

A High Performance CABAC Decoding Architecture

Wei Yu and Yun He, *Senior Member, IEEE*

Abstract — In this paper, we propose a high performance hardware architecture of CABAC decoder. CABAC is the Context Adaptive Binary Arithmetic Coding used in H.264/AVC video standard, which achieves significant compression enhancement while bringing greater complexity and costs in implementation. The necessity of hardware implementation for real-time CABAC decoders is introduced, and then a fast and cost effective architecture is proposed. The new architecture can achieve decoding speed of averagely 500cycles/macroblock, for typical 4M bit stream of D1 resolution, 30frame/s. An ASIC implementation of the new architecture is carried out in a 0.18 μm silicon technology. The estimated area is 0.3mm² and the critical path is limited within 6.7ns¹.

Index Terms — CABAC decoder, H.264/AVC, video coding, ASIC, hardware implementation.

I. INTRODUCTION

In nowadays, a large number of consumer products such as digital cameras, Personal Digital Assistants, video telephony, portable DVD player as well as storage, broadcast and streaming of standard definition TV are common practice. All those applications demand efficient management of the large amount of video data. This motivated a large body of research in industry as well as in academia to develop advanced video coding technology. H.264/AVC [1] video coding standard developed by the ITU-T/ISO/IEC is the latest international standard developed by ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. The new standard provides gains in compression efficiency of up to 50% over a wide range of bit rates and video resolutions compared with the former standards. However, the decoder complexity is about four times that of MPEG-2 and two times of MPEG-4 Visual Simple Profile. Some of the major tools in H.264/AVC contributing to the computational complexity include variable block sizes, Hadamard transform, RD-Lagrangian optimization, B-frames, 1/4 displacement vector resolution, increasing search range and reference frame number, and CABAC [4].

Context Adaptive Binary Arithmetic Coding (CABAC) [2] is one of the two alternative entropy coding method specified in H.264. Compared to the other technique Context Adaptive Variable Length Coding (CAVLC), CABAC entails an access frequency increase from 25% to 30% with bit rate reduction up to 16% [3].

Hardware acceleration of CABAC decoders is very important for real time applications. Here we show the necessity with some experimental data. We record the number

of bins (bin is a binary value and the basic arithmetic decoding unit in CABAC) and bits in one macroblock (this is abbreviated as “MB” in the following text) for a typical 4M bit stream of D1 resolution, 30frame/s. There are 1350 MBs in one frame, so averagely there are $4 \times 10^6 / (30 \times 1350) \approx 100$ bits per MB.

There are more bins per MB because bins are further compressed before put into bit stream. Statistical results show that the compression ratio is about 1.5. Averagely 30~40 cycles are needed to decode a single bin on DSP. That means for such typical 4M bit stream, averagely about $1.5 \times 100 \times 30 = 4500$ cycles are needed simply to implement the arithmetic decoding task for one MB, while the cycles for other controls are not counted in. This speed is unacceptable for real time applications, where 30 frames of D1 resolution are required to be decoded within 1s with 100MHz clock, i.e. a MB has to be decoded within at most 2000 cycles. So hardware acceleration is necessary for a commercially viable H.264/AVC based video application, especially with increase in image size and quality settings in the future.

In this work, we present a highly efficient architecture for CABAC decoding that takes advantage of the new characters found in CABAC. In this architecture, the whole decoding process inside one MB is implemented by hardware, and software communicates with hardware above the MB layer. The two most time consuming parts of CABAC decoder, i.e. binary arithmetic decoding process and context model maintenance, are carefully designed to make good tradeoff between decoding efficiency and chip cost. Basic arithmetic decoding circuit units are designed to be efficiently shared by all syntax elements. Context models are carefully organized to make sure that during decoding one MB, interruptions for loading context models from RAM and writing back the renewed context models to RAM are kept at the lowest rate.

In section II, we introduce the primary steps of CABAC encoding and decoding process, and point out where the bottleneck of the efficient implementation is. In Section III, we describe the basic scheme of our CABAC decoder architecture. We present an overview of the framework of our architecture, including a high-level description of the building blocks. In this section, we focus on the motivation and the basic principles of our design. A more detailed description of our architecture can be found in section IV, where the implementation is discussed in depth. In Section V, we summarize the performance and main conclusions of our work.

II. CABAC ENCODING/DECODING PROCESS

This section covers the basic principles of CABAC encoding and decoding process. And we briefly introduce the bottleneck of CABAC encoder/decoder implementation.

The CABAC encoding process consists of three elementary steps [10]:

¹ This work is supported by NSFC in grants 69972021 and 60333020.

Wei Yu is with the Department of Electronic Engineering, Tsinghua University, 100084 P.R.China (e-mail: yuwei@video.mdc.tsinghua.edu.cn).

Yun He is with the Department of Electronic Engineering, Tsinghua University, 100084 P.R.China (e-mail: hey@video.mdc.tsinghua.edu.cn).

Contributed Paper

Manuscript received October 13, 2005

0098 3063/05/\$20.00 © 2005 IEEE

1) binarization, which is the unique mapping process of possible values of a syntax element onto a set of bin strings. The syntax element is an element of data represented in the bit stream to be encoded. The bin string is a string of bins, which is an intermediate binary representation of value of syntax elements. Each syntax element is uniquely mapped to a bin string and each bin of the bin string is arithmetically encoded into the bit stream.

2) probability modeling. There are totally 399 context models used in CABAC. Context models are used for the probability estimation, each of which corresponds to a certain bin or several bins of a certain syntax element. Sometimes several context models correspond to the same bin. Context models keep updating during the whole encoding process to keep track of the probability estimation. Before arithmetically encoding one bin, the corresponding context model is selected for the bin and the model is updated after the bin is encoded. Each context model comprises of a pair of two values, a 6-bit state value *state* as the probability index, and a binary value of the most probable symbol *MPS*. The state value is used as an index of the estimated probability value of the least probable symbol “*LPS*” ($LPS = !MPS$).

3) Binary arithmetic coding. The original principle of the binary arithmetic coding is based on recursive interval subdivision of the interval width R [12]. Given the estimation of the probability p_{LPS} of *LPS*, the interval is subdivided into two subintervals: one interval width $rLPS = R \cdot p_{LPS}$ which is associated with the *LPS*, and the other interval width $rMPS = R - rLPS$, which is assigned to the *MPS*. Depending on whether the observed bin to be encoded is *MPS* or *LPS*, the corresponding subinterval is chosen as the new interval. The binary arithmetic coding process keeps updating two registers, i.e. the interval width register R which marks the range of the interval and the code register C which marks the lower bound of the interval.

In practical implementation of binary arithmetic coding, there are two main factors deciding the throughput. One is the multiplication operation of $R \cdot p_{LPS}$ required to perform the interval subdivision, and the other is dynamically estimating probability p_{LPS} . CABAC realizes a binary coder that is related to the Q-coder family [6]-[7]. It inherits the table-based mechanism in Q-coder but achieves much better compression efficiency. In CABAC, approximated multiplication results are pre-stored in a fixed table for the sake of low complexity. Range value is approximated by four quantized values using an equal-partition of the whole range $2^8 \leq R < 2^9$ into four cells. The value of p_{LPS} is approximated by 64 quantized values indexed by the 6-bit state value. With those 2-bit of range used as the quantization index for range value and the 6-bit probability state value used as the quantization index for p_{LPS} value, the approximated subinterval range value $rLPS$ is looked up from a 4×64 2-D table. The probability estimation process is also table-based. State value is updated after encoding each bin by looking up a fixed table.

To use fixed precision integer arithmetic, the interval width register R and the code register C must be renormalized after

encoding each bin. In CABAC, both registers use 9-bit precision. During the renormalization process, the register R is shifted left to guarantee the most significant bit *MSB* of current interval R is always 1.

Actually in CABAC arithmetic encoding implementation, two sub-engines are used: one for bins of regular coding mode as described above, and another so-called bypass coding engine for a fast encoding of bins. For such bins, a default value of 0.5 is used as the estimation of p_{LPS} . The two sorts of bins are referred to as “regular bin” and “bypass bin” in the following text. For more detailed description of CABAC, please refer to [10]-[11].

The CABAC decoding process is the inverse process of CABAC encoding. First, the corresponding context model is selected for the bin to be decoded. Then, the bin is decoded by binary arithmetic decoding engine. During the decoding process, the context model should be updated. Binary arithmetic decoding engine is similar to that of binary arithmetic encoding. The current interval is kept in the 9 bits register *range*, and the position of the input bit stream is kept in the 9 bits register *offset*. Whether the bin is decoded as *MPS* or *LPS* is decided by whether the offset falls into the subinterval of $rLPS$ or $rMPS$. Those two registers are iteratively updated. Binary arithmetic decoding engine is further discussed in more details in section III-A. Finally, bins decoded are post processed to reconstruct the syntax element for output.

The bottleneck for efficient CABAC decoder implementation rests with the context model maintenance and the binary arithmetic decoding process. Context model maintenance demands lots of memory access. For decoding each bin, the corresponding context model has to be loaded and the renewed value has to be written back. The arithmetic decoding engine for one bin is decisive for the critical path. When several engines are concatenated together to achieve higher speed, the critical path is nearly several times of the delay of one such engine. The optimization of the structure of multi-engine is what we will discuss in detail in section IV-A.

III. OVERVIEW OF OUR CABAC DECODING SCHEME

As we have mentioned in the previous section, the bottleneck of efficient CABAC decoder implementation primarily rests with the context model management and the concatenation of the arithmetic decoding engine for one bin. The motivation of our design is to guarantee 1) that one or more bins are decoded per cycle, and bins of each syntax element are decoded consecutively without interruption and 2) that only a few cycles are consumed for the context loading and writing back tasks.

Fig. 1 shows an outline of our CABAC decoder architecture. Software/hardware interface unit in Fig. 1 is the interfacing of the hardware units with the outside software engine. The outside software engine provides bit stream to hardware and reads the decoded syntax elements from the output unit. The hardware units here include all the other units in Fig. 1. The “CABAC Basic Decoding Circuit Units” module is the processing core of the arithmetic decoding process. It

includes high speed decoding engines that can decode several bins per cycle. The “Context Model RAM” stores all the context models. The “Context Model Register Group” module controls a group of registers, and manages their loading context models from RAM, updating models and writing back models to RAM tasks. The “Context Selector” selects the correct context model dependent on the neighboring MB information. The decoded bins are post-processed and reorganized to reconstruct the syntax elements in the module “Output”, and the software reads the decoded syntax elements from this module. The hardware units described above cooperate to accomplish the whole MB decoding task. They also take charge of the task of decoding several syntax elements which are also encoded using CABAC above the MB layer. Those syntax elements include: *mb_skip_flag*, *mb_field_decoding_flag* and *end_of_slice_flag*.

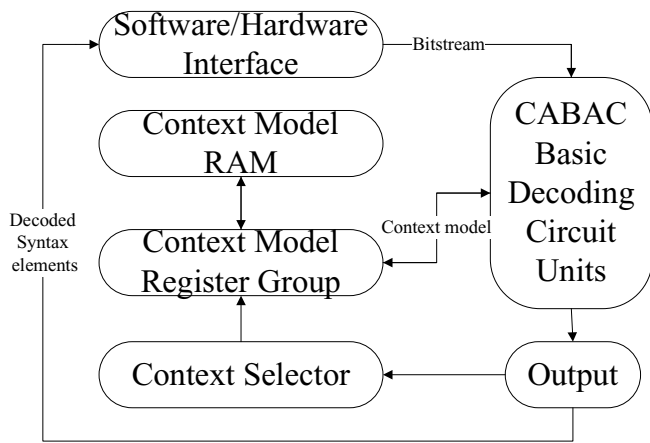


Fig. 1. Framework of CABAC decoder

To well explain the principle of our CABAC decoding architecture, we first introduce the AC decoding engine for one bin in section III-A. Then, in section III-B, we give a general introduction of our CABAC basic decoding circuit units, in which several elementary decoding engines for one bin are combined to form multi-hierarchy decoding tree. In section III-C, we briefly discuss our context model maintenance, where we focus on explaining why we manage context models in that way and how it works efficiently.

A. Binary arithmetic decoding engine for one bin

We have presented principles of the arithmetic encoding and decoding process in the previous section. Here we discuss the procedure in more details for comprehension of our whole decoding architecture. Fig. 2 is a sketch map of binary arithmetic decoding engine for one bin.

As we have said in section II, the arithmetic decoding engine keeps updating two 9-bit registers *range* and *offset* during the whole decoding process. The register *range* keeps track of the width of the current interval while the register *offset* keeps track of the input bit stream. When decoding a bin, *range* is split to two subintervals: *rLPS* corresponding to the estimated probability interval of the *LPS* and *rMPS* corresponding to the estimated probability interval of the *MPS*.

During the encoding process, the value of *rLPS* is read from a fixed 2-D table of 256 bytes, addressed by 2 bits from the *range* value and 6bits from the *state* value. Which subinterval the input bit stream (marked by the *offset*) falls into decides whether the bin is *MPS* or *LPS*.

In Fig. 2, the left-side plot shows the case that *MPS* occurs, where the *offset* is less than *rMPS*. The right-side plot shows the case that *LPS* occurs, where the *offset* is greater than or equal to *rMPS*. The renewal of the *range* and the *offset* are:

$$\begin{aligned} &\text{if } MPS, \begin{cases} range_new = rMPS \\ offset_new = offset \end{cases} \\ &\text{else, } \begin{cases} range_new = rLPS \\ offset_new = offset - rMPS \end{cases} \end{aligned}$$

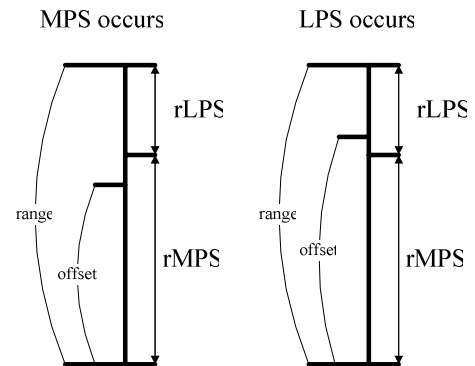


Fig. 2. Arithmetic decoding engine for one bin

To keep the precision of the whole decoding process, *range_new* and *offset_new* have to be renormalized to ensure the most significant bit *MSB* of *range* is always 1. For example, *range_new* is 9'b001010110, *offset_new* is 9'b000110010, during the renormalization process, *range_new* is left shifted two bits so that the *MSB* is 1 and the last two bits are stuffed as 2'b00; *offset_new* is synchronously left shifted two bits and the last two bits are stuffed from the bit stream. In this way, *offset* receives bits from the input bit stream to keep track of the position of the bit stream in the current interval.

The process discussed above is for decoding bins encoded in regular mode, where the *LPS* probability p_{LPS} is estimated by the corresponding context model. For bypass mode, p_{LPS} is estimated as 0.5 and no context model is needed. In such case, theoretically the *offset* should be compared with a value $range/2$ to determine the bin is *MPS* or *LPS*. However in CABAC bypass mode, to keep integer precision during the whole decoding process, the *offset* is left shifted one bit and then receives one bit from bit stream. The new *offset* value is compared with *range* to determine the bin is 1 or 0.

B. Basic decoding circuit units

We have discussed the process of decoding one bin in section III-A. In this section we will introduce how to efficiently organize the decoding engines for one bin to achieve high decoding speed by using characters found in CABAC.

After exploring the binarization modes of different syntax elements and their occurring frequency during the decoding process, we find there are two classes of syntax elements. The syntax elements in the first class contain a few bins and occur only once or a few times in one MB; while the syntax elements in the second class contain several bins and occur tens or hundreds of times in one MB. Syntax elements of the first class contribute only a small part to the arithmetic decoding workload, so in our architecture, one bin is processed per cycle for such kind of syntax elements to decrease the complexity and hardware cost. However to accelerate decoding of the syntax elements in the second class, a new architecture that can decode two regular bins together with one bypass bin in one cycle is proposed. Those syntax elements in the second class include `abs_mvd`, `significant_coeff_flag`, `last_significant_coeff_flag` and `coeff_abs_level_minus1`.

Another character of CABAC is that bins encoded by bypass mode only appear in two cases, either as sign or as Exp-Golomb (EG) code. The EG code can only appear as suffix of the bin string of `abs_mvd` or `coeff_abs_level_minus1`. For bypass bin as sign, only one bypass bin is decoded for each `abs_mvd` and `coeff_abs_level_minus1`. For the second case, only those `abs_mvd` larger than 8 or `coeff_abs_level_minus1` larger than 13 can contain EG code as suffix. Actually such cases are rare occasions, because at least 90% of those two syntax elements have small values and hence do not contain such suffix. So there is no need to specifically accelerate the decoding process for bypass bins.

Based on those characteristics of CABAC described above, we organize the elementary engines for one bin as following: 1) a two-hierarchy decoding tree for decoding two regular bins *RB1* and *RB2* with or without a sign, which is illustrated as Tree-(a) in Fig. 3; and 2) a two-hierarchy decoding tree for decoding two bypass bins *BP1* and *BP2*, which is illustrated as Tree-(b) in Fig. 3.

The following paragraphs explain how Tree-(a) and Tree-(b) are employed to decode all syntax elements.

All syntax elements are decoded one regular bin per cycle, utilizing the first branch of the Tree-(a), except for `abs_mvd`, `coeff_abs_level_minus1`, `significant_coeff_flag`, and `last_significant_coeff_flag`.

Bin string of syntax elements `abs_mvd` and `coeff_abs_level_minus1` comprises prefix and suffix. The prefix is unary code, i.e. a string of 1s plus one bit 0, and it is decoded two regular bins per cycles until reaching the end. If there is no EG code as suffix, the sign in the bypass mode is decoded within the same cycle of the last regular bin of the prefix, utilizing the Tree-(a) with a sign. For example, `coeff_abs_level_minus1` of value -1, the prefix is 10, with no EG code as suffix and one bypass bin 1 representing the sign “-”. The two regular bins “10” are decoded with a bypass bin, using Tree-(a) with a sign. Otherwise, when there is an EG code as suffix, the Tree-(b) is applied to decode the suffix, decoding two bypass bins per cycle until reaching the end.

For `significant_coeff_flag` and `last_significant_coeff_flag`, each of them is one bin and a pair of them is encoded for each residual coefficient of the 4x4 block. The value of the pair includes two cases: 1) the `significant_coeff_flag` is ‘0’,

indicating that the coefficient is zero and no `last_significant_coeff_flag` exists; (2) the `significant_coeff_flag` is ‘1’, indicating that the coefficient is not zero and there is `last_significant_coeff_flag` appended, indicating whether this coefficient is the last non-zero coefficient or not. Those two syntax elements account for a great part of the workload in decoding process, because there are at most 15 pairs of them for each 4x4 block and there are at most sixteen 4x4 blocks inside one MB. So in our architecture, one pair of them is decoded per cycle by using Tree-(a) without sign.

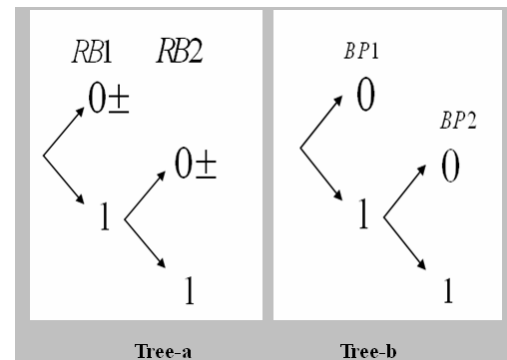


Fig. 3. Two hierarchy decoding tree

Here, we'd like to emphasize one point that multi-hierarchy decoding tree is the most important part that decides the delay of the critical path in hardware implementation. CABAC decoding is essentially a serial process, i.e. to decode several bins in one cycle will bring about several times the critical path for decoding one bin. When several basic arithmetic decoding engines are concatenated together, delay of critical path may include the calculation of $rLPS$ indexed from the fixed 256 bytes table, the subtraction in those decoding engines, the updating process of the context model as well as the supply of the bit stream. Our target is to keep main critical path in 6.7ns using 0.18 μm technology. So, we do much effort to explore the parallelism of multi-hierarchy structure to shorten the critical path. In section IV, we will discuss in detail about the structure of our two-hierarchy decoding tree and pertinent optimization.

C. Context model maintenance

Multi-context model is the crucial technique used in CABAC to achieve high compression ratio. In CABAC, different syntax elements are allocated with different context models, and there are totally 399 context models. For most of the cases, one context model is assigned to specific bin of a syntax element or several consecutive bins of a syntax element; but for some special cases, several context models are assigned to the same bin of a syntax element to increase the compression efficiency. For the special cases mentioned above, one correct context model has to be chosen from several context models, based on the information of the same syntax elements of the adjacent left and top MB/block.

After the correct context model is chosen, it must be loaded and updated to keep track of the probability estimation of the

corresponding bin during the decoding process. To keep up with the high decoding speed of the basic decoding circuit units described in the previous section, i.e. one or more bins are decoded per cycle and the bin string of a syntax element is decoded without interruption, the context models must be accessed, updated and saved at the same speed. So, it's not practical to load the context model from RAM and write the renewed value back to RAM during decoding each bin, because at least two cycles are needed to maintain one context model, which is a considerable speed reduction. Context models may be updated simultaneously if they are stored in registers. However, it will cost larger area non-economically if all context models are stored in registers, because totally 399 registers of seven bits are needed. And there'll be more cost for control of those registers.

Considering that context models used in decoding one MB have its own sequential characteristics, a good tradeoff can be made between the decoding efficiency and the implementation cost. In our design, we divide context models into several groups according to the order of being called. If we divide all syntax elements properly, we can load and write back all the context models in the same group at the same time, and only once for each group during decoding one MB. Each group of context models is kept in the module "Context Model Register Group" in Fig. 1. In our architecture, we divide all context models into 18 groups. With this mechanism, access frequency to the RAM storing context models is greatly reduced, and there is nearly no damage to the decoding speed from this part.

IV. DETAILED DESCRIPTION OF CABAC DECODER

In this section, we discuss our hardware implementation architecture in details. We will present the specific implementation structures for the hardware units described in section III-B and III-C.

A. Basic decoding circuit units

As we have discussed in section III-B, Tree-(a) and Tree-(b) are shared by all syntax elements. In Fig. 4-(a), we show the structure for decoding single regular bin, and in Fig.4-(b) the structure for decoding single bypass bin. Fig. 5 shows the implementation structure of *RB1* concatenating *RB2* of Tree-(a). The "rlpsg_1", "rlpsg_2" and "state_1 renewal and preparation for rlpsg_2" will be talked about later. The concatenation of two bypass bin decoding engines is similar but much simpler without context models.

In practice, we instantiate totally four elementary decoding engines for one bin: two for regular bins *RB1*, *RB2*, and two for bypass bins *BP1*, *BP2*. To realize the sharing of those four elementary engines, a set of control signals is defined to coordinate their work:

1) *dec_mode* (2 bits): *dec_mode* have 3 possible values. *dec_mode* of value '00' stands for decoding regular bins, in which case another control signal *regbinnum* is used to co-decide how many regular bins are to be decoded in one cycle, i.e. whether only the first branch of Tree-(a) is utilized or all two branches are used. The other two values of *dec_mode* are

for decoding EG code. EG code has its own prefix and suffix. The prefix of EG code is unary code, in which case the syntax element ends up when decoding out one bin of value 0. The string length of the suffix of EG code is determined by the value of its prefix. *dec_mode* of value '01' stands for decoding the prefix of EG code, using Tree-(b); *dec_mode* of value '10' stands for decoding the suffix of EG code, which also utilizes Tree-(b), but in which case another control signal *dec_bp_num* is used to co-decide how many bypass bins are to be decoded in this cycle.

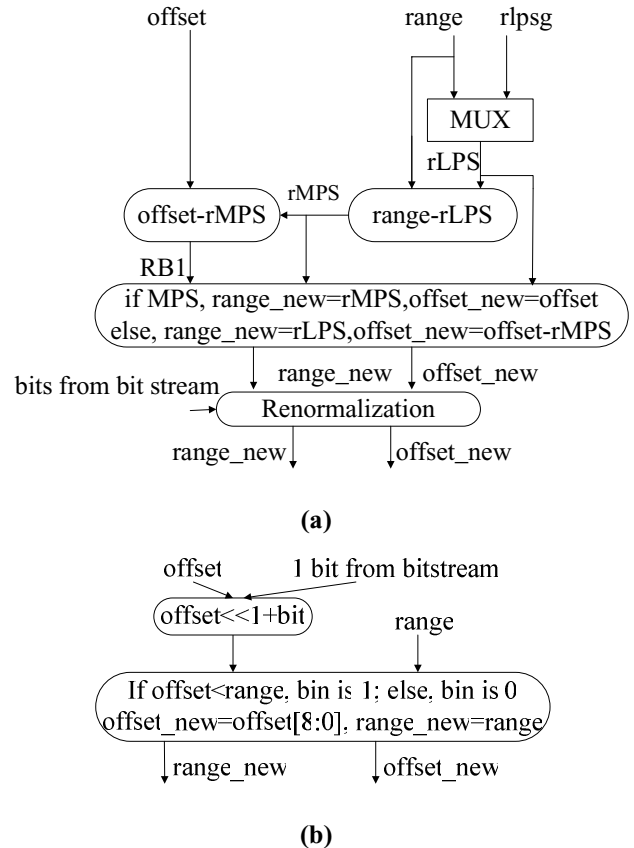


Fig. 4 (a). Single Regular Bin decoding circuit
(b). Single Bypass Bin decoding circuit

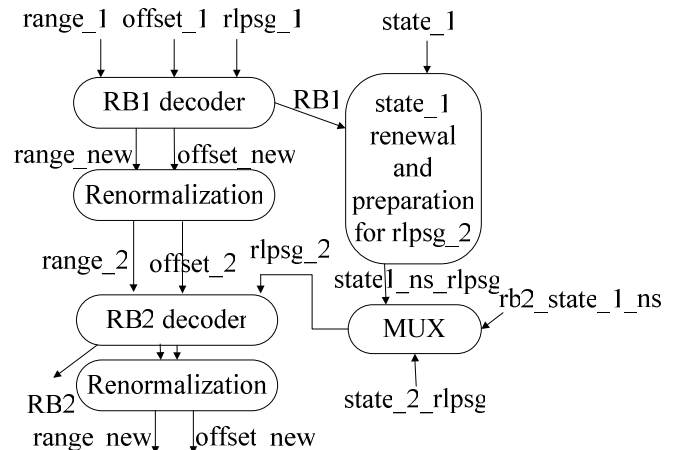


Fig. 5 Concatenation of two regular bin decoding engines

2) *regbinnum* (1 bit): this signal cooperates with the *dec_mode* to decide how many regular bins are to be decoded in one cycle, only when *dec_mode* is '00', this signal is useful. When decoding prefix of *abs_mvd*, *coeff_abs_level_minus1*, *significant_coeff_flag*, and *last_significant_coeff_flag*, *regbinnum* is 0 indicating two branches of Tree-(a) are both working. When decoding the other regular bins, *regbinnum* is 1 indicating only the first branch of Tree-(a) is working.

3) *rb2_state_1_ns* (1 bit): this signal indicates if decoding the second regular bin uses the renewed context model of the first regular bin or another context model. This signal is useful only when Tree-(a) is in use.

4) *sign1_exist* (1 bit), *sign2_exist* (1 bit): these two signals are useful only when Tree-a is in use. They decide whether to decode a bypass bin as sign after the regular bin(s) are decoded. They are used when decoding *abs_mvd* and *coeff_abs_level_minus1*. *Coeff_abs_level_minus1* contributes most to the decoding workload for I frames. *Coeff_abs_level_minus1* and *abs_mvd* contribute most to the decoding workload for P/B frames. Statistical data shows that small value less than 10 of these two syntax elements accounts for up to 99% of total. Small value less than 6 accounts for about 90% of total. So, with the sign decoded with those regular bins in the same cycle, decoding speed is greatly enhanced. We can decode those syntax elements of value less than 2 in one cycle, of value less than 4 in two cycles, of value less than 6 in three cycles. And this can be achieved without bringing extra cost or prolonging the critical path.

5) *dec_bp_binnum* (1 bit): this signal cooperates with *dec_mode* to decide how many bypass bins are to be decoded in one cycle, only when *dec_mode* is '10' (for EG code suffix), this signal is useful.

With the above control signals, we are now able to decode all syntax elements sharing four elementary decoding engines *RB1*, *RB2*, *BP1* and *BP2*. Fig. 6 illustrates our kernel "Basic decoding circuit units", in which how control signals works with four decoding engines is shown. The "ctx1" and "ctx2" in Fig. 6 are two context models for *RB1* and *RB2*.

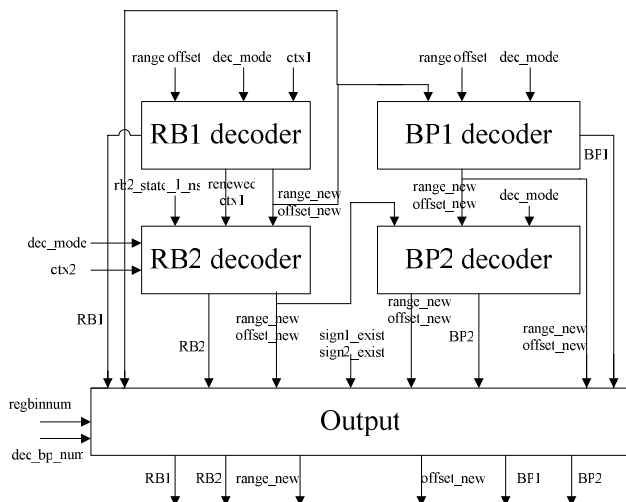


Fig. 6 Basic decoding circuit units

We have mentioned in section III-B that multi-hierarchy decoding tree structures are the key factor for the delay of the critical path. To restrict the delay of the critical path within 6.7ns, we optimized the structure of concatenating the decoding engines as well as the basic decoding engine for one bin. Here are the main points of our architecture:

1) The value of *rLPS* is needed for decoding one bin. In CABAC, this value is indexed from a 256 bytes table, and the index consists of 2 bits from *range* and 6 bits *state* value of context model. To save time of calculating *rLPS* for decoding *RB1*, four different values of *rLPS* addressed by *state* ("rlpsg_1" in Fig. 5) are pre-fetched in the previous cycle, so that in the current decoding cycle *rLPS* is selected by a multiplexer only using 2 bits of *range* (see Fig. 5). In the following text, "rlpsg" is specified for a group of four *rLPS* indexed only by the *state* value.

2) Calculation of four different values of *rLPS* (rlpsg_2) needed for decoding *RB2* is carried out in parallel with the decoding procedure of *RB1*. This corresponds to the module "state_1 renewal and for preparation for rlpsg_2" in Fig. 5. The module is further illustrated in Fig. 7. The module is used when the context model for *RB2* is what's renewed from *RB1*. It will result in quite a long path if the calculation of rlpsg_2 is carried out after *RB1* is decoded out and *state* is updated. So we calculate all two possible rlpsg (*state_ns_ifLPS_rlpsg*, *state_ns_ifMPS_rlpsg* in Fig. 7) for *RB2*. *state_ns_ifLPS_rlpsg* is the rlpsg indexed from the renewed *state* value if *RB1* is *LPS*, and *state_ns_ifMPS_rlpsg* is the rlpsg indexed from the renewed *state* value if *RB1* is *MPS*. This can be done in parallel with decoding *RB1*. When *RB1* is decoded out, rlpsg prepared for *RB2* is directly chosen from *state_ns_ifLPS_rlpsg* and *state_ns_ifMPS_rlpsg* according to whether *RB1* is *MPS* or *LPS*. Synthetic results show that with this technique, about 15% area is saved when the time slack is met and area cost is optimized.

3) *MSB* (Most Significant Bit) of the result of the subtraction *offset - rMPS* is used to determine if *MPS* or *LPS* happens (see Fig. 4-(a)). This saves a 9 bits comparator in the basic decoding engine for one bin.

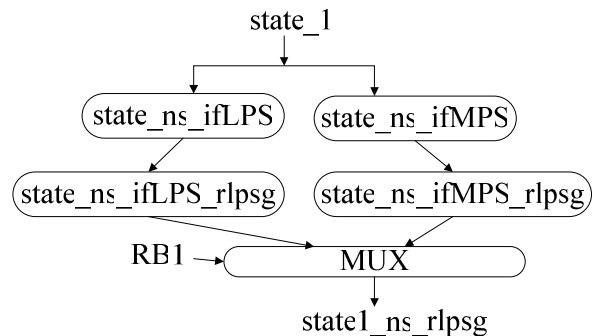


Fig. 7 state_1 renewal and preparation for rlpsg_2

B. Context maintenance

We have said in section III-C that context models are divided into 18 groups and they are loaded and written back to RAM group by group. The principle of organizing context models is

to organize those used repeatedly and consecutively together so that all context models of the same group are loaded and written back at most once during decoding one MB. We illustrate the principle using the following examples:

1) syntax element *mb_type* for I frame uses 7 different context models. This syntax element is decoded only once in one MB, the related models can only be called consecutively and will not be used while decoding the other syntax elements in the MB. So, we put these context models in one group.

2) syntax element *coeff_abs_level_minus1* in 4x4 blocks of *ctxBlockCat* of value 1 (*ctxBlockCat* is a classification label of blocks. *ctxBlockCat* of value 1 corresponds to luma AC transform coefficient levels of MB encoded in Intra_16x16 prediction mode) uses 10 different context models. But this syntax element may appear many times during the process of decoding one MB, and they are interleaved with other three syntax elements *coded_block_flag*, *significant_coeff_flag*, *last_significant_coeff_flag*. To avoid loading and writing back the context models too frequently during the decoding process, the context models for *coded_block_flag*, *significant_coeff_flag*, *last_significant_coeff_flag* and *coeff_abs_level_minus1* of one MB of a certain *ctxBlockCat* are gathered in one group. In this way, this group of at most 44 context models is loaded at the start of decoding the residual block of a certain *ctxBlockCat*, and written back after all residual blocks of the same *ctxBlockCat* are decoded out.

Thus, all 399 context models correspond to certain syntax elements are divided into 18 groups as following (we show groups of syntax elements, all context models corresponding to the bins of the syntax elements in the same group are grouped together):

1) *mb_skip_flag*; 2) *mb_field_decoding_flag*; 3) *mb_type* for SI slice; 4) *mb_type* for I slice; 5) *mb_type* for P slice; 6) *mb_type* for B slice; 7) *sub_mb_type* for P slice; 8) *sub_mb_type* for B slice; 9) syntax elements for intra prediction mode, including *prev_intra4x4_pred_mode_flag*, *rem_intra4x4_pred_mode*, *intra_chroma_pred_mode*; 10) *ref_idx*; 11) *abs_mvd*; 12) *coded_block_pattern*; 13) *mb_dq_delta*; 14) syntax elements for decoding residual data of a certain *ctxBlockCat*, including *coded_block_flag*, *significant_coeff_flag*, *last_significant_coeff_flag*, *coeff_abs_level_minus1*. There are five different values of *ctxBlockCat*, and each of them corresponds to one group.

Only the group of the last class described above contains at most 44 context models, other groups contains at most 14 context models. So forty-four 7bits register is enough. In our architecture, all 399 context models are stored in a 32×105 bits single-port RAM. Each row contains 15 context models with some vacant space. This ensures all groups of context models other than that of the last class are placed in the same row, and thus can be loaded and written back in one cycle, and the group of the last class can be loaded and written back within three cycles.

V. CONCLUSION

In this work we have presented a novel architecture for CABAC decoding. CABAC is a key technology adopted in

H.264/AVC standard as it offers a 16% bit-rate reduction when compared to baseline entropy coder while increasing access frequency from 25% to 30%.

We have shown that CABAC decoding is a highly time consuming process. Processing time estimation is presented that demonstrates CPU or DSP not being the appropriate choice for real-time CABAC decoding applications. So hardware implementation architecture is proposed.

Our architecture deals with the whole decoding procedure of MB by hardware. To the best of our knowledge, no other hardware implementations designed for CABAC coder/decoder is based on one MB. The best of the architecture proposed till now is the arithmetic coding architecture proposed by Osorio et al. [5], where only *significant_coeff_flag*, *last_significant_coeff_flag* and *coeff_abs_level_minus1* are accelerated by hardware, with much more complicated control and interaction cost between software and hardware.

In our architecture, we propose the new and efficient basic decoding circuit units which can decode 1~3 bins per cycle. The basic units are shared by all syntax elements and hence greatly enhance the decoding speed as it takes advantage of the characters of syntax elements. To the best of our knowledge, other arithmetic encoding/decoding architectures averagely encode/decode at most one bin per cycle and they do not take the characteristics of syntax elements into consideration [5], [9], [13].

We also propose an efficient context managing mechanism which divides context models into several groups according to the order of being called. This guarantees that loading and writing context models cost only a few cycles during decoding one MB. Only 44 registers of seven bits are needed to keep updating the context models. Some caching and pre-fetching mechanism is proposed to manage context models in earlier papers [5], [7]. However, their control is much more complicated and the whole decoding speed is not as fast as ours.

We have implemented this architecture in verilog using 0.18 μm technology. Synthetic results show that the total area is about 0.3mm² with critical path limited within 6.7ns. This result does not count in the 32×105 bits RAM memory for storing the 399 context models. We conducted tests on a typical 4M/s bit stream of D1 resolution, and results show that averagely one MB can be decoded within 500 cycles. This is quite rich for D1 resolution real-time application. Much space is left for real-time applications of higher video quality and larger image resolutions in the future.

ACKNOWLEDGMENT

The authors wish to thank Lin Bo, Yang Kai, Wang Lin, Lin Yi and all members of the work team of Fundamental Dept. R & D, Huawei Technologies Co., Ltd for their support and discussions. The authors also wish to thank Li Yu for his valuable comments and suggestions.

REFERENCES

- [1] JVT-G050, "Draft ITU-T Recommendation H.264 and Draft ISO/IEC 14496-10 AVC" in JVT of ISO/IEC doc, T.Wig,Ed., Pattaya, Thailand, Mar.2003
- [2] M.Mrak, D.Marpe & T.Wiegand, "A context modeling algorithm and its application in video compression", presented at ICIP, Barcelona, Spain, Sept.2003
- [3] S.Saponara, C. Blanch, K.denolf & J.Bormans, "The JVT advanced video coding standard: Complexity and performance analysis on a tool by tool basis", in IEEE Packet Video 2003
- [4] Jorn Osterman et. "Video coding with H.264/AVC: Tools, Performance and Complexity", IEEE Circuits and Systems Magazine 2004.
- [5] Osorio, R.R.; Bruguera, J.D. "Arithmetic coding architecture for H.264/AVC CABAC compression system", Digital System Design, 2004. DSD 2004. Euromicro Symposium on, 31 Aug.-3 Sept. 2004 Page62 – 69.
- [6] W.B.Pennebaker, J.L.Mitchell, G.G.Langdon & R.B.Arps, "An overview of the Basic principles of the Q-Coder Adaptive Binary Arithmetic Coder", IBM J.Res.Develop.32 (Nov.1988), 717-726.
- [7] W.B.Pennebaker & J.L.Mitchell, "Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder", IBM J.Res.Develop.32 (Nov.1988), 727.
- [8] Osorio, R.R.; Boo, M.; Bruguera, J.D. "Arithmetic image coding decoding architecture based on a cache memory by use", Euromicro Conference, 1998. Proceedings. 24th, Volume: 1, 25-27 Aug. 1998 Pages:139 - 146 vol.1
- [9] K. Andra, C. Chakrabarti, and T. Acharya. "A high performance JPEG2000 architecture", IEEE Transactions on CSVT, 13(3):209-217, March 2003
- [10] 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 Pages:620 – 636
- [11] M.Mrak, D.Marpe & T.Wiegand, "A context modeling algorithm and its application in video compression", presented at ICIP, Barcelona, Spain, Sept.2003
- [12] I.H.Witten, R.M.Neal & J.G.Cleary, "Arithmetic Coding for Data Compression", Comm.ACM30 (June 1987),520-540
- [13] J.L.Nunez, V.A.Chouliaras, "High-performance arithmetic coding VLSI Macro for the H264 Video Compression Standard", IEEE Transactions on Consumer Electronics, Vol. 51, February 2005



Wei Yu was born in Wuhan, China, in 1983. She received the B.S. degree from Tsinghua University, Beijing, China, in 2003. She is currently pursuing the M.S. degree at the State Key Lab on Microwave & Digital Communications of Dept. of Electronic Engineering, Tsinghua University. Her current interests are in the areas of moving picture coding and codec VLSI architecture design.



Yun He (M'97-SM'01) received the B.S. in signal processing from Harbin Engineering University, China, 1982, the M.S. degree in ultrasonic signal processing from Shanghai Jiaotong University, China, 1984, and the Ph.D degree in image processing from Liege University, Belgium, 1989.

She is now a Professor in the Department of Electronic Eng., Tsinghua University. Her research interests include Video Coding/Compression, Video Communication, Video Codec VLSI structure, Error Resilience/Concealment Video Communications, and Multi-Stream/Multi-View Video Communications.

She serves as a Technical Committee member of Visual Signal Processing and Communications of the IEEE Circuits and Systems Society, Picture Coding Symposium, PCS Steering Committee member, and Program Committee member in SPIE-Conference of Visual Communications and Image Processing, VCIP 2000-2001. She was invited as session Chairs in several important conferences of picture coding, such as in PCS, VCIP, PCM. She will be the General Chair of the 25th PCS.