A VLSI Architecture for High Performance CABAC Encoding

Hassan Shojania and Subramania Sudharsanan

Department of Electrical and Computer Engineering
Queen's University, Kingston, ON K7L 3N6, Canada

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 significantly 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 an efficient, pipelined VLSI architecture for CABAC encoding along with an analysis of critical issues. The solution encodes a binary symbol every cycle. An FPGA implementation of the proposed scheme capable of 104 Mbps encoding rate and test results are presented. An ASIC synthesis and simulation for a 0.18 μm process technology indicates that the design is capable of encoding 190 million binary symbols per second using an area of 0.35 mm^2. [*]

Keywords: H.264, CABAC, Arithmetic Coding, VLSI.

1. 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 corresponding to 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 divisions or multiplications. However the complexity of the encoding process in its totality is far higher than the table driven entropy encoding schemes such as Huffman coding.

The CABAC encoding process is also bit serial and multi-bit parallelization as in Huffman type encoding is difficult to achieve. Use of a modern microprocessor for encoding a bit consumes hundreds of cycles per bit.[3,4] For a high definition video encoder working at an average rate of 20 million symbols per second can translate into a multi-giga hertz RISC processor requirement. Such large frequencies may not suit low power devices such as cameras where H.264 is to become a dominant standard. Furthermore, instantaneous symbol rates for such encoders can be significantly higher for multiple reasons: picture type (intra or inter) variations and pipelined or stream-processing architectures with macro-block level granularity.[5,6] Such pipelined architectures are preferred in processors that aim to reduce memory and inter-computational block bandwidth requirements.[6] Additionally, if a motion estimator uses rate constrained motion estimation technique, the CABAC encoding symbol rate requirement can go up significantly higher. Under these possibilities, a highly tuned hardware architecture for CABAC encoding is a better alternative than programmable processor-based solutions.

Email: {shojania, sudha}@ee.queensu.ca

[*]This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, Canadian Microelectronics Corporation and Sun Microsystems, Inc.

Several recent papers have attempted to provide efficient schemes for this problem.[3,7,8] Another paper proposed an efficient binary arithmetic coder with a corresponding VLSI architecture as an alternative to the highly complex CABAC process.[4] The solution however is not compatible with the H.264 standard. The scheme proposed in Ref. 7 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, with FPGA implementation results, that under certain circumstances, the circuit could achieve the speed of single bit encoding for every two clock cycles.[3] One critical step in arithmetic coding is the renormalization of the state registers.[9] The design in Ref. 3 addressed renormalization using a simple and bit serial circuit that affected overall performance. The renormalization solution presented in Ref. 7 is based on a QM-coder implementation.[9] 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 Section 2.3). This problem was addressed in our subsequent work to obtain an encoding rate of 54 million symbols per second.[8] That particular architecture has a three cycle per binary symbol throughput, which is significantly improved in this work.

In this paper, we provide efficient solutions for the arithmetic coder and the renormalizer that guarantee a single cycle performance per binary symbol, and also address a number of issues that help reduce the silicon area while maintaining the coprocessor architecture presented in Ref. 3. The proposed solution is tested using encoder data generated by H.264 reference software[10] for several standard video sequences. The remainder of the paper provides an overview of the problem, discusses existing solutions, details the proposed architecture and implementations. We provide details of two implementations, one based on an Altera FPGA platform and the other for a 0.18 μm ASIC synthesis and simulation with timing, power, and area estimations.

2. 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. For brevity, we shall use the terminology in the international standard document[1] without proper introductions.

2.1. 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. The binarization process consists of several schemes that depend on the syntactical elements[1] and consists of kth order exponential Golomb (EGk), unary, truncated unary, and fixed length coding mechanisms. In addition to these four primary techniques, the CABAC employs the concatenation of these methods. For example, a transform coefficient level is coded with a truncated unary prefix and a 0th order exponential Golomb code suffix. A list of syntax elements and their associated types of binarization is provided in Table 9-24 in the H.264 specification document.[1]

2.2. 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 with the *bin* is retrieved through a 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 Ref. 1). In both cases, references to the multiplier (*RangeLPS*) and next probability state

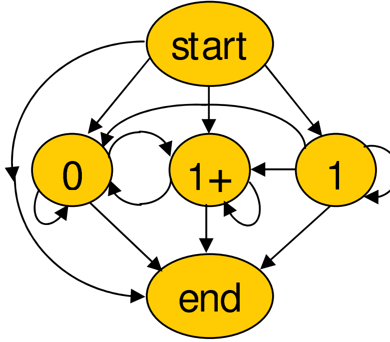

Figure 1. Flow of branches in renormalization iterations shown as a state diagram

(*TransIdxMPS/LPS*) ROMs are 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 (e.g. to context RAM or look-up ROMs) 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 Ref. 1) to make the calculation simpler. At the first glance this process may look to require completely separate logic for its implementation but this is not necessarily the case.

2.3. 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 1 names branches on Fig. 9-8 of Ref. 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 step corresponding to encode of a future symbol that could be several symbols away. As suggested in Ref. 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 with no predetermined bounds. 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.

2.4. Bit Generation

The final stage of bit generation produces 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.

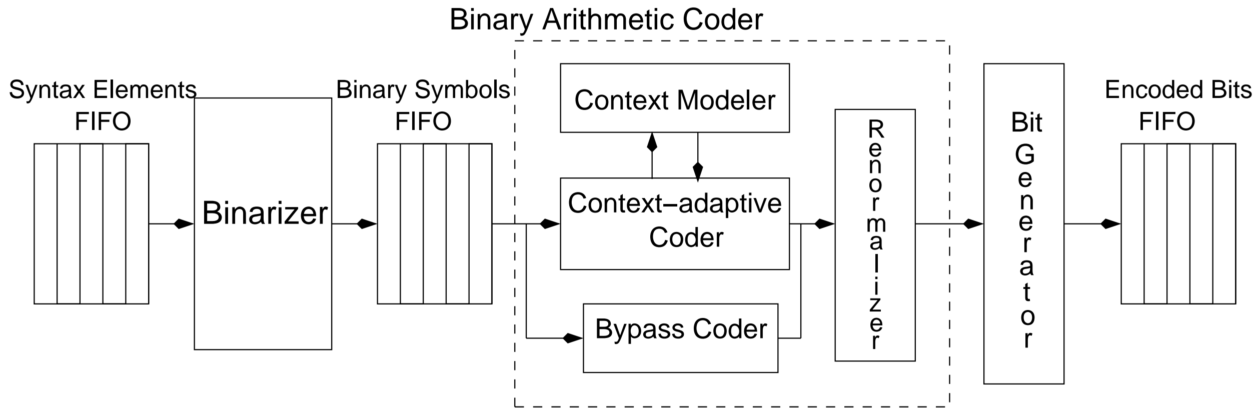

Figure 2. High-level architecture of CABAC encoder.

3. EXISTING SOLUTIONS

The CABAC encoder can be designed as a hardware acceleration block as part of a system-on-chip for encoding H.264 video as shown in Figure 2.[3] A higher level software generates H.264 syntax elements (e.g. motion vector differences, transform coefficients) which are issued to the CABAC block. A preliminary FIFO stores each syntax element accompanied by some side information. Each syntax element is binarized first resulting in one or more binary symbols called *bins*. Each *bin* along with other side information (e.g. encode mode, context index) is placed in a second FIFO to be arithmetically encoded. The binary arithmetic coding phase (either through context-adaptive or bypass mode) is highly serial and can't proceed till the end of the renormalization operation (i.e. till state of arithmetic coder is fully updated). Renormalization stage scales the arithmetic coding state variables *codIRange* and *codILow*, and generates the output bits to be placed in an output FIFO. The higher level software consumes from this FIFO and creates the final video bitstream.

3.1. Binarizer Architecture

The solution proposed in Ref. 3 requires a high-level RISC processor to interface with the binarizer. Syntactical elements and their corresponding binarization methods are sent to the binarizer via the input FIFO. This low-level abstraction helps to simplify the hardware and adds a degree of streamlining, treating all syntax elements in the same manner. The binarizer is composed of six main blocks: the controller, unary, truncated unary, exponential Golomb, fixed length and the context ROM block. The block diagram of the front-end part of the binarizer unit is given in Figure 3.

The control block shown in Figure 3 needs the base context information from the high-level software. Given that many syntactical elements depend on past coded blocks and other global data, a binarizer requires access to several data structures in shared memory in which the overall encoding process uses. Such an approach introduces unnecessary burden on the CABAC unit and also restricts the "pluggable" aspects of the unit in different system-on-chip processors. In order to provide a clean separation between the processor and the CABAC unit to reduce shared memory accesses by the CABAC engine, Ref. 3 defines an interface that has several pieces of information totaling 53 bits corresponding to each syntax element.

The control block contains a state machine that controls the four types of binarizers. It makes the read requests from the input FIFO and instantiates an encoding request of one of the other blocks. The control also sends out a "selector" signal that chooses the output of the encoding block using 4-1 multiplexers. In this way, the entire binarizer has only one output path that is controlled by the multiplexers. An important aspect of the proposed architecture is the method in which the binarizer deals with contexts. In the algorithm specifications, each type of syntax element is assigned a context offset. Also, every bit is assigned a specific "bin index" (*binIdx*). Using Table 9-29 in the H.264 standard specification,[1] one can see how the context index is determined. The context offset finds the row and the *binIdx* chooses the column in the context table. The resulting table entry contains an integer value. This integer is then added to the context offset to obtain the context index. Many of the table entries contain not single numbers but lists of numbers, indicating that a further subclause is defined

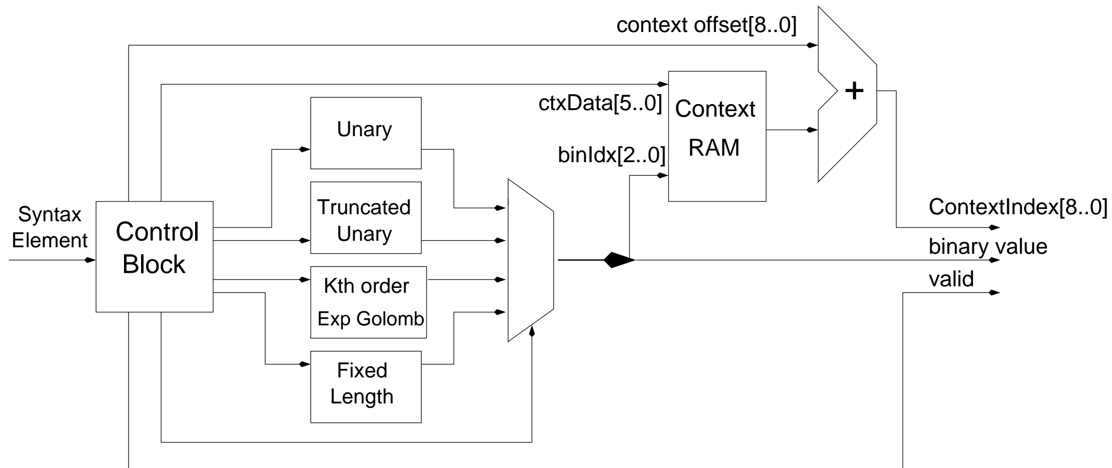

Figure 3. Architecture of the Binarizer

in order to determine the integer to be added. These subclauses look at information from previous frames, or surrounding blocks in order to choose the most appropriate probability model. The results of these subclauses are determined by the control processor in the system, which will interface to the CABAC encoding engine. The information will be given to the binarizer in the form of a single ROM address (*ctxData*). The *ctxData* indicates a row in a ROM such that when combined with *binIdx* results in a unique integer that serves as the base of the context data. In the row indicated by *ctxData* and the column indicated by the *binIdx* lies the cell containing exactly the number which must be added to the context offset in order to obtain the context index. The computed context index and the corresponding binary value are placed in an intermediate FIFO to be drained by the arithmetic encoder.

3.2. Arithmetic Coder

A high performance architecture for arithmetic coding, renormalization and bit generation of CABAC encoding was presented in Ref. 8. The solution divided the renormalization process of Ref. 1 into *codIRange* renormalization, *codILow* renormalization and bit generation. *codIRange* renormalization was simply implemented using a lead zero detector. For renormalizing *codILow*, barrel shift of *codILow* formed a parsing area which was processed by special rules to retrieve the updated *codILow* and generated bits. As Figure 4 shows, update of *codILow* in Low Renormalizer block was completely separated from the more complicated bit generation logic in the Bit Generator block. As a result, the architecture could be broken to three pipeline stages. Encode of the next *bin* can not proceed till updates of *codIRange* and *codILow* associated with encode of the previous *bin* have finished. Also, Ref. 8 modified renormalization of *codILow* for the bypass coding path so a unified solution for portions of bypass and context-adaptive codings can be employed to streamline renormalization and bit generation.

For context-adaptive coding, multiple memory accesses to different tables were required. Since the design was carried out using the Altera Stratix FPGA family that only had synchronous memory blocks available, a multicycle approach with higher frequency was employed instead of using a single cycle approach adjusted according to the longest path. The longest path spans from *RangeLPS* memory access (multiplier lookup table) to end of the Low Renormalizer block marked as *stage 1* on Figure 4. The implementation could be clocked up to 163 MHz where *stage 1*, the bottleneck, takes three cycles (*stage 0* and *stage 2* take one and three cycles respectively). Since a new *bin* could be processed every three cycles, the design effectively achieved an encoding rate of 54 Mbps. Because of the complexity of the Bit Generator block, it was pipelined into three sub-stages here though a none-pipelined but multicycle combinational logic would have worked. A fully pipelined Bit Generator with a single cycle throughput was preferred to allow further improvements in bypass coding as presented in section 4.2.

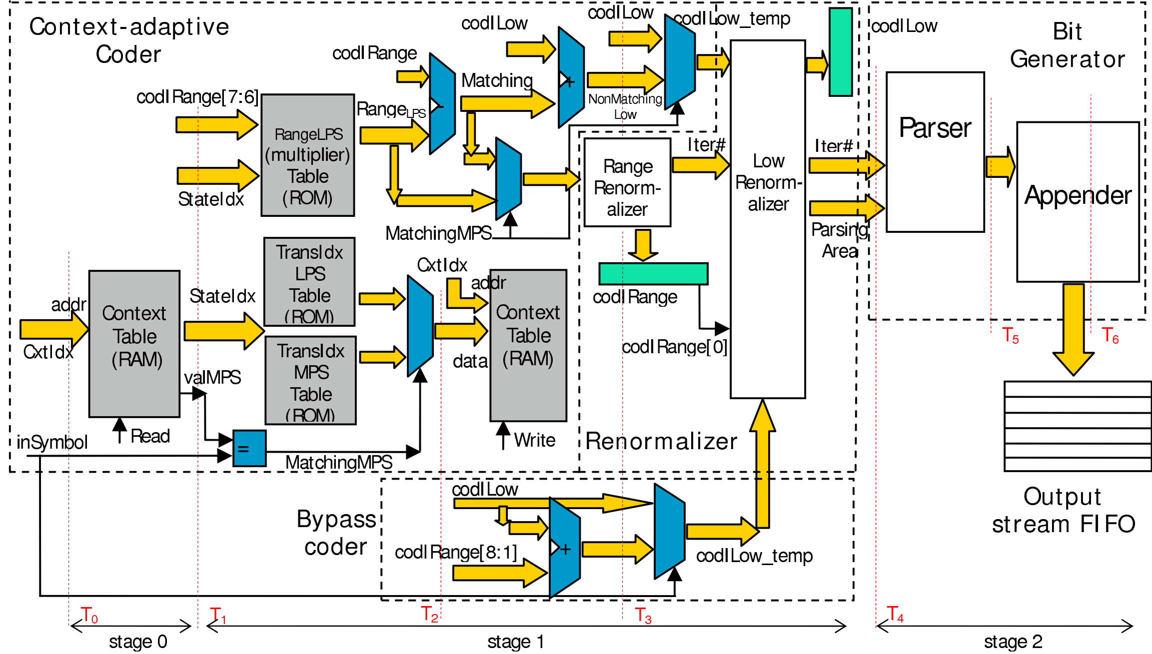

Figure 4. A high-level model of the original architecture[8]

4. PROPOSED ARCHITECTURE

In this section, we discuss several critical aspects of designing a high performance CABAC encoder and provide details of an improved architecture. We first introduce the potential problems in efficient handling of outstanding bits and provide a solution for it. In section 4.2, we propose a method to improve bypass coding for the architecture presented in Ref. 8. In section 4.3, a new pipelined architecture that significantly improves the method in Ref. 8 for arithmetic coding is presented followed by implementation details.

4.1. Efficient Handling of Outstanding Bits

A major architectural challenge for designing a high performance CABAC encoder is that of the ability to handle the outstanding bits. Proper handling of outstanding bits is an important issue both in renormalization and bit generation blocks. Outstanding bits is a pattern of bits (a single one/zero bit followed by *count* number of bits of opposite polarity) where *count* is incremented whenever the middle branch of renormalization is taken (Fig. 9-7 of Ref. 1). The pattern will be known at the *resolve* time when a non-outstanding zero or one bit, the *resolve bit*, is generated. The arrival of a *resolve bit* resets the *count* value after resolving the pending outstanding bits. The problem arises since the polarity of the *resolve* bit is not known till its arrival which can happen within the current renormalization process or in later renormalizations corresponding to encode of future *bins*. This suggests that an "intermediate buffer" is necessary before the output FIFO to accumulate the generated bits. The standard document[1] does not enforce a maximum bound on the size of outstanding bits and only suggests a counter size as big as the whole slice size. This becomes problematic in a hardware implementation as discussed below.

Figure 5 shows a simplified form of the bit generation block. As explained in Ref. 8, renormalization of *codILow* can be done with a barrel left-shift with a shift size equal to *iter* where *iter* is the number of leading zeros of *codIRange*. By inspecting the multiplier table *RangeLPS*, it can be shown that the maximum value for *iter* can never go more than seven though it might seem that value of eight be possible. Not only the shifted-out bits of barrel left-shift of *codILow* (of size *iter*), also the bit at position nine of shifted *codILow* must be considered together as a *parsing area* to be processed based on some parsing rules.[8] The Parser block applies the required rules and resolves the outstanding bits that can be resolved internally. Now different scenarios are possible for the *parsed area* which each needs its own handling where potentially a mix of raw bits and resolved outstanding

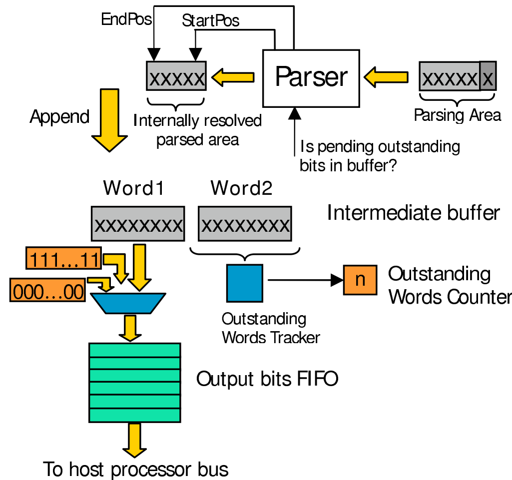

Figure 5. Simplified model of bit generation block

bits are appended to the intermediate buffer. Whenever enough resolved bits are available in the intermediate buffer, they are removed from the buffer and transferred to the output FIFO. Size of the intermediate buffer is intentionally made double of the output FIFO width to handle possible overflow scenarios because of limited bandwidth of access to the output FIFO.

An output FIFO of 32-bit width will be used with a 64-bit intermediate buffer split into first and second words each of 32-bit length. When first word is filled, it is transferred to the output FIFO as shown Figure 5. If outstanding bits with no resolve fills the first word, the second word is used as a place holder since these words cannot yet be sent to the output FIFO. Furthermore, when the second word is completely filled with outstanding bits, an *outstanding words* counter tracks them and the second word is emptied. This will allow buffering of more generated bits in the second word. When the outstanding bits are resolved, the first word is written to the output FIFO and in the following cycles, word count number of words (all either ones or zeros) are written to the output FIFO and the word count is reset. This process could take up several cycles depending on the length of the outstanding bits. During this time, incoming bits from the parser can produce an overflow. To alleviate this, a stall mechanism is employed.

In a hypothetical worst case scenario where encode of successive symbols all generate maximum possible bits of seven, an intermediate buffer size of 64 can handle a maximum of 96 outstanding bits. A solution adds a stall logic in the design to stall the arithmetic encoding engine and stop it from generating further bits when such overflow scenario becomes imminent. The stall continues till enough bits become available in the intermediate buffer to accommodate the next chunk of generated bits. Since there is a trade-off between increasing the intermediate buffer size and incurring stalls, it is necessary to come up with a reasonable target for maximum length of outstanding bits expected to be handled in a stall-free fashion. Figures 6 and 7 respectively show empirical results for length of outstanding bits at resolve time and size of generated bits at renormalization process. This data gathered through modifying the reference software[10] and encoding a few standard test contents. The test contents, the maximum length of outstanding bits sequences and the rate of occurrence of such sequence are demonstrated in Table 1.

Table 1. Size and probability of the longest sequence of outstanding bits

Sequence	Size & number of frames	Longest sequence of outstanding bits	Probability of occurrence of the longest sequence
Foreman	CIF 300	35	$8.01 * 10^{-9}$
Mobile	SD 90	42	$5.79 * 10^{-9}$
Coastguard	CIF 300	41	$4.63 * 10^{-9}$
Football	SD 90	38	$4.37 * 10^{-9}$
Susie (grey)	SD 150	36	$1.57 * 10^{-9}$

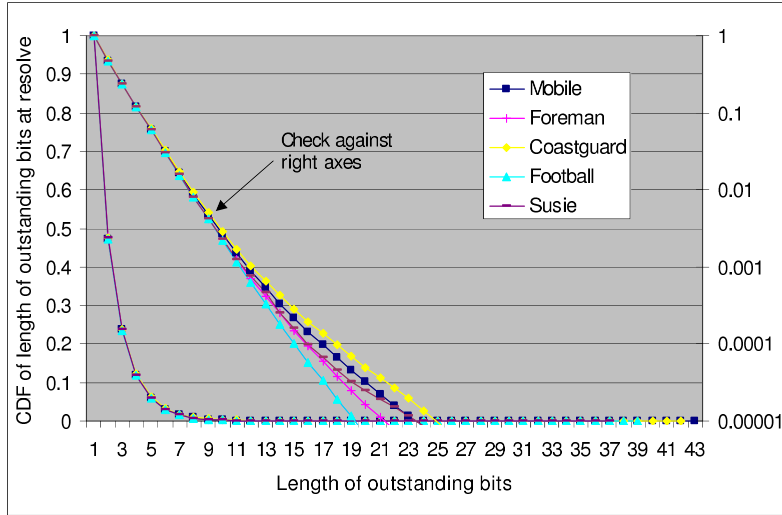

Figure 6. CDF of length of outstanding bits at resolve time

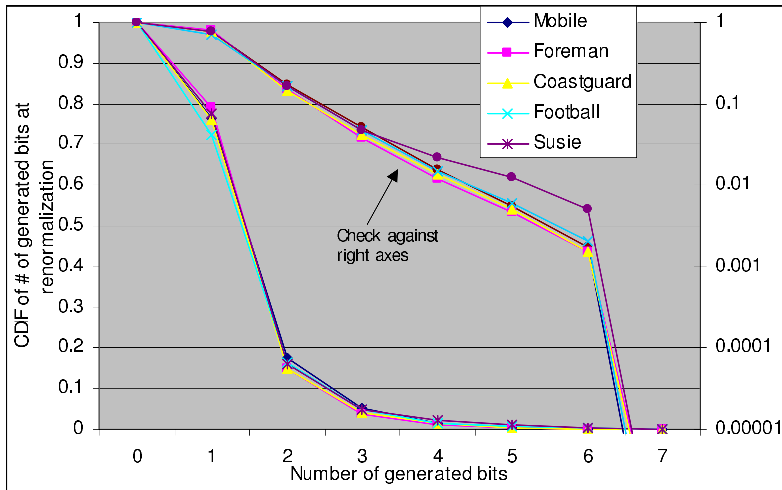

Figure 7. CDF of number of bits generated at renormalization

Figure 6 reflects a cumulative distribution function (CDF) for lengths of outstanding bit sequences from zero to the maximum possible value for different video contents. Since the frequency of longer sequences goes down sharply, the graphs are also shown in logarithmic scale. As the curves show, the probability of encountering an outstanding sequence of length 12 or more is less than 0.1% while it is less than 0.001% for sequence of 25 or longer. This data suggests that stall-free support for outstanding sequences up to 96 bits provided by a 64-bit intermediate buffer is possibly an overdesign. Figure 7 shows the cumulative distribution of number of bits generated within a single renormalization. The exponential decrease of the graph means that in each renormalization step, probability of generating high number of bits is much lower compared to shorter bits. For all sequences except Susie, less than one percent of renormalizations generate five or more bits. This suggests the worst-case calculation for handling stall-free outstanding sequences can relax its assumption based on statistical data. For example, an intermediate buffer size of 32 can handle the Mobile content that has a longest sequence of size 42 without stall. Based on the worst-case calculation, stall-free operation for outstanding sequences longer than 32 can not be guaranteed with an intermediate buffer size of 32. We provide experimental results comparing the hardware complexity of different buffer sizes in section 4.4.

Table 2. Statistics for Bypass Coded Binary Symbols

Sequence	Size & number of frames	% of bins coded in bypass mode	% of bypass coded bins followed by a context coded bin
Foreman	CIF 300	12.6	77.0
Mobile	SD 90	14.6	77.4
Coastguard	CIF 300	13.8	76.1
Football	SD 90	13.8	81.2
Susie (grey)	SD 150	8.9	77.5
Average	-	12.74	77.8

4.2. Improved Bypass Coding

Bypass coding is a much simpler operation than context-adaptive coding as it does not require access to the probability model. By tweaking bypass coding, Ref. 8 simplified it by employing generic renormalizer and bit generation blocks for both coding paths. By proper arrangement, bypass coding can finish within a single cycle rather than the original three cycles. As a result, the *bin* issue logic can be improved to issue the next *bin* based on completion time of the encode of the previous bin, i.e. a completion time of three cycles for a context-adaptive coded *bin* and of one for bypass coded *bin*. By modifying H.264 reference software,[10] frequency of binary symbols arithmetically encoded in bypass mode was studied with encoding several standard test sequences. Table 2 shows this result. As the second column shows, almost 13% of the total coded *bin*s are bypass coded. Since bypass coding can be issued every cycle, an improvement of two cycles per each bypass-coded *bin* is made. Considering the frequency of bypass-coded *bin*s, this results in an average throughput of $0.87 * 3 + 0.13 * 1 = 2.74$ cycles per *bin* which is equivalent to 9.5% improvement over the original three cycles throughput.

Even a further improvement can be made by issuing two successive *bin*s in a single cycle when the first *bin* is bypass coded and the second bin is context-adaptive coded. This is possible since bypass coding does not change *codIRange*. Only *codILow* is updated which is not used in the first cycle of the subsequent context-adaptive coding. Since 77.8% of bypass coded *bin*s are followed by an arithmetic coded *bin*, this means that in average 77.8% of bypass coded binary symbols effectively will not consume any issue slot. This results in an average throughput of $0.87 * 3 + 0.13 * (0.778 * 0 + 0.222 * 1) = 2.64$ cycles per *bin* which is a 13.6% improvement over the original three cycles throughput (equivalent to 61Mbps encoding rate when compared to Ref.[8] result). The pipelined implementation of Bit Generator block of Ref. 8 allows implementation of above schemes as now the Bit Generator is potentially fed with a new data every cycle rather than every three cycles as it was the case before.

4.3. Fully Pipelined Arithmetic Coding

The earlier steps of context-adaptive coding in proposed architecture of Ref. 8 involves calculation of the new *codIRange* followed by its renormalization while later calculations are for update of *codILow* (Figure 4). This suggests that a fully pipelined implementation of arithmetic coding could be possible if all references to *codIRange* can be limited to one cycle while another cycle is dedicated to use and update of *codILow*. Note that it is not efficient to update both *codIRange* and *codILow* in the same cycle as dependency of *codILow* update on *codIRange* update would make the cycle too long. On the other hand, the read access to the multiplier ROM table at the first step of *codIRange* update is lengthening the latency of *codIRange* calculation. A technique similar to Ref. 7, re-arranges the multiplier table to read all four possible table entries in an earlier cycle without using *codIRange* and selects the desired entry through a multiplexer controlled by *codIRange* in the following cycle. This approach replaces the delay of memory access in *stage 2* with a smaller multiplexer delay allowing a higher clock rate. The whole arithmetic coding block can be re-arranged to allow a fully pipelined approach as shown in Figure 8.

A four-stage pipeline manages arithmetic coding block: *stage 0* reads the probability model from the context RAM. *stage 1* reads the multiplier entries from *RangeLPS* ROM and the next probability states from *TransIdx* ROMs. *stage 2* calculates the new *codIRange* value and also writes the updated probability model back to the context RAM. *stage 3* calculates the updated and renormalized *codILow* for both context-adaptive and bypass modes. This approach requires proper handling of some new issues as well. First, a single ported RAM for

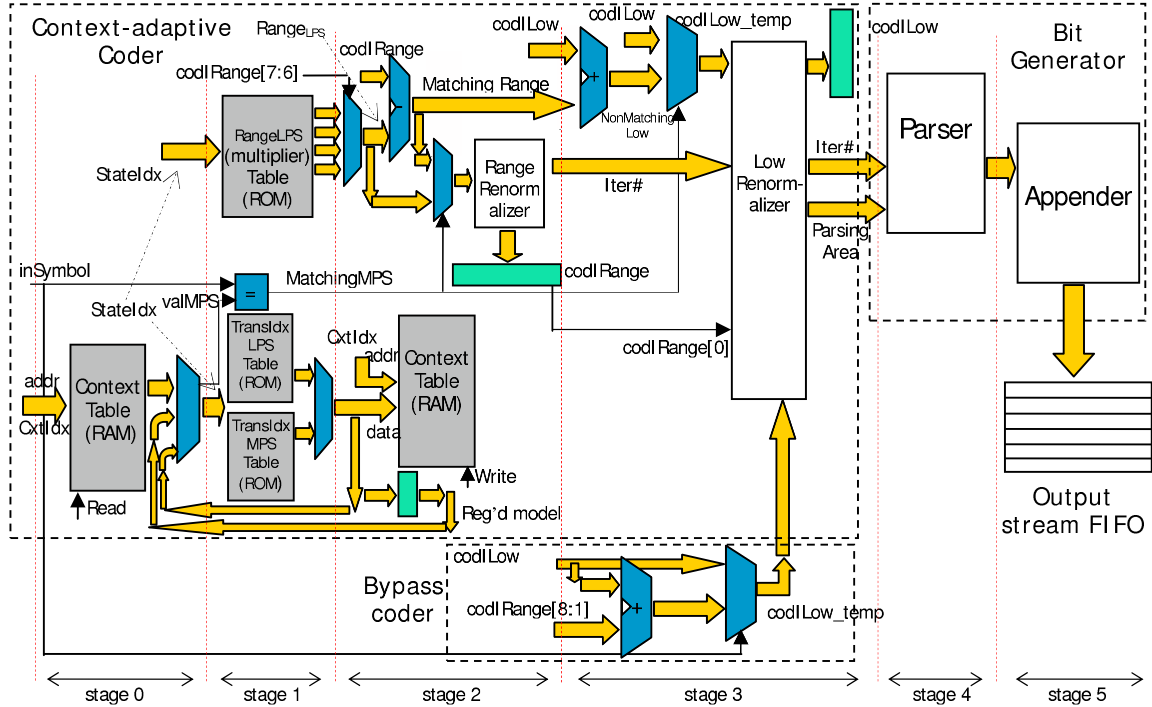

Figure 8. A high-level model of the improved fully pipelined architecture

context table was enough in the original architecture[8] as there was no overlap between read and write accesses to the context RAM for two successive symbols. But in the new approach, all stages are active at every cycle so a dual-ported context RAM is required to carry out concurrent read and write accesses. Another issue arises when successive context-based coded *bins* are encoded using the same probability model (i.e. with equal context index). This requires data forwarding from the updated probability model written to the context RAM within the last two cycles (as read and write are two cycles apart) to *stage 0* of the pipeline as shown in Figure 8. Now a multiplexer selects between the forwarded data and the data read from context RAM which is potentially out of date as that entry might being updated at the same time in *stage 2*. This is similar to bypassing or forwarding in standard processor design.

4.4. Implementation Results

Several implementation exercises were carried out to test the proposed architecture. An implementation of the described architecture on Altera's Stratix 1S80 FPGA achieved a speed of 104 MHz which means 104 million binary symbols per second can be encoded. This encoding rate is almost twice as that of our previous work[8] due to the fully pipelined approach. Number of logic cells for arithmetic coding and renormalization blocks increased to 309 from the previous 237 cells because of the new logic elements of $Range_{LPS}$ multiplexer, data forwarding logic and the extra registers between the pipelining stages. The design however now runs at a lower speed of 104 MHz compared to 163 MHz for the previous non-pipelined, multi-cycle architecture. The Appender block of Bit Generator does not need a two-stage pipeline anymore and can be done in a single stage. This reduces size of the bit generator block with a 64-bit intermediate buffer to 681 from the original 1082 cells. As a result, the design size, excluding memories, is reduced by 18% from 1320 to 1084 logic cells. The memory size remains unchanged at 10 Kbits. The longest path is now *stage 2* of the pipeline for calculation of *codIRange* as it was expected. The proposed fully pipelined design clearly outperforms the original design.

A 0.18 μm ASIC design of the above architecture was carried out using Synopsys's Design Compiler and Virage memory compiler for a generic standard cell TSMC process. The design was simulated using ModelSim. Based on synthesis results, the design runs at 155 MHz occupying an area of 0.321 mm^2 where the memories take 76.6% of it. Here the Appender block, *stage 5* of the pipeline, of Bit Generator becomes the bottleneck and

by breaking it into two stages, the speed can further be improved to 190 MHz with an area increase to 0.355 mm^2 of which 69.3% is taken by the memory elements. The estimated power consumption is 28 mW.

We also conducted experiments to evaluate the effect of intermediate buffer size on the design size. If the buffer size is reduced to 32 bits using a 16-bit FIFO interface from a size of 64 bits, the FPGA design sees a reduction of 26% in number of logic cells with the memory elements staying constant at 10 Kbits. The ASIC design is reduced in area by only 7.5% as the major portion of the design area is taken by the memories. Depending on the probability of the event of very long length outstanding bits, a designer could choose either a 32-bit or a 64-bit intermediate buffer. For our standard test contents, a 32-bit intermediate buffer was satisfactory as no stall case was detected.

5. CONCLUSION

We have presented a comprehensive CABAC encoding engine addressing issues related to binarizer, arithmetic coding, and bit generation. The proposed solution provides significant improvement to our previous work using a fully pipelined approach. The fully pipelined design achieves an encoding rate of 104 Mbps on an Altera FPGA platform. Synthesis and simulation results for a 0.18 μm ASIC design showed that the design is capable of encoding up to 190 million symbols per second. The design issues related to outstanding bits were discussed in detail and related empirical data from real test contents presented. Work in progress looks into integrating the CABAC engine as part of an H.264 encode system-on-chip.

REFERENCES

1. ITU, *ITU-T Recommendation H.264: Advanced video coding for generic audiovisual services*, 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* **13**, pp. 620–636, July 2003.
3. S. Sudharsanan and A. Cohen, "A hardware architecture for a context adaptive binary arithmetic coder," in *Proc. of the SPIE, Embedded Processors for Multimedia & Communications II*, pp. 104–112, Mar. 2005.
4. J. L. Núñez and V. A. Chouliaras, "High-performance arithmetic coding VLSI macro for the H.264 video compression standard," *IEEE Transactions on Consumer Electronics* **51**, pp. 144–151, Feb. 2005.
5. U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *IEEE Computer* , pp. 54–62, Aug. 2003.
6. Tung-Chien Chen, Yu-Wen Huang, and Liang-Gee Chen, "Analysis and design of macroblock pipeline for H.264/AVC VLSI architecture," in *Proc. International Symposium on Circuits and Systems*, **II**, pp. 273–276, 2004.
7. R. Osorio and J. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system," in *Proc. Euromicro Symposium on Digital System Design*, pp. 62–69, 2004.
8. H. Shojania and S. Sudharsanan, "A high performance CABAC encoder," in *Proc. of the 3rd International IEEE Northeast Workshop on Circuits and Systems (NEWCAS'05)*, June 2005.
9. J. Mitchell and W. Pennebaker, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.
10. ITU, *H.264/AVC Reference Software.* http://iphome.hhi.de/suehring/tml, ver. JM 8.2, July 2004.