A High Performance CABAC Encoder

Hassan Shojania and Subramania Sudharsanan
Department of Electrical and Computer Engineering
Queen's University, Kingston, Ontario K7L 3N6, Canada
Email: {shojania, sudha}@ee.queensu.ca

***Abstract*— One key technique for improving the coding efficiency of H.264 video standard is the entropy coder, context-adaptive binary arithmetic coder (CABAC). However the complexity of the encoding process of CABAC is far higher than the table driven entropy encoding schemes such as the Huffman coding. CABAC is also bit serial and its multi-bit parallelization is extremely difficult. For a high definition video encoder, multi-giga hertz RISC processors will be needed to implement the CABAC encoder. In this paper, we provide efficient solutions for the arithmetic coder and the renormalizer. An FPGA implementation of the proposed scheme capable of 54 Mbps encoding rate and test results are presented. A 0.18 μm ASIC synthesis and simulation shows 87 Mbps encoding rate utilizing an area of 0.42 mm^2.** [1]

I. Introduction

The H.264 video standard includes several algorithmic improvements for the hybrid motion compensated, DCT-based video codecs [1]. One key technique for improving the coding efficiency is the entropy coder, context-adaptive binary arithmetic coder (CABAC) [2]. The CABAC utilizes a context-sensitive, backward-adaptation mechanism for calculating the probabilities of the input symbols. The context modeling is applied to a binary sequence of the syntactical elements of the video data such as block types, motion vectors, and quantized coefficients binarized using predefined mechanisms. Each bit is then coded with either adaptive or fixed probability models. Context values are used for appropriate adaptations of the probability models. There can be a total of 399 contexts representing various different elements of the source data. Each processing step of binarization, context assignment, probability estimation, and binary arithmetic coding is designed with some computational complexity constraint. For instance, the binary arithmetic coder uses a version that has no multiplications. However the complexity of the encoding process in its totality is far higher than the table driven entropy encoding schemes. The CABAC is also bit serial and multi-bit parallelization as in Huffman type encoding is extremely difficult. For a high definition video encoder, multi-giga hertz RISC processors will be needed to implement the CABAC encoder [3]. Such large frequencies may not suit low power devices such as cameras where H.264 is to become a dominant standard and hence more efficient implementations are needed.

Two recent papers on this subject reported solutions [3], [4] for a CABAC engine. The scheme proposed in [4] uses a hybrid hardware - software approach with some estimation on the number of cycles per bit and the required silicon area. Our previous paper [3] introduced a novel architecture for a CABAC coprocessor that can be easily integrated on system-on-chip designs. It was shown in [3], with FPGA implementation results, that under certain circumstances, the circuit could achieve the speed of single bit encoding for every two clock cycles. One critical step in arithmetic coding is the renormalization of the state registers [5]. However, the design in [3] addressed renormalization using a simple and bit serial circuit that affected overall performance. The renormalization solution presented in [4] is based on a QM-coder implementation [5]. The solution does not elaborate how this is applicable for H.264, particularly with respect to handling "outstanding bits" which is a complex problem (described in II-C). In this paper, we provide efficient solutions for the arithmetic coder and the renormalizer that guarantee the performance in number of cycles per bit, and also address a number of issues that help reduce the silicon area while maintaining the coprocessor architecture presented in [3]. The proposed solution is tested using encoder data generated by H.264 reference software [6] for several standard video sequences. The remainder of the paper provides an overview of the problem, details the proposed architecture and implementations using an Altera FPGA platform and a generic 0.18 μm technology ASIC synthesis, and concludes with a summary and future work in Section V.

II. Overview of CABAC encoding

The CABAC encoding operation consists of major steps of binarization, context-based and bypass binary arithmetic coding, renormalization, and bit generation. We provide an overview of each step to introduce possible challenges in a hardware implementation. A block-level diagram of the proposed architecture is given in Fig. 1 that shows different stages of encoding as described in the H.264 standard specification [1]. For brevity, we shall use the terminology in the standard [1] without proper introductions.

The CABAC encoder is designed as a hardware acceleration block as part of a system-on-chip for encoding H.264 video as presented in [3]. A higher level software running in a control processor generates H.264 syntax elements (e.g. motion vector difference, transform coefficients, ...) which are sent to the CABAC encoder using a FIFO. Each syntax element is binarized to produce one or more binary symbols called *bin*s. Each *bin* along with some other side information (e.g. encode mode, context index) is placed in a second FIFO to feed the binary arithmetic coder. The binary arithmetic coding phase is highly serial and a new bit can only be coded once the previous

[1]This work was supported in part by the Canadian Microelectronics Corporation (CMC) and Sun Microsystems, Inc.

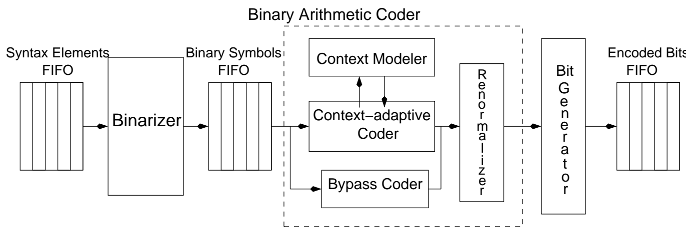

Fig. 1. High-level architecture of CABAC encoder.

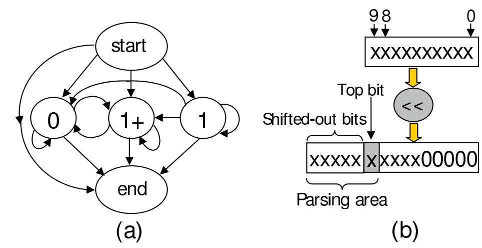

Fig. 2. (a) Flow of renormalization branches (b) Sample update of *codILow*.

bit has been coded and the coding states updated through the renormalization process. The renormalization stage also generates the output stream to be placed in an output FIFO which will be drained by a higher level software.

A. Binarizer

Binarization is a form of pre-processing step that reduces the alphabet size of syntax elements to a maximally reduced binary alphabet. The result is a unique intermediate binary codeword (*bin* string) for each syntax element. The statistical behavior of individual *bin*s can be better modeled in the subsequent context modeling stage than the whole syntax element [2]. Depending on the syntax element, each of its *bin*s can be associated with a context index which represents the probability model of the *bin*. Certain syntactical elements do not use a context-adaptive model and are considered to be equiprobable. Our previous work [3] provided a unified binarizer that handled all syntactical elements of the H.264 format using a well defined generic interface.

B. Arithmetic Coding

Arithmetic coding represents a coded sequence by a tag (*codILow*) and an interval (*codIRange*). As more symbols are coded, the interval decreases and higher precision is needed to represent the sequence identifiers. To address this, the interval and tag values are scaled using a *renormalization* process enabling incremental encoding. The implementation of the basic arithmetic coder is straightforward. The state of binary arithmetic coder is represented by *codIRange* (9 bits) and *codILow* (10 bits) values and updated at encode of each incoming symbol. For the context-adaptive path, the probability model and the most probable symbol (MPS) associated to the bin is retrieved through the context table RAM addressed using a (context) index calculated by the binarizer. Depending on whether the polarity of the input *bin* matches the MPS, one of two coding paths is taken (Fig. 9-7 of [1]). In both cases a reference to the Multiplier (*RangeLPS*) and Next Probability State (*TransIdxMPS/LPS*) ROMs is made. Also, the updated probability state and the MPS are written back to the context table RAM (7-bits in total).

For equiprobable symbols, no probability model is needed since the corresponding *bin*s show a nearly uniform distribution [2]. Hence, a simpler bypass mode is used where no memory access to context RAM or look-up ROM is required. As a result, bypass coding is much simpler compared to context-based coding. The standard specification [1] has combined the coding and renormalization phases of bypass coding (as shown in Fig. 9-10 of [1]) to make the calculation simpler. At the first glance this process may look to require completely separate logic for its implementation. Section III-A will present a unified solution for portions of bypass and context-adaptive codings to streamline renormalization and bit generation.

C. Renormalization

The renormalization process rescales arithmetic coding states. It takes a variable number of iterations to scale *codIRange* to a minimum value of 256 with successive left shifts [1]. The number of iterations, *iter*, varies from zero to eight depending on the incoming *codIRange* calculated in the arithmetic coding stage. Each iteration updates *codILow* by potentially resetting one of its two top bits and then shifting it to the left. A single output bit is generated at each iteration to be added to the output stream. The polarity of generated bit depends on the taken branch. Figure 2(a) names branches on Fig. 9-8 of [1] as *1*, *1*+ and *0* from right to left, and shows the flow of iterations for renormalization of a single *bin* as a state diagram. While the polarity of generated bit for *1* (one) and *0* (zero) branches are already determined, the polarity for *1*+ branch is unknown till a future bit (zero or one) is generated. This future bit could be generated either in the current renormalization process or in a renormalization corresponding to encode of a future symbol that could be several symbols away. As suggested in [1], a counter, *count*, can keep track of the number of these *1*+ bits (bits associated with *1*+ branch: "outstanding bits") until a future bit resolves them to a known value. This dependency on the future bits introduces a serious challenge to hardware implementations as the length of these bits can grow. For example, the standard document [1] does not set an upper limit on *count* and suggests it could grow as large as the slice size. The outstanding bits are resolved to either a one followed by *count* number of zeros or a zero followed by *count* number of ones depending on whether the resolving bit is a one or zero respectively.

The variable number of iterations could force frequent stalls in the arithmetic encoder if not addressed properly since renormalization has to be completed before processing the next incoming bit. This reduces the overall throughput of the coder.

D. Bit Generation

The final bit generation block generates the output bits (based on the instructions received from the renormalizer) and appends them to the output stream. It accumulates the bits while keeping track of number of outstanding bits. It also manages the bit packing of the output stream so that the stream can be presented to the main processor with a FIFO interface.

Clearly some layer of buffering is required for bit packing, size of which depends on the output FIFO width. Until the outstanding bits are resolved as mentioned earlier, they can not be transferred from the buffer to the FIFO.

III. The Proposed Architecture

As previously mentioned, the arithmetic coding and renormalization block together form the bottleneck of CABAC as they are highly serial and a new symbol can not be encoded till the final update of *codIRange* and *codILow* after renormalization. So shortening this path is an important design issue. The proposed architecture that addresses this issue is shown in Figure 3.

A. Arithmetic Coding implementation

The context-adaptive arithmetic coding path requires several memory accesses. This enforces the longest path of the encoder which spans from the multiplier ROM look-up to the end of renormalization. There is not much flexibility in design of this coding mode while the case for bypass coding is different.

By rearranging the bypass coding process of the standard specification [1], a unified renormalization and bit generation mechanism can be used for both context-adaptive and bypass coding modes. Instead of the integrated coding and renormalization for bypass coding as described in Fig. 9-10 of [1], a modified form will allow use of the generic renormalization of context-based coding but with a fixed iteration size of one. Since bypass coding always generates a single bit, it is more beneficial to break bypass coding to *codILow* update and renormalization stages and fold this one-iteration renormalization to the generic renormalization of context-based coding. The procedure described in [1] performs a left shift of *codILow* first, adds it up with *codIRange* when *bin* is equal to one, and renormalizes *codILow* by testing its ninth and tenth bit (referred bit positions are zero indexed). Instead, left shift of *codILow* can be skipped and possible addition with *codIRange* can be done only against its top eight bits. The updated *codILow* is obtained by a single extra OR operation with the zero-th bit of *codIRange*. Then, the rest is taken care of by the generic renormalization and bit generation.

B. Renormalization implementation

A closer look at the renormalization shows that the number of scaling iterations *iter* is equal to the number of leading zeros of *codIRange*. Hence, the new *codIRange* can be simply calculated by left shifting by the lead zero count. Obtaining the new *codILow* is, however, more complex. A straightforward solution could implement renormalization of *codILow* through a ROM lookup. The 10-bit *codILow* could take 1024 different values and number of iterations required to scale *codIRange* to a minimum value of 256 could be eight in the worst case. So a table of 8096 entries would be sufficient where each entry keeps fields such as output bit string, number of outstanding bits generated, polarity of resolving bit, new *codILow*, etc. totaling 28 bits. Further improvement can halve the number of entries by adding special handling for successive taken *1* branches. Even this improvement results in a $4K$ entry ROM.

Due to the subtraction operations on *codILow* in *1* and *1+* branches of renormalization, a simple barrel shifter can not be used directly to mimic the cumulative effect of the iterations for *codILow* update and output bits generation. However, since each subtraction only affects a single bit of value at positions 8 or 9, an *iter*-size left shift of *codILow* still preserves all the necessary information to retrieve the updated *codILow* value. A few special rules are required to derive the updated value:

- The shifted-out bits and the top bit of *codILow* (bit 9) form an *iter+1*-bit *parsing area* to be interpreted from left to right. Fig. 2(b) shows an example with $iter = 5$.
- Only the leading 1's in the *parsing area* are proper output bits which won't need further processing. The other 1's are outstanding bits which need to be resolved either by a following zero in the current *parsing area* or other deterministic 1 or 0 of the subsequent *parsing area*s resulting from encoding of next symbols.
- The first encountered zero bit is to be always ignored.
- The updated *codILow* receives bits 0 to 8 of the shifted *codILow*. If the shifted-out bits (top *iter* bits) are all 1, bit 9 is copied over too; otherwise bit 9 must be set to 0.

These rules help construct a parser using combinational logic to obtain updated *codILow* and a bit-string of length *iter*. This combinational logic effectively replaces the $4K$ entry ROM discussed earlier. If no outstanding bits are present, the string will correspond to output bits. To speed up update of *codILow*, the combinational circuit is split into two stages corresponding to update of *codILow* (*Low Renormalizer*) and generation of the output bits (*Parser*). Since *codILow* is updated at the end of first stage, encode of next symbol can start right after this update. This enables decoupling of bit generation from the arithmetic coding and renormalization stages as shown in Figure 3.

C. Bit Generation implementation

The string of parsed bits can not be directly sent to the output stream since the polarity of the potential outstanding bits is yet to be resolved. An intermediate buffer is required to accumulate these parsed bits and issue them to the output FIFO when the polarity of the outstanding bits is resolved. The parsing mechanism tries to resolve the outstanding bits within each generated *parsing area* whenever possible. By this mechanism only the outstanding bits that absolutely need to wait for encode of later symbols remain unresolved. The parser provides start/end positions of outstanding bits within the parsed area to the intermediate buffer. An intermediate buffer of 64-bits (twice the size of a typical output FIFO width of 32-bits) equipped with a couple of pointer registers (e.g. indicating start position of unresolved outstanding area) could successfully handle incoming parsed bits. To prevent long sequence of outstanding bits from overflowing the intermediate buffer, such a sequence is reduced through an outstanding words counter. In the worst-case scenario, this method can handle a sequence of at least 96 outstanding bits without the need to stall the coding process. The number 96 is well above the longest sequence of outstanding bits of 42 encountered in our test contents. Stall is only needed when a long sequence of

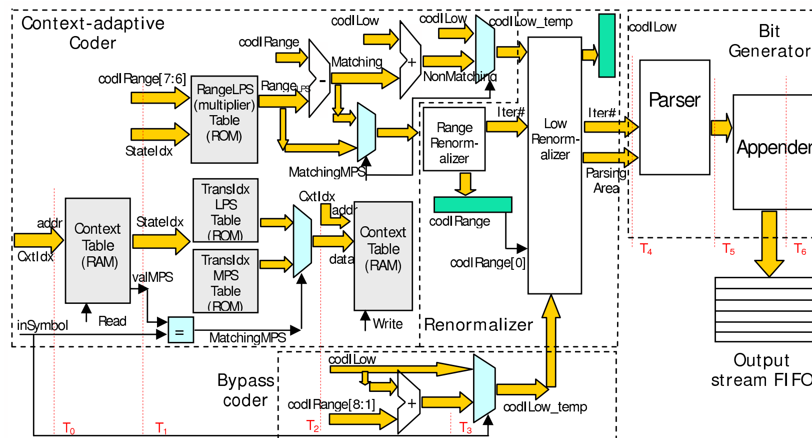

Fig. 3. A high-level model of the proposed architecture.

resolved outstanding bits/words takes several cycles to write to FIFO while renormalization is generating a burst of new bits at the same time. A stall circuit will block coding of further input symbols till intermediate buffer has enough space to accommodate the parsed bits.

The Parser block receives up to nine bits of the *parsing area*. It also needs to know if the last bit currently in the intermediate buffer is an outstanding bit increasing the parser's inputs to ten. A further improvement is made by realizing that the maximum number of renormalization iterations (*iter*) is seven instead of eight because *codIRange* sent to renormalization can never go below 2 as per the multiplier table (*RangeLPS*) defined in [1]. This reduces the total number of parser's inputs to nine bits. The parser generates up to 7 internally resolved bits and two 3-bits start/end pointers to indicate the position inside the parsed area where unresolvable outstanding area starts or ends.

IV. Implementation

Figure 3 shows the proposed architecture for CABAC excluding the binarizer and its related FIFOs. The bottleneck spans from *codIRange* input of the multiplier ROM table to the outputs of Low Renormalizer block. First, a high-level model of the above architecture was integrated into H.264 reference software (version JM 8.2 [6]) and validity of the architecture was verified using standard test sequences. Then, Verilog implementation of the proposed architecture carried out targeting Altera Startix 1S80 device. The design occupied 10K bits of memory and 1320 logic cells (1.6% of the total cells) of which 18% were occupied by Arithmetic Coder and Renormalizer. The rest was taken by the Bit Generator. Since Stratix family provides only synchronous memory and the context-adaptive coding path requires several memory accesses, the longest path would take several clock cycles. This FPGA design currently achieves a speed of 163 MHz with the longest path of coding and renormalization taking three cycles. The bit Generator employs a three-stage pipeline. This effectively achieves an encoding rate of 54 Mbps which is significant on an FPGA implementation and well above average HDTV rates.

Since the bulk of the design is consumed by the Bit Generator, the effect of reducing the output FIFO width was investigated. Decreasing the FIFO width implies reduction of the intermediate buffer which is twice as long as the FIFO width, and its associated appending logic. If FIFO width is reduced to 16 from the original 32, Bit Generator size will be reduced by 36% which reduces the total design by 30% to 933 logic cells. Reducing the FIFO width decreases the longest sequence of outstanding bits that can be guaranteed without stall on encoding from 96 to 32 even in the worst-case scenario. An ASIC synthesis and simulation using a 0.18 μm generic TSMC technology resulted in a 263 MHz circuit with a power consumption of 48 mW, thus enabling 87 Mbps encoding rate. The circuit occupied a total of 0.423 mm^2 area with a 32-bit output FIFO.

V. Conclusion

A new architecture for high performance CABAC encoding along with results of FPGA and ASIC implementations have been presented. The challenges of an effective renormalization and bit generation were discussed. Ongoing work focusses on further speedup of the arithmetic coding stage to increase throughput. Since bypass coding is much simpler than context-adaptive coding, employing a variable cycle completion for context-adaptive and bypass coding modes can also improve the overall throughput.

References

[1] *ITU-T Recommendation H.264: Advanced video coding for generic audiovisual services*, ITU, May 2003.

[2] D. Marpe, H. Schwarz, and T. Wiegend, "Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620–636, July 2003.

[3] S. Sudharsanan, A. Cohen, "A hardware architecture for a context adaptive binary arithmetic coder," in *Proc. of the SPIE, Embedded Processors for Multimedia & Communications II*, Mar. 2005, pp. 104–112.

[4] R. Osorio, J. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system," in *Proc. Euromicro Symposium on Digital System Design*, 2004, pp. 62–69.

[5] J. Mitchell and W. Pennebaker, *JPEG: Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[6] *H.264/AVC Reference Software*, http://iphome.hhi.de/suehring/tml, ver. JM 8.2, July 2004.