.

III. DESIGN

The video stream parser consists of three main functionalities: reading NAL units, parsing NAL units, and a memory component (see Fig. 3). The recovery of all the picture parameter and data information occurs within the parsing component, where the Basic, Exp-Golomb, and CA VLC 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: slice header, sequence, or picture data; and 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 CA VLC decoder is utilized. The memory component is accessed to store and read the recovered data throughout the initial parsing process and throughout the remaining stages of the H.264/AVC decoder.

A. Basic Decoding

The design of the basic decoding scheme uses the current NAL payload and the size of the desired syntax element, which could range from 1 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.

B. Exp-Golomb Decoding

A structural approach is taken with the Exp-Golomb design and a modified version of [2] 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 decreased complexity and power consumption. The hardware design can be seen in Fig. 4 and consists of first-one detector, two bit shifters, an adder, and a post-processing module.

Figure 4: Hardware design of the Exp-Golomb decoder.

C. CAVLC Decoding

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 steps that assist in the completion of parsing CAVLC coded information:

1. parse the coeff token value to recover the amount of trailing ones and total coefficients
2. parse the number of trailing ones and level values for all non-zero coefficients
3. parse the total amount of zeros and the location of each zero within the coefficient array

Figure 5: Architecture of the CAVLC Decoder.

a. parse_coeff_token:
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. 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.

b. trailing_ones_level_wrapper:
After the number of total coefficients (TotalCoeff) and amount of trailing ones (TrailingOnes) are found using parse_coeff_token, the signs of the TrailingOnes and values (or levels) of the coefficients are calculated. This design consists of three processes, where each one performs a specific task:
1. control the use of the level parser, latch the recovered level value, calculate the suffix length
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

Figure 6: Architecture of trailing_ones_level_wrapper component.

c. totalZeros_runBefore_wrapper:
The final stage in performing the CAVLC 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. The design encapsulates both algorithms and controls their utilization with a state machine. The number of zeros is 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 of zeros, if any. The final coefficient array is compiled with the use of fifteen adders and multiplexers, where the subsequent coefficient values are dependent on the previous. Even though the range of TotalCoeff is fixed, the architecture accounts for its dynamic value by placing multiplexers before the input of one adder operand, where the previous coefficient value could be used, if it existed.

Figure 7: Architecture of totalZeros_runBefore_wrapper component.

IV. RESULTS

The designed architecture was targeted for an FPGA and low power ASIC, where both ASICs out performed the FPGA implementation. For all platform targets, it was found 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 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.

A single macroblock is represented by a 16x16 luma, an 8x8 Cb, and an 8x8 Cr array. Because 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.

Figure 8: Platform comparisons between the 65nm ASIC, 130nm ASIC, and FPGA for various resolutions.

A. Comparison between platforms

Synopsys Design Compiler and VHDL were used to synthesize and describe the design while targeting a low power 65nm ASIC (TCBN65LPBC). The worst case components were used, 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. The results show the video parser design is low power and can handle NTSC frames at real-time speeds. The dynamic and leakage power was 5.46 mW and 0.07 mW, while operating at 6 ns (166 MHz) and taking about 7.9 us to recover one macroblock.

The video parser ASIC design showed lower gate usage and higher performance compared to the FPGA implementation and the observed differences are 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.

B. Comparison 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), 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 [4] 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, 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. Another high performance design was presented in [6] and reported a power usage of 16.8701 mW and throughput of 6.72 us for the recovery of one macroblock, using an 180nm process technology. While the design 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.

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

V. 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. It was observed that the ASIC platforms 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.

REFERENCES


200