A LOW COST CONTEXT ADAPTIVE ARITHMETIC CODER FOR H.264/MPEG-4 AVC VIDEO CODING

Jian-Long Chen, Yu-Kun Lin and Tian-Sheuan Chang, Member, IEEE
Dept. Electronics Engineering, National Chiao-Tung University
Hsinchu, Taiwan, R.O.C.
{yklin,tschang}@twins.ee.nctu.edu.tw

Abstract

This paper presents a fast and low cost context adaptive binary arithmetic encoder for H.264/MPEG-4 AVC video coding standard through both algorithm level and architecture level optimizations. First in the algorithm level, we process the binarization and context generation in parallel to reduce the encoding iteration cycles to three or four cycles from five cycles in the previous design. Second, in the architecture level, we reduce the cycles of renormalization loops by employing one-skipping and bit-parallelism, and save hardware cost of arithmetic coder by merging three different modes. The implemented design shows that it can achieve the 333MHz frequency with only 13.3K gate count.

Index Terms— *CABAC, H.264*

1 INTRODUCTION

H.264/MPEG-4 AVC is the latest video compression standard that achieves the same video quality with almost half of the bit rate than previous video coding standards [1]. In which, the entropy coder, Context-based Adaptive Binary Arithmetic Coding (CABAC), plays an important role and can save, 9%~14% of bit rate in typical broadcast applications [2]. However, the design and implementation of the CABAC is difficult due to its inherent bit-serial nature. The coding result of one bit often has a direct effect on the coding process of the successive bits. Thus, it is hard to use parallel and pipeline techniques to enhance the speed of CABAC, and thus becomes the speed bottleneck.

Though many arithmetic coding architectures have been proposed, few CABAC designs have been proposed [3][4][5]. In [3], they proposed several software optimization methods to reduce the CABAC loop iteration into five cycles. In [4], their design focuses on the arithmetic coding of CABAC and uses the software to implement the rest of CABAC flow. In [5], they use multi-symbol coding to speedup the process.

In this paper, we optimize the CABAC design through both algorithm and architecture levels. This paper has three contributions. First, we rearrange the whole CABAC algorithm to have more parallelism such that the execution cycle is reduced to three or four cycles for one iteration from five cycles as in [3]. Second, in the architecture we explore its characteristics to optimize the architecture design of whole CABAC process, including the table reduction for binarization, simplified MPS transition for context modeling, and the optimized arithmetic coding. Third, the optimized arithmetic coding merges three modes into one, and adopts one-skipping and bit-parallel method to speedup renormalization loop. The presented design can achieve high throughput but only with low gate count.

The organization of this paper is as follows. In Section II, we will briefly describe the operation of CABAC in H.264. In Section III, we will show how to rearrange the algorithm and optimize the architecture design. Then, in Section IV, we will show the implementation results and comparison. Finally, a conclusion will be made in Section V.

2 OVERVIER OF CABAC IN H.264

CABAC is used as one of the entropy coding method for H.264 video coding that is consisted of three stages: *binarization*, *context modeling* and *arithmetic coding* (AC). First, a given non-binary value syntax element will pass to *binarization* to form a uniquely bin-string. Second, except for suffix of syntax element motion vector and level information, all of bins from *binarization* will enter into decision mode, and a probability model will be selected to assign context model. The selection of probability models depends on previously encoded syntax elements or bins. After receiving bin and context, AC can encode and output the compressed data directly. AC consists of two sub-engines and is classified into three modes. These two engines are called decision coding engine and bypass coding engine, while the three modes are: (1) "decision" mode" that includes adaptive probability models and interval maintainer, (2) "bypass" mode for fast encoding of symbols, (3) "termination" mode for ending of encoding.

2.1. Binarization

For a given non-binary valued syntax element, H.264/AVC adopts four schemes to do binarization. Such fours schemes are: (1) unary, (2) truncated unary (TU), (3)

concatenated unary/k-th order Exp-Golomb (UEGk), and (4) the fixed-length. They are constructed as follows.

(1) The unary code word consists of x "1" bits plus a terminating "0" bit for a given unsigned integer x,

(2) For truncated unary code, unary code is used only when x < *cMax*. If x=*cMax*, the terminating "0" bit is neglected.

(3) A UEGk bin-string is a concatenation of a prefix bit string with TU and a suffix bit string with Exp-Golomb code.

(4) The FL codeword of x is simply x with a fixed (minimum) number $FL_{bits}=\log_2(cMax+1)$ of bits.

2.2. Context Modeling

In the Context Modeling, the encoder should calculate context index (*ctxIdx*) from 0 to 459 With *ctxIdx* as memory address, it can get probability state (pStateIdx) and Most Probable Symbol (MPS) from context table. The pStateIdx is in range from 0 to 63, and MPS is either 0 or 1. CABAC provides two equations to calculate *ctxIdx*. Except for syntax element *coded_block_flag*, last*_significan_flag, significan_flag* and *coeff_abs_level_minus1*, eq (1) is used for calculating ctxIdx. Otherwise, eq (2) is used.

$$ctxIdx = ctxIdxOffset + ctxIdxInc \quad (1)$$

$$ctxIdx = ctxIdxOffset + ctxIdxInc + ctxCatOffset \quad (2)$$

In (1) and (2), both of *ctxIdxOffset* and *ctxCatOffset* are constant for calculating *ctxIdx*. The *ctxIdxInc* is calculated from the information of neighbor macroblock

2.3. Arithmetic Coding

Fig. 1 shows the flow diagram of AC encoding for a given bin value, *binVal*, in the Decision mode. AC is consisted of three parts, (1) Interval Maintainer, (2) Probability Updating and (3) Renormalization.

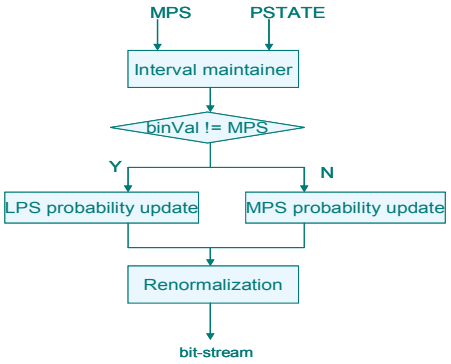

Fig. 1 Flow diagram of arithmetic coding

3 THE PROPOSED FLOW AND ARCHITECTURE

3.1 The Proposed Algorithm Flow and Architecture

In [3], they rearrange the overall CABAC flow into four stages as shown in Fig. 2(a). The number in front of each block means the coding order. With help of software analysis, first, we find that *ctxIdx* calculation is depended on previous *binVal* not current ones. So, we can process binarization and context generation in parallel. Secondly, to read *pstateIdx* and *MPS* from context memory and update them at the same time, a dual-port memory is adopted here to increase encoding speed. With this approach, the encoding iteration can be reduced from 5 to 3-4 cycles as shown in Fig. 2(b) and architecture is easier to be pipelined into three stages as in Fig. 3.

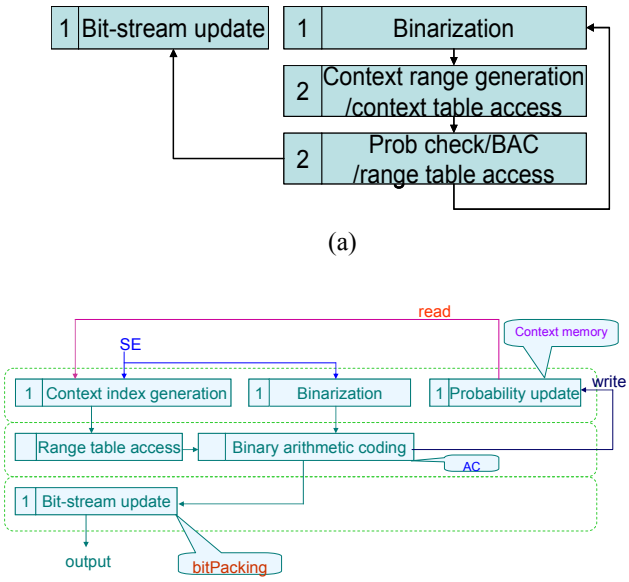

Fig. 2 (a) Original serial schedule of CABAC. (b) Modified parallel algorithm for CABAC

In the first stage of CABAC as in Fig.3, the binarization stage will output the bin-string to second stage and context memory will be updated and output *ctxIdx* for AC at the same time. However, if the ctxIdx are the same for the successive processing, a stall signal should be added to avoid *pstateIdx* be read out before updating. This is the reason why proposed iteration is 4. The second stage is AC, it takes responsibility for calculating interval and output bit-stream. The last stage is FIFO, it collects data from AC. Because output wordlength of bit-stream varies from 0 to 2 Bytes, a FIFO is needed.

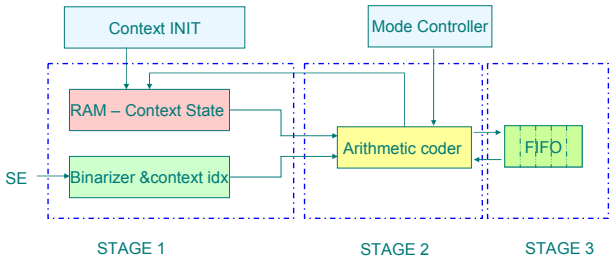

Fig. 3 Pipelined CABAC encoding flow

3.2 Architecture of Binarization

In the binarization stage, although there are four schemes, it can be simply reduced into two types. In which, we classify U, TU and FL schemes into the table based type because they are easier to be realized by combinational logic, On the other hand, the table based UEGk will cost a lot due to the large table. To minimize the table cost, we use the arithmetic method to calculate it by adapting the table partition introduced in [2]. Thus, we use the parameter "base" to find the partition block and a carry save adder (CSA) to calculate its suffix. The proposed architecture is showed in Fig. 4.

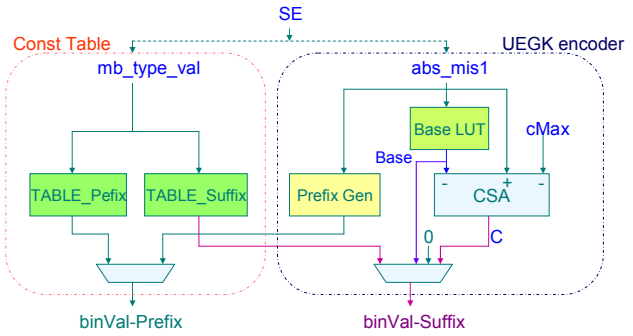

Fig. 4 Architecture of Binarization

3.3 Architecture of Context Modeling

The architecture of context modeling is showed as Fig. 5. As mentioned above, we adopted a dual-port memory for context memory. Besides, the probability updating is extracted from AC, because it depends on values of *pstateIdx* and *MPS*, but does not depend on *codIRange* and *codILow*. Further more, transition table of *MPS* is reduced into simple one by its regular characteristic.

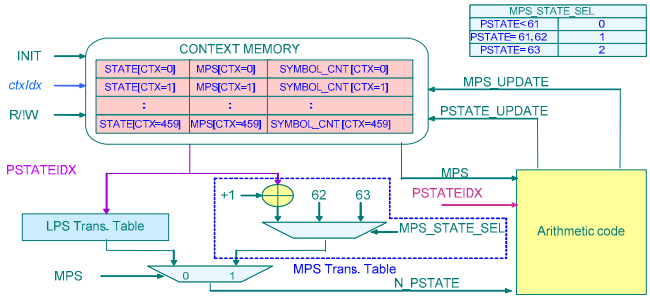

Fig. 5 Architecture of Context Modeling

3.4 Architecture of AC

Fig. 6 showed the architecture of AC. There are two loops in AC [1]. One is controlled by *codIRange* and the other one is controlled by *bitsOutStanding*. To speed up the first loop, we skip the successive one by the Leading-Zero Detector (LZD) and Barrel-shifter to generate new interval. At the same time the output of LZD will be sent to FSM to calculate the renormalization. The idea for the second loop speedup is by bit-parallelism as described below.

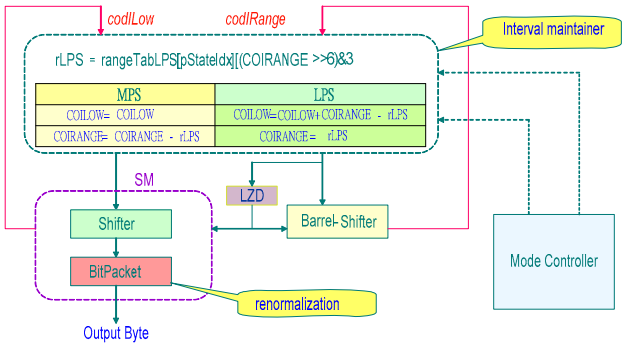

Fig. 6 Architecture of AC

3.4.1 *Interval maintainer in AC*

For the sake to maximize hardware sharing, we analyze *codILow*, and *codIRange* between three modes as shown in Table 1. Here, we can find no matter which mode is selected, *codILow* will involve a three input adder (when *binVal* is equal to *MPS* or equal to 0). Thus we use a carry-save adder to compute new *codILow* to save hardware. This adder also help calculate *codIRange*, since *binVal* is equal to zero and *binVal* is non-equal to zero will not happen at the same time when mode was on termination. After this calculation, the interval will be sent to renormalization. The proposed architecture is showed in Fig. 7.

Table 1 Optimized *codIRange* and *codILow*

codIRange		
DECISION	binVal = MPS(F)	codIRange - rangeTabLPS [pStateIdx][qCodIRangeIdx]
	binVal != MPS(T)	rangeTabLPS [pStateIdx][qCodIRangeIdx]
BYPASS	binVal = 0(F)	codIRange
	binVal != 0(T)	codIRange
TERMINATION	binVal = 0(F)	codIRange-2
	binVal != 0(T)	2

codILow		
DECISION	binVal = MPS(F)	codILow
	binVal != MPS(T)	codILow + codIRange - rangeTabLPS
BYPASS	binVal = 0(F)	codILow << 1
	binVal != 0(T)	(codILow << 1) + codIRange
TERMINATION	binVal = 0(F)	codILow
	binVal != 0(T)	codILow + codIRange -2

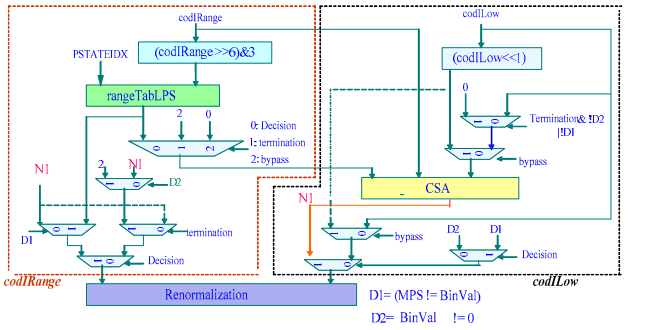

Figure 7 Architecture of Interval Maintainer

3.4.2 Renormalization in AC

When renormalization is happened, [1] uses adder for updating *codILow*. However, with a detailed analysis, we can find that *codILow* is just trying to eliminate its MSB when *codIRange* is less than 0x100. Thus, to minimum the hardware cost, we adopt a FSM instead of adders. After that, *BitsPacking* in the renormalization will receive the bit-stream from AC and pack them in byte. Within *BitsPacking*, *bitsOutstading* has to solve the carry over problem that requires a loop to output data. To break such multi-cycle operations, we use two masks to generate output data in parallel. With such bit-parallelism, we can process this loop in one cycle. Fig. 8 shows the architecture of the renormalization stage.

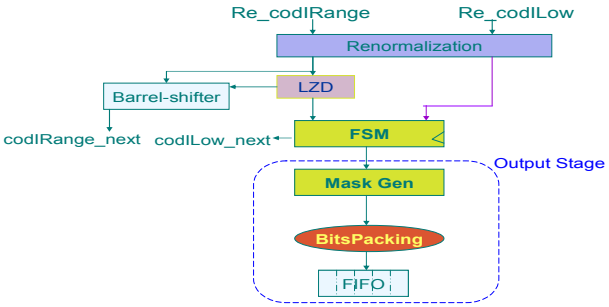

Fig. 8 Architecture of Renormalization

4 EXPERIMENTAL RESULT

The proposed pipeline CABAC has been synthesized from Verilog-HDL description with INCENTIA by using 0.15um CMOS standard cell library. Without considering memory size, the proposed design uses 13.3k gate counts, achieves 333MHz, and takes 1.8 cycles for calculating a bin. Table 2 shows comparisons between conventional and proposed designs for different video size. As an effect of proposed approaches, computing time is saved by more than 9% between different parts of test sequence. Table 3 presents the comparison with previous works. It is obviously that our design uses the least gate count because the modified algorithm in binarization and AC saves large tables and several combinational logics. Moreover, [5] only implements AC, and [4] only implements AC and parts of binarization. In summary, our design is more complete than these two designs.

Table 3. The hardware cost comparison with previous works

	Ours	[4]	[5]
Process	0.15um	0.35um	0.18um
Operating Frequency	333MHz	186MHz	400MHz
Gate Count	13.3K	19.4K	44k

5 CONCLUSION

This paper has described a hardware efficient pipeline CABAC architecture, which exhibits low cost, low latency and high throughput. Experimental results show that our design is adequate for HDTV applications with the proposed optimization.

REFERENCES

[1] *Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification* (ITU-T Rec. H.264/ ISO/ IEC 14496-10 AVC), Mar. 2003

[2] Marpe. D, Schwarz. H, Wiegand. T, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Transactions on Circuits and Systems for Video Technology*, Volume 13, Issue 7, July 2003 p.p 620 – 636.

[3] Ha. V.H.S, W. S. Shim, and J. W. Kim, "Real-time MPEG-4 AVC/H.264 CABAC entropy coder," in *International Conference on Consumer Electronics Digest of Technical Papers*, p.p 255 - 256 , Jan. 8-12, 2005

[4] R. R. Osorio, and J. D. Bruguera, "High-Throughput Architecture for H.264/AVC CABAC Compression System," to be published in *IEEE Transaction on Circuits and Systems for Video Technology*

[5] C. H. Tsai, Y. J. Chen, and L. G. Chen, "Analysis and Architecture Design for Multi-Symbol Arithmetic Encoder in H.264/AVC," in *Proceedings of 2005 SOC Design Conference*, Seoul, Korea, October 2005.

Table 2 Processing Bins of Different Number of Candidates

	HDTV(1280x720) @30		CIF(352x288) @30		QCIF(176x144) @30	
Test sequence	parkrun	Stockholm	foreman	news	container	akiyo
Frame Number	400	500	300	300	300	300
cycles of Conventional	404168707	217089946	46464619	55354588	17288399	12337802
cycles of Proposed	363034386	196122497	41685936	49620501	15475896	11154008
Saving Cycles %	10.18%	9.66%	10.29%	10.36%	10.48%	9.59%