國 立 交 通 大 學

電子工程學系

碩 士 論 文

H.264/AVC 算數編碼器和算數解碼器之硬體架構設計

Arithmetic Coder and Decoder Architecture Designs for H.264/AVC

指導教授：蔣迪豪　博士

研 究 生：林承毅

中 華 民 國 九 十 四 年　七 月

研 究 生：林承毅	Student: Cheng-Yi Lin
指導教授：蔣迪豪	Advisor: Tihao Chiang

國 立 交 通 大 學

電子工程學系電子研究所碩士班

碩 士 論 文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Electronics Engineering

July 2005

HsinChu, Taiwan, Republic of China

中華民國九十四年七月

H.264/AVC 算數編碼器和算數解碼器之硬體架構設計

研究生: 林承毅　　　　　　　　　　　　指導教授: 蔣迪豪 博士

國立交通大學

電子工程學系 電子研究所碩士班

摘　　要

H.264/AVC是最新壓縮標準。與其他標準比較，H264/AVC提供了較高的壓縮效率，但是H.264的複雜度也相對較高。在H.264/AVC有兩種熵編碼法，CAVLC和CABAC。在二熵編碼法之中，CABAC 能比CABLC節省10-15%位元率。基本上，熵編碼是一種二位元的操作，且一般多功能處理器不能有效率的處裡。在高解析度及時的系統中,給熵編碼用的一個高處裡能力的算術編碼器和解碼器是非常需要的。

在這篇論文裡，我們提出給H.264/AVC用的算術編碼器和算術解碼器之硬體架構。為了增加算術編碼器的處裡能力，架構設計上能擴充到把每個週期可以編碼多個位元。為降低架構上的長路徑，我們在算術編碼器和算數解碼器架構中裡重新安排迴圈中的處裡順序。而且，我們的算數編碼器設計能容易修改去支援JPEG2000。全部設計被用硬體描述語言實現並且在FPGA環境中作過驗證。算術編碼器的最大處裡能力是每秒545百萬個位元，算數解碼器的最大處裡能力是每秒330百萬個位元。他們分別花費9300和3500個邏輯單元。

Arithmetic Coder and Decoder Architecture Designs for H.264/AVC

Student: ChengYi Lin Advisor: Dr. Tihao Chiang

Institute of Electronics
National Chiao Tung University

Abstract

The most recently developed compression standard H.264/AVC provides outstanding performance than other standards, but the complexity of H.264/AVC is also higher than others. There are two entropy coding methods, Context Adaptive Variable Length Code (CAVLC) and Context Adaptive Binary Arithmetic Code (CABAC) in H.264/AVC. Between the two entropy coding methods, CABAC can provide bit-rate saving of 10-15% than CAVLC. The serial and bit-level operation of the entropy coding is a kind of bit-level operation and can not be effectively handled by general purpose processor. A high throughput arithmetic coder for entropy coding and decoding is strongly required for high resolution real-time applications.

In this thesis, we propose architecture for the arithmetic encoder and decoder in H.264/AVC. To meet the high performance requirement, the encoder design can be extended to encode multiple symbols per cycle. To reduce the critical path, we rearrange the sequence of range operation in the encoder and the decoder architectures. All designs are implemented in Verilog and verified on FPGS. The maximum throughput of the 3-sybmol arithmetic coder is 545M symbols per second and the maximum throughput of decoder is 330M symbols per second costing 9.3k and 3.5k gates respectively.

誌謝

研究生的生活過到現在已經兩年了，隨著這篇論文的出現，研究生的生崖就要結束了。在這兩年的生活，過得很充實，也學到了很多。在整個研究生的兩年中，最主要學到的實作研究和解決問題的方法。兩年來老師和學長不斷的教導還有同學跟學長學弟們間的互相學習，讓我成長了不少。要感謝的人，真的不少。

首先要感謝的是實驗室的長輩們，老師跟學長。指導教授蔣迪豪老在研究之路上給了一個方向和提供了不少想法。而彭文孝學長，平常在 我研究內容的細節跟方向上，提供了我很多意見，並且在這篇論文的寫作中，給了不少幫助。在研究的工作環境中，非常感謝王士豪學長在工作站的維護，提供良好的工作環境讓我們做研究，使研究可以順利進行，沒有後顧之憂。

還要感謝的是實驗室的同學，思浩跟世騫在有功課或研究上有問題的時候互相討論，並在 group meeting 的時候互相討論跟聊天，提供了不少歡樂的氣氛。在平常的實驗室生活中，還要感謝漢光，阿志，機車竹…等，平常打屁聊天，一起買飯吃，一起討論鄭教授的助理，這些都使在實驗室生活過的很歡樂。

最後，要感謝的是我的家人，求學生崖中對我默默的支持，讓我走過沒有後顧之後的求學之路。

Index

LIST OF FIGURES

List of Tables

Chapter 1

Introduction

1.1 Overview of Dissertation

In recent years, the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) jointly develop a video coding standard H.264/AVC for a wide range of applications such as storage, video conferencing, broadcasting, Internet streaming etc. As compared to other video coding standards, H.264/AVC can provide the same quality with a significantly reduced bit rate. Specifically, as compared to MPEG-4, H.263, MPEG-2, the ratio of saving on bit rate is 39%, 49%, and 64% respectively.

Although the coding efficiency of H.264/AVC is better, the complexity of

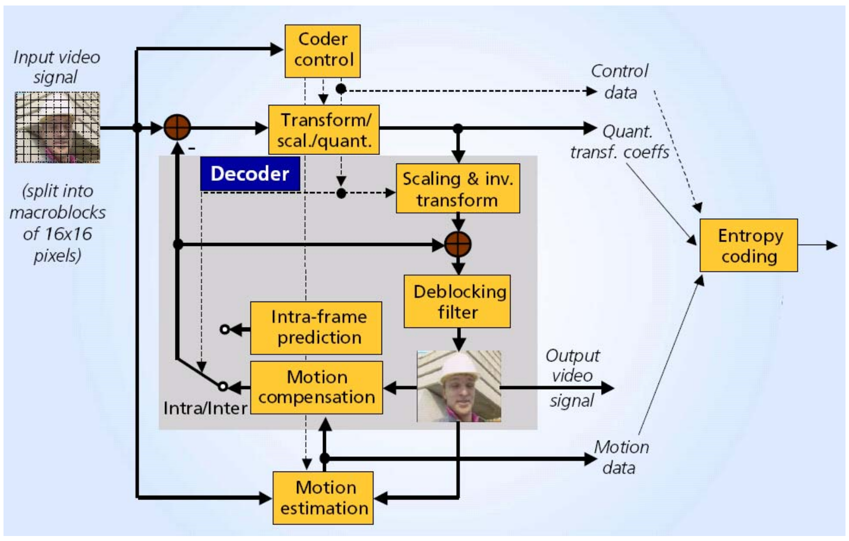

Figure 1.1 Basic Coding Structure of H.264/AVC for a macroblock [?]

H.264/AVC is relatively high. Figure 1.1 shows the basic coding structure. Among the modules, there are many new tools such as long-term prediction, motion estimation of variable blocks size are designed for increasing coding efficiency. However, the tools need more computation and the complexity is increased significantly. For the decoding, H.264/AVC is 2-3 times more complex than MPEG-2, H.263, and MPEG-4. For the encoding, the complexity of H.264/AVC is 4-5 times than that of MPEG-4.

Among these modules, CABAC is one tool that needs intensive computation. Basically, the operations of CABAC include the following steps:

1. Binarization: The binarization of syntax element is to maximize the efficiency of binary arithmetic coding. In H.264/AVC, all the syntax elements are binarized into multiple bits. There are four basic binarization methods in CABAC of H.264/AVC. For instance, fixed-length code is used for syntax element with a nearly uniform distribution. The code word of x in fixed-length code is simply given by the binarization representation of x with a fixed (minimum) number l_{FL}=ceil(*log2S)* of bits, where 0≦x≦S.
2. Context Modeling: The context model is to fully utilize the existing correlations. After the binarization, the coding bits of a syntax element may refer to different context models. The outcome of a context model is assigned with a context index and each index is associated with a binary probability model. Specifically, the context probability model is represented by a most probable symbol (MPS) and the probability of least probable symbol (LPS).
3. Binary arithmetic coding: The binary arithmetic coding is to reduce the bit rate. During the coding, the interval state of the arithmetic coding engine is as usual characterized by two quantities: the current interval range (range) and the base of the current code interval (low). We make the encoding

decision by comparing the encoding symbol and the MPS in the context probability model. Depending on the encoding decision, the range and low change their values.

Different from the other modules, the CABAC operates at bit level. It cannot be efficiently handled by general purpose processors. Moreover, the encoding/decoding in CABAC refers to many coding context. A strong data dependency exists in the successive operations of CABAC. For instance, the encoding/decoding of a specific syntax may refer to the state of previously encoded/decoded syntax. Such data dependency makes the operations of CABAC not easy for parallel processing. Furthermore, CABAC requires many branching instructions that also pose a big challenge for hardware design.

In recent years, a few hardware architectures [4][5][6][7][8] about CABAC codec have been proposed for different coding standards. In common, these architectures can encode one symbol per cycle. However, some of them [7] can have more than 1 symbols coded in a cycle. In [4], an architecture with 4 pipeline stages is proposed for JPEG-2000. They divide the computations of range and low into two stages. Also, the adder for updating the low value is pipelined so as to reduce the path delay. Their architecture can encode one symbol per cycle. Based on [4], in [7] the architecture is further extended so that it can encode multiple symbols per cycle. To achieve this, the functional units for the computations of range and low are duplicated. However, simple duplication of these functional units will lead to a longer critical path. Therefore, in [7] the sequence of some elementary operations is inverted and six branches to map all the allowable scenarios are used. Such a method is named as inverse multiple branch selection (IMBS). In [5] and [6], architecture with 3 pipeline stages is proposed for supporting the CABAC operations in multiple standards. They integrate the context information of JPEG, JPEG-2000, JBIG and JBIG2 into the FSM and employ a parallel leading zeros detection and bit-stuffing handling for encoding one symbol per cycle. In [8], they propose an architecture with 2 pipeline stages for H.264/AVC. In their architecture, the first stage handles the computations of range and low and the second stage uses a technique known as byte-stuffing to handle output bits.

To improve the performance of H.264/AVC system, in this thesis, we propose a scalable architecture for the CABAC encoding. In addition, based on the architecture for CABAC encoding, we further develop an architecture for CABAC decoding. Different from the conventional designs [4], [5], [6], and [8], our scalable encoding architecture can simultaneously encode multiple symbols per cycle and support both JPEG-2000 and H.264/AVC.

Particularly, in our scheme, we rearrange the operations for CABAC encoding to remove the data dependency in CABAC operations. For instance, the computations of range in the encoding iteration include the following three steps:

(1) Table look-up for the range of LPS (rLPS).

(2) The range of MPS computation and new range selection.

(3) Renormalization of new range.

In the step 1, we partition the table for the range of LPS (rLPS) into four parts. Therefore the table look-up is independent of range. Therefore, the step 1 can compute without waiting the step 3 of pervious iteration. The step 1 of this iteration and the step 3 of pervious iteration can be done in parallel. Then, we rearrange the operation of range iteration. In the iteration, we put step1 and step 3 in parallel. Then the step 2 is the final operation in the new iteration. With such rearrangement, the critical path can be reduced.

In addition, for the decoding of CABAC, we reuse the design employed by the encoding. Particularly, the data dependency for decoding is stronger than that for encoding. The operations of range in decoder iteration are composed of following steps:

(1) Table look up for rLPS.

(2) Computation of rMPS.

(3) Comparison of rMPS and offset to make decision.

(4) Renormalization of range.

(5) Renormalization of offset.

We remove the dependency of rang and the table rLPS as we do in encoder. Therefore Step1 and Step2 can be put in parallel with setp 4 and step 5. We do step 3 in the final iteration. Therefore, we can reduce the delay of the path.

To validate the proposed architecture design, we implement our CABAC codec using the cell-based synthesis approach. Specifically, we use 0.18um CMOS technology to synthesize our designs. Moreover, for the verification at system level, we wrap our designs in AHB interface. Our designs with AHB interface are slave modules on AHB. The CPU can use the slave modules to through AHB. Our all designs can correctly encode and decode a sequence with the codec of H.264/AVC

In our designs, the CABAC encoding can support the throughput of 545Mbits/s and the throughput of decoding is up to 330Mbits/s. In other words, our design can meet the requirements of real-time encoding/decoding in HDTV resolution. Respectively, the gate counts used are 9.3k and 3.5k.

1.2 Organization and Contribution

In this thesis, we propose a scalable encoding architecture for CABAC in H.264/AVC and JPEG2000. Also, the encoding architecture is modified to support the decoding of CABAC. For the verification, we use an ARM-based platform for the co-simulation of hardware and software at system level. The proposed solution can meet the real-time requirements for HDTV applications. As compared to the state-of-the-art designs, our architecture has higher throughput and lower cost. For more detail, the rest of this thesis is organized as follows:

- Chapter 2 details the algorithm of CABAC and the prior works for the hardware designs.
- Chapter 3 introduces the proposed architectures for the CABAC encoding and decoding. Specifically, our contributions include the following:
  - Encoder
    - We rearrange the operations of CABAC for parallel processing so as to improve the performance.
    - We propose a 3 pipeline stages architecture which can encode more than 1 symbol per cycle.
    - Furthermore, our architecture can be easily modified to support both JPEG2000 and H.264/AVC.
  - Decoder
    - There is stronger data dependency in decoder than in encoder. For reducing the critical path, we rearrange the range operation sequence as we do in encoder architecture.
- Chapter 4 shows the implementation results and comparison with other designs. In brief, the performance of our design is summarized as below:
  - As compared to [8], our maximum throughput is 2.7 times faster.
  - As compared to [4], [5], [6], and [7], we have less gate counts, i.e., lower cost.
- Chapter 5 summaries our works and shows the conclusions of this thesis.

Chapter 2

Algorithm of CABAC

In this chapter, we introduce the algorithm of CABAC. Figure 2.1 shows the generic block diagram for encoding a signal syntax element in CABAC. In the CABAC framework, the encoding processing include threes steps which are (1) binarization, (2) context modeling and (3) binary arithmetic coding. If the input is binary syntax element, the first step is skipped. There are two encoding processes, regular and bypass encoding, in CABAC. In the regular coding process, it need to a context model for encoding. Each type of bin has its own context model. When getting the context model, the regular coding engine encodes the bin value with the context information. The design purpose of bypass coding is a speedup of the whole encoding/decoding process. The mean of speedup is to simplify coding engine without the usage of an explicitly assigned model.

In this chapter, we describe the three steps of CABAC frame work respectively. The detail algorithm of the steps will be showed in the followed section. In the previous

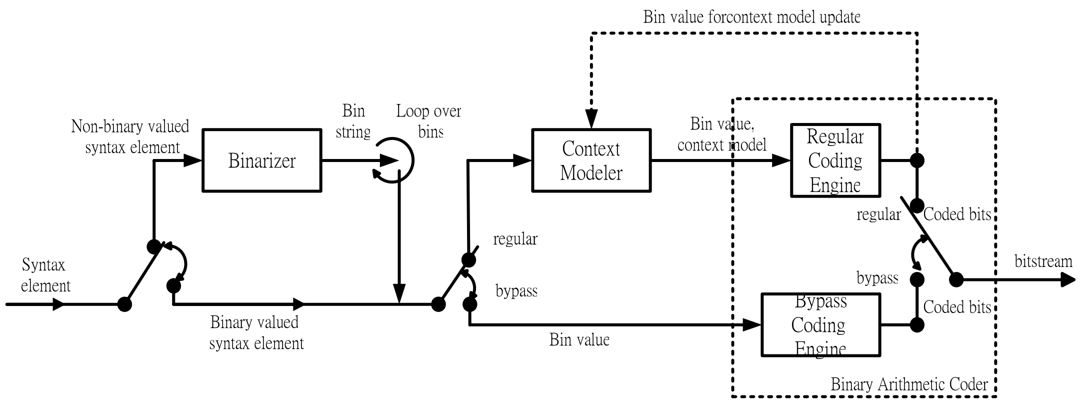

Figure 2.1 Block diagram of CABAC [2].

work, there are many kinds of arithmetic coder and decoder designs for the different standard. We show the previous work in the final section of this chapter.

2.1 Binarization

The design goal of binarization is for getting minimum redundancy code. The minimum redundancy code can reduce the workload of binary arithmetic coding stage. There are four basic types: the unary code, the truncated unary code, the kth order Exp-Golomb code, and the fixed-length code. In addition there are binarization schemes based on a concatenation of these elementary types.

Here, we show the algorithm of four basic binarization types.

1. Unary code:

   For each unsigned integer valued symbol x≧0, the unary code work in CABAC consists if x ”1” bits plus a terminating “0” bits.

   Example: 5 => 111110

2. Truncated unary code:

   The truncated unary code is only designed for 0≦x≦S. If x < S, the code word is given by unary code. Else if x equal S, the code word is consisted of x “1” bits.

   Example: Truncated unary code with S = 5:

   3 => 1110, 5 => 11111

3. Kth order Exp-Golomb code:

   The code words of Kth order Exp-Golomb code are constructed by prefix and suffix parts. The prefix part is unary code corresponding to the value *l(x)= floor(log2(x/2k+1))* and the suffix part is the binary representation of *x+2k(1-2l(x))* using *k+l(x)* significant bits.

4. Fixed-length code:

   The code word of x in fixed-length code is simply given by the binarization representation of x with a fixed (minimum) number l_{FL}=ceil(*log2S)* of bits, where 0 ≦x≦S. This kind of type is used for syntax element with a nearly uniform distribution.

Example: S = 7

1=> 001, 2=> 010… 7=> 111.

There are three more binarization schemes derived in binarization of CABAC. The first one is a concatenation of a 4-bit fixed-length prefix and a truncated unary suffix with S=2. Both the second and third concatenated schemes are derived from the truncated unary and the kth Exp-Golomb binarization. These schemes, which are referred to as Unary/kth order Exp-Golomb (UEGk) binarizations, are applied to motion vector difference and absolute value of transform coefficient levels.

2.2 Context Model

For increasing coding efficiency, each type of bin has its own context model to estimate the probability model. One context model includes the probability of least probable symbol (LPS) and the most probable symbol (MPS). In this section, we show the basic types of context models. Then we show the probability estimation of context model.

For encoding each symbol, a conditional probability is estimated by switching between different models of probability according to the already coded neighboring symbols. There are four basic types of context models in CABAC of H.264/AVC. The first type of context models involves a context template with up to two neighboring syntax elements in the past. Usually, the relating element is to the left and on the top of current element. As figure 2.2 shows, when encoding syntax C, it based on the two syntax element, A and B, to choose the suitable context model. The second type of context models is only designed for the syntax elements of mb_type and syb_mb_sype.

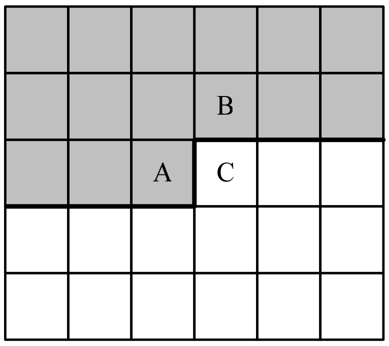

Figure 2.2 Illustration of an example in the first type context model

The values of prior coded bins are used for the choice of a model. Thus when we encode ith bin (b_i) of a syntax element, the values of b_1, b_2... b_{i-1} are used for choosing the context model of b_i. Both the third and fourth type of context models is applied to residual data only. The third type relies on the scanning path of the syntax element. For the fourth type, modeling function is involving the evaluation of the accumulated number of the encoded levels with a specific value prior to the current level bin to encode. In another word, we use the level information to choose the context model in the fourth type of context models.

In the CABAC of H.264/AVC, each syntax element has its own context model. Each context model is given an index number. The index number is called context index. The total number of context models in H.264/AVC is 399. The range of context index is from 0 to 398. Thus the context model can be efficiently represented by 7-bit unsigned integer values. Table2.1 [2] shows the association of syntax elements and the range of context indices in H.264/AVC. The context indices in the range from 0 to 72 are related to syntax element of macroblock type, submacroblock type, and prediction modes of spatial and temporal type. Context indices in the range from 73 to 398 are related to the coding of residual data.

Table 2.1 syntax element and associated range of context indices [3]

Syntax element	Slice type		
	SI/I	P/SP	B
mb_type	0/3-10	14-20	27-35
mb_skip_flag		11-13	24-26
sub_mb_type		21-23	36-39
mvd(horizontal)		40-46	40-46
mvd(vertical)		47-53	47-53
ref_idx		54-59	54-49
mb_qp_delta	60-63	60-63	60-63
intra_chroma_pred_mode	64-67	64-67	64-67
prev_intra4x4_pred_mode_flag	68	68	68
rem_intra4x4_pred_mode	69	69	69
mb_field_decoding_flag	70-72	70-72	70-72
coded_block_pattern	73-84	73-84	73-84
coded_block_flag	84-104	84-104	84-104
significant_coeff_flag	105-165, 277-337	105-165, 277-337	105-165, 277-337
last_significant_coeff_flag	166-226, 338-398	166-226, 338-398	166-226, 338-398
coeff_abs_level_minus1	227-275	227-275	227-275
end_of_silce_flag	276	276	276

In CABAC, 64 values represent the probability of the LPS. The range of the 64 values is [0.01875, 0.5]. It is derived form the following recursive equation:

$$P_\sigma = \alpha * P_{\sigma-1} \text{ for all } \sigma=1,\ldots,63$$

$$With\ \ \alpha=(0.01875 / 0.5)\ (1/63) \fallingdotseq 0.95 \text{ and } p0=0.5$$

Hence, a context model can be completely represented in 7-bit. The 7 bits includes 6 bits of LPS probability index and 1 bit of MPS value.

After encoding a symbol, the used probability context model is updated. As figure 2.3 illustrates, when the encoding symbol equal MPS, the state index is simply incremented by 1. When a MPS occurs at state index 62, the state index does not

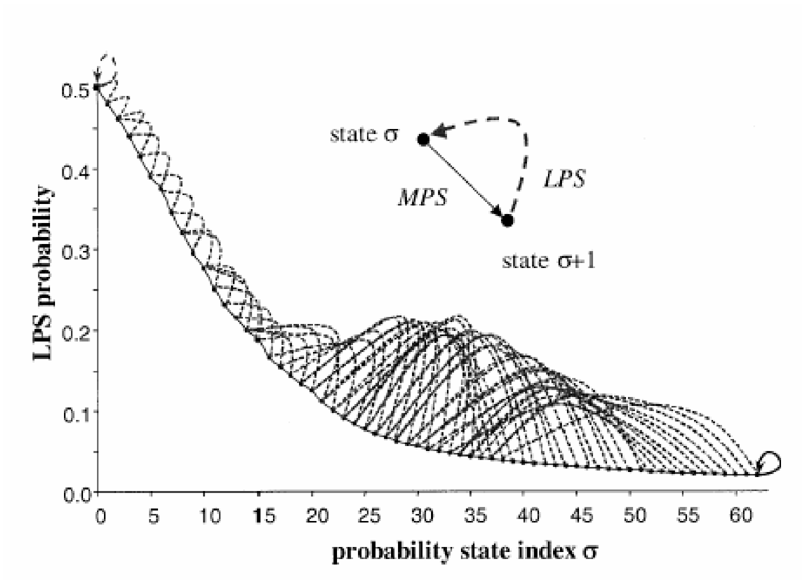

Figure 2.3 Transition rules for updating the probability of LPS [2]

change. The probability of LPS is already at its minimum. If encoding symbol equal LPS at state index 0, the value of MPS will change. It means that the original value of LPS becomes the new value of MPS.

2.3 Binary Arithmetic Coding

In the CABAC of H.264/AVC, the binary arithmetic coding includes two coding engine. One is for regular coding mode. The regular mode includes the utilization of adaptive probability mode. Another coding engine is for a fast coding of symbols. The second coding engine is so-called “bypass” coding engine. In the bypass coding process, an approximately uniform probability is assumed to be given. In this section, we show the more detail information about the regular coding engine and the bypass coding engine. First we show the encoding engine. Then we describe the algorithm of decoding engine.

2.3.1 Regular Encoding Engine

The algorithm of arithmetic coding in H.264/AVC is composed of two parts. One is

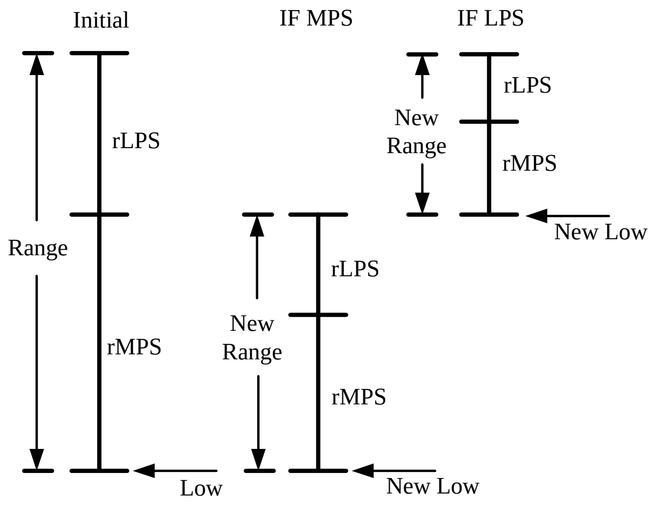

Figure 2.4 Encoding process

encoding process and the other part is renormalization and output generator. The encoding process decides the new range. The output generator generates bits for output. The total number of output bits is variable. It might be from 0 to 7 bits.

The basic operations of encoding process are recursive interval division and selection. Figure 2.4 shows the encoding process. The encoding equations are:

MPS(Most probable symbol) happen

$$\text{Lownew} = \text{Low}$$
$$\text{Rangenew} = \text{Range - LPS} = \text{rMPS}$$

LPS(Least Probable Symbol) happen

$$\text{Lownew} = \text{Low} + \text{rMPS}$$
$$\text{Rangenew} = \text{rLPS}$$

The Low and the Range of the equations indicate the bottom of the interval and the length of current interval. All operations in the encoding process are multiplication-free. The Range of LPS gets from table instead of multiplication. The value of rMPS get from the subtraction of Range and rLPS. Figure 2.5 illustrates flowchart of encoding process. When encoding one symbol, the value of range is smaller and the value of low is equal or bigger. For using the integer arithmetic coding, the value of is renormalized

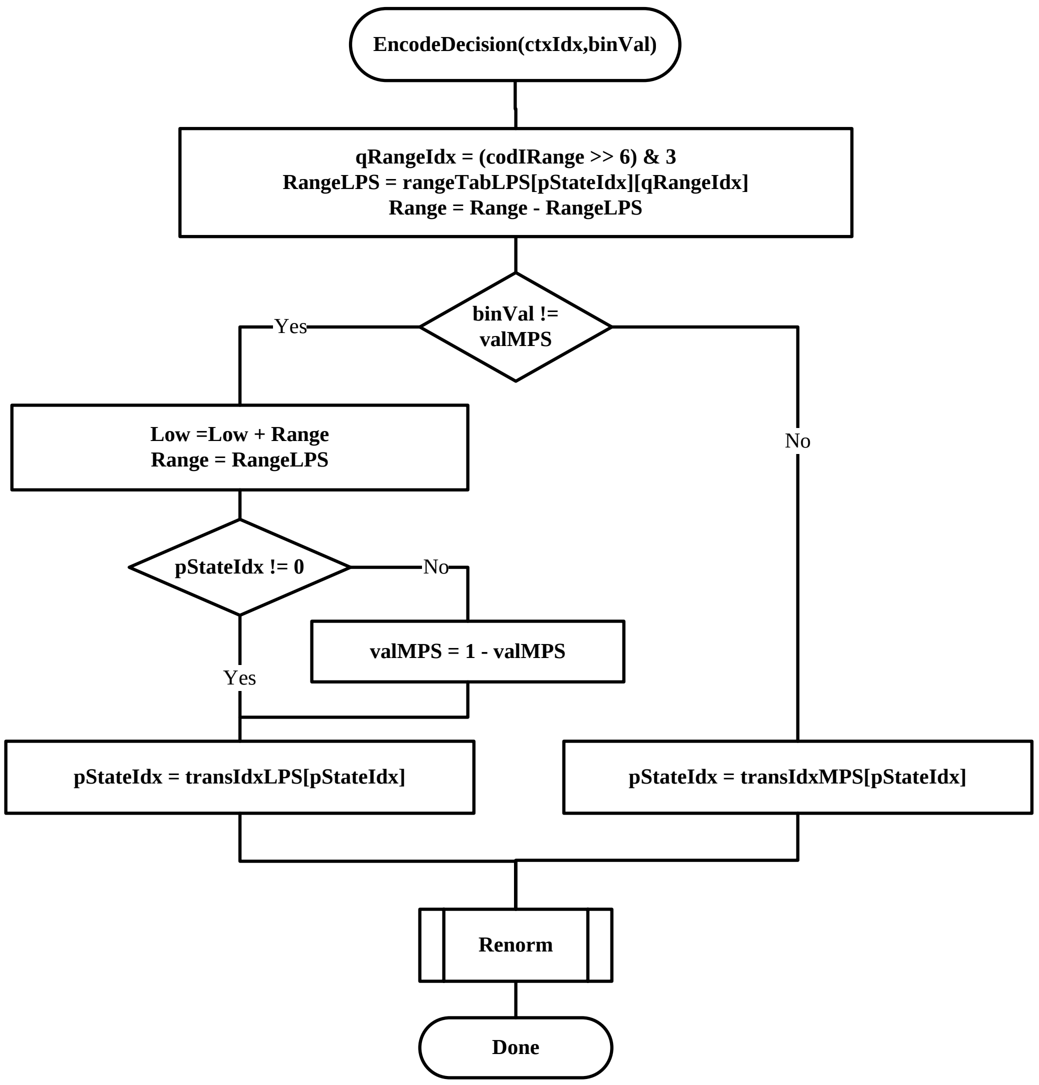
EncodeDecision(ctxIdx,binVal)
qRangeIdx = (codIRange >> 6) & 3
RangeLPS = rangeTabLPS[pStateIdx][qRangeIdx]
Range = Range - RangeLPS
binVal != valMPS
Yes
No
Low =Low + Range
Range = RangeLPS
pStateIdx != 0
No
valMPS = 1 - valMPS
Yes
pStateIdx = transIdxLPS[pStateIdx]
pStateIdx = transIdxMPS[pStateIdx]
Renorm
Done

Figure 2.5 Flowchart of encoding process

after each encoding one symbol. The value of low is renormalized with the value of range.

The context information is updated after encoding a symbol. There are two tables for probability transition of LPS. The transition rules are defined in previous Figure 2.3. When the value of MPS equals the value of symbol, we use the table of the transition index MPS. Otherwise, we use the table of the transition index LPS. If the state index equals 0 and the value of symbol equal the value of LPS, the value of MPS will change. In the state index “0”, the probability of LPS is 0.5. If we get that the encoding symbol equal LPS in state index “0”, the probability of original LPS will exceed 0.5. For

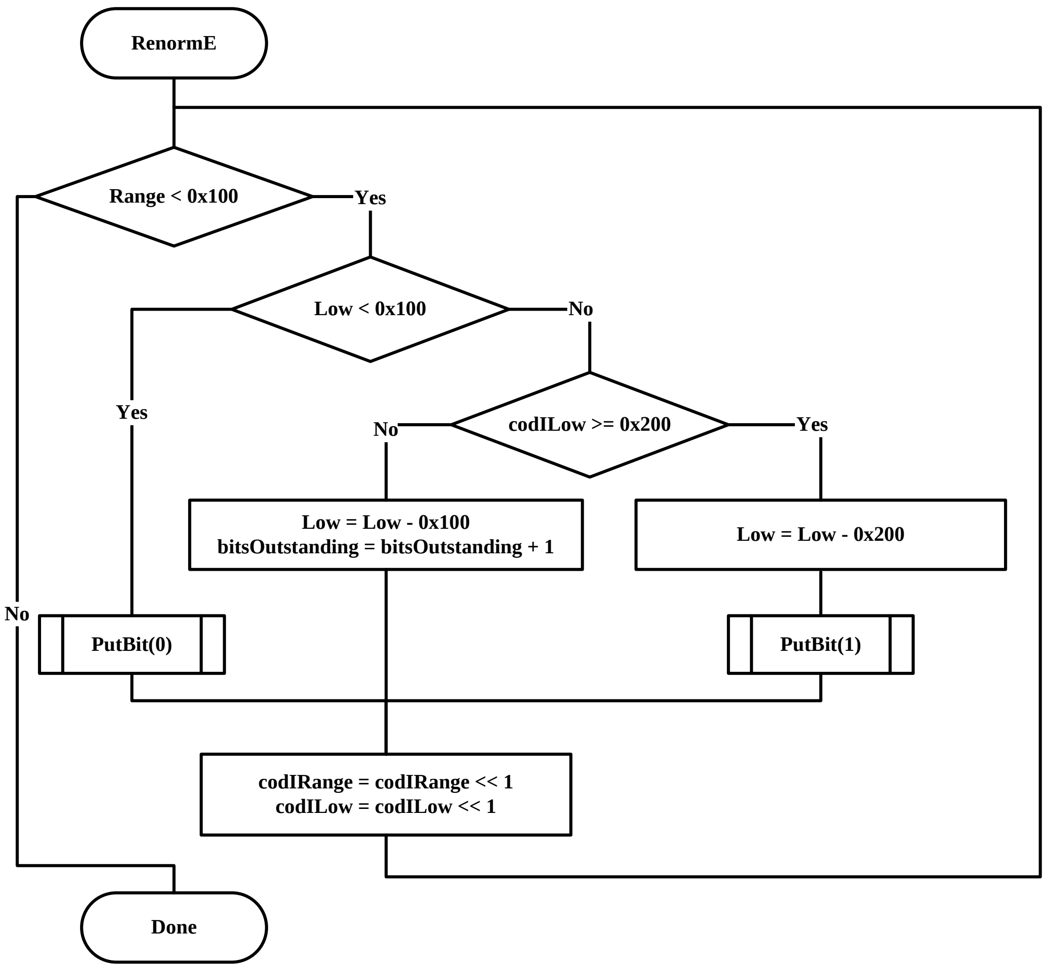

Figure 2.6 shows the renormalization and output process.

limiting the value of probability from 0 to 0.5, we change the value of LPS. It means the values of MPS and LPS exchange.

For maintaining the precision, the range and the low are renormalized after every encoding process. In CABAC of H.264, it needs 9 and 10 bits to present the values of ranges and the low respectively. The value of range gets smaller after every encoding process. Therefore the value of range is limited from 256 to 511. If range is smaller than 256 (0x100), the range needs renormalization. The renormalization of low follows the renormalization of range. Figure 2.6 shows the flowchart of the renormalization. In Figure 2.6, each recursive process handles one shift. The process continues until range is bigger than 0x100.

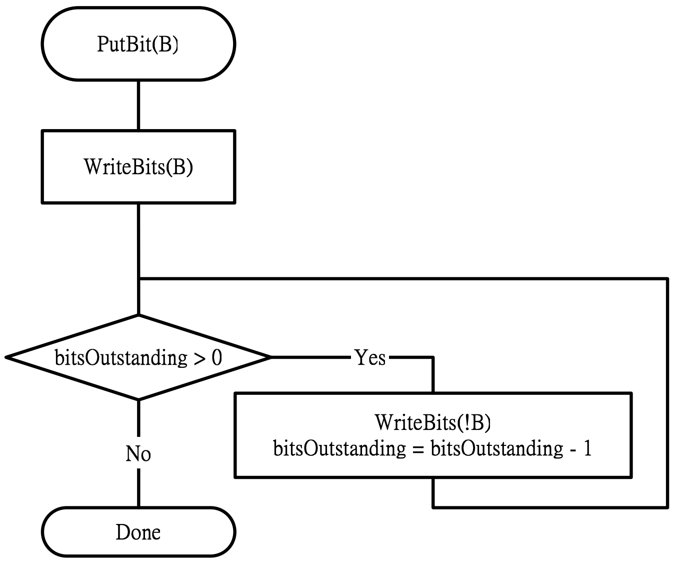

Figure 2.7 Flowchart of PutBit(B)

The output is completed in the renormalization. As Figure 2.6 shows, there are two kinds of situation in the output process. The first situation is that the first two bits of low are 00 or 10 or 11. We can determine the bit for output immediately. Then the first bit of low is useless. Before low shifts, we desert the first bit of low. The second situation is that the first two bits of low are 01. We can't determine the bit for output immediately. We use a register, called bitOutstanding, to count the continuous times of this condition. The flowchart of PutBit(B) function is shown in Figure 2.7. As we determine the bit for output, we put a sequence of bits after the bit. The values of the each bit in the sequence are the inverse value of the determined output bit and the length of the sequence is the value of bitOutstanding.

2.3.2 Bypass Encoding Engine

Bypass encoding is a simplified edition of regular encoding. In the bypass encoding, the probability of LPS is assumed 0.5. Hence there are no state indices used in the process. For determining the decision of encoding, its uses double of low instead of half of range. Therefore it needs 11 bits for presentation of low in bypass encoding. Figure

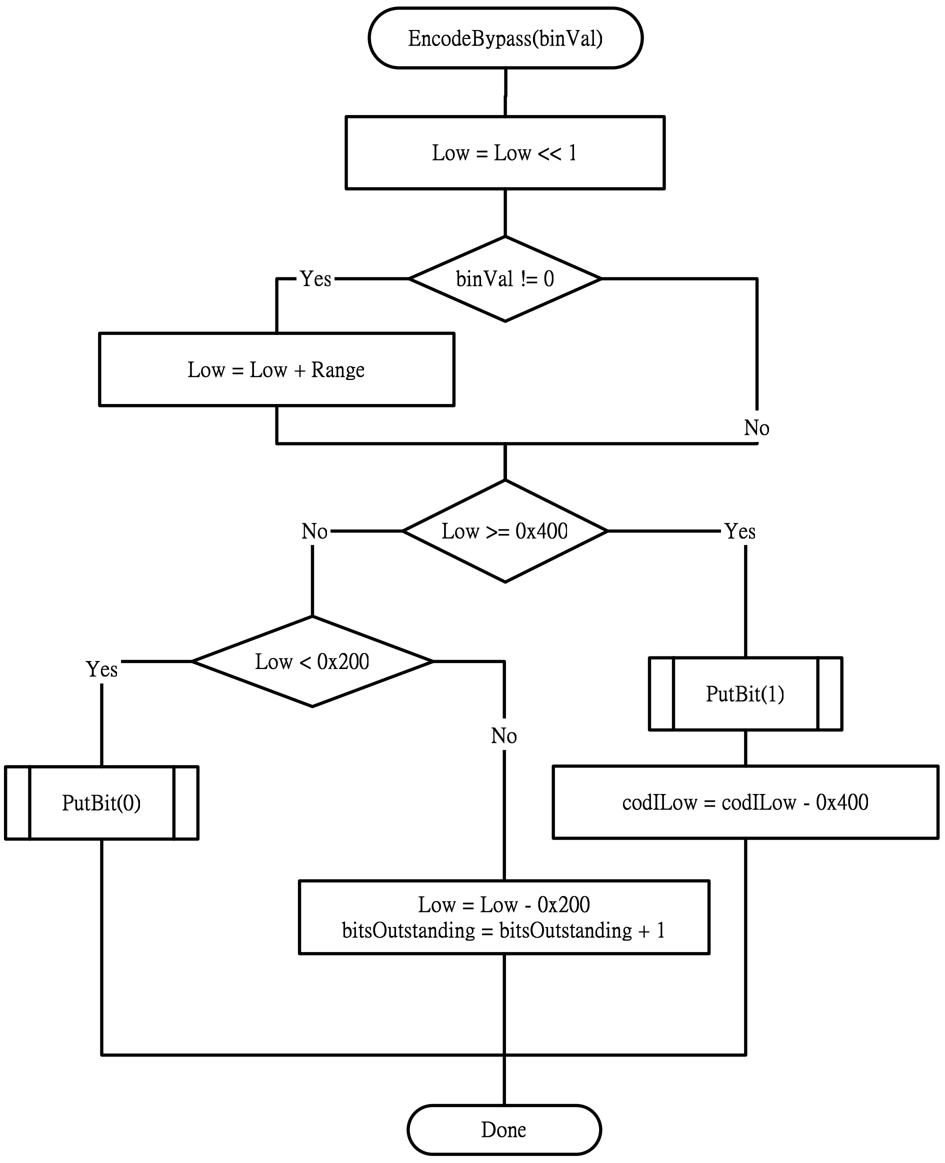

Figure 2.8 Flowchart of bypass encoding

2.8 shows the bypass encoding process. The under part in Figure 2.8 is the renormalization. There are no iterations in bypass encoding. As the renormalization of the regular encoding, the renormalization in the bypass encoding is almost the same. The output decision also depends on the first two bits of low.

2.3.3 Decoding Engine

The decoding engine has two kinds of coding engine as the encoding engine. The regular decoding engine handles normal situation. The bypass decoding engine decodes

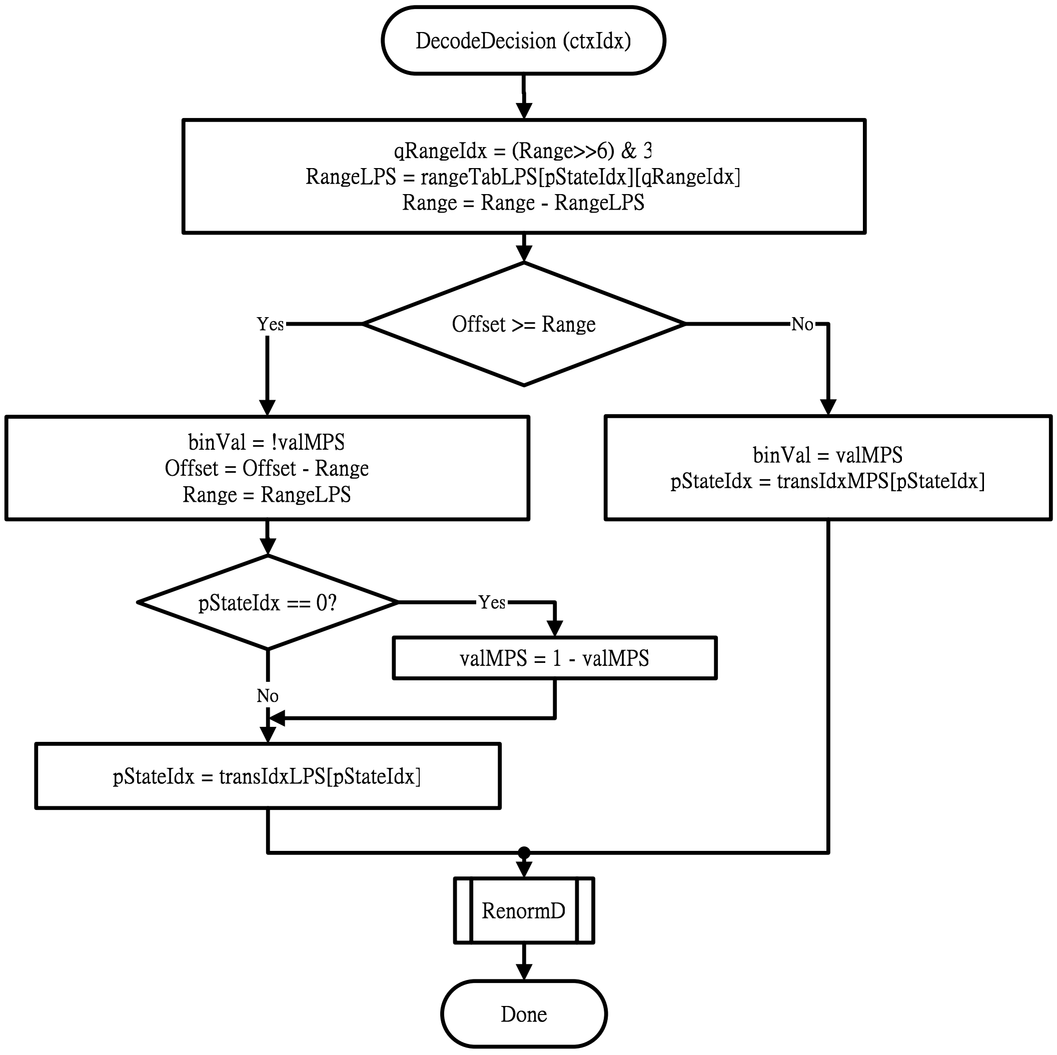

Figure 2.9 Flowchart of decoding engine

the symbol that we assume the probability of LPS is 0.5. The bypass decoding engine can accelerate the total decoding rate.

The flowchart of the decoding engine is shown in Figure 2.10. In the decoding process, the range of LPS is also from the table. The table indices are composed of the 7th and the 8th bits of range and the probability state of LPS. The decision of decoding is according to the relationship of the offset and the rMPS. The offset means the probability of the symbol sequences. In Figure 2.10(A), if the value of offset is bigger than range of MPS, the decoding symbol is MPS and the new values of range is rMPS. Otherwise the decoding symbol is LPS and the new range is rLPS. In Figure 2.10(B), if the value of the offset is smaller than the range of MPS, we subtract the original offset

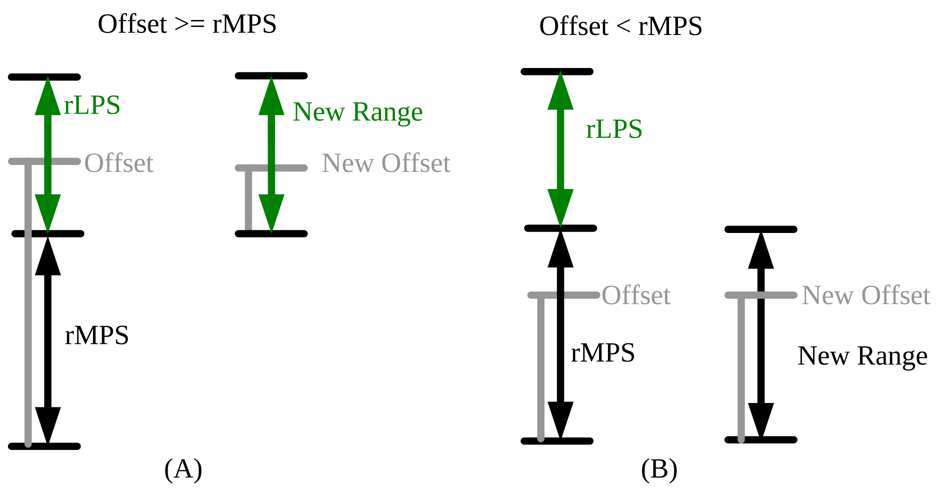

Figure 2.10 Decoding decision

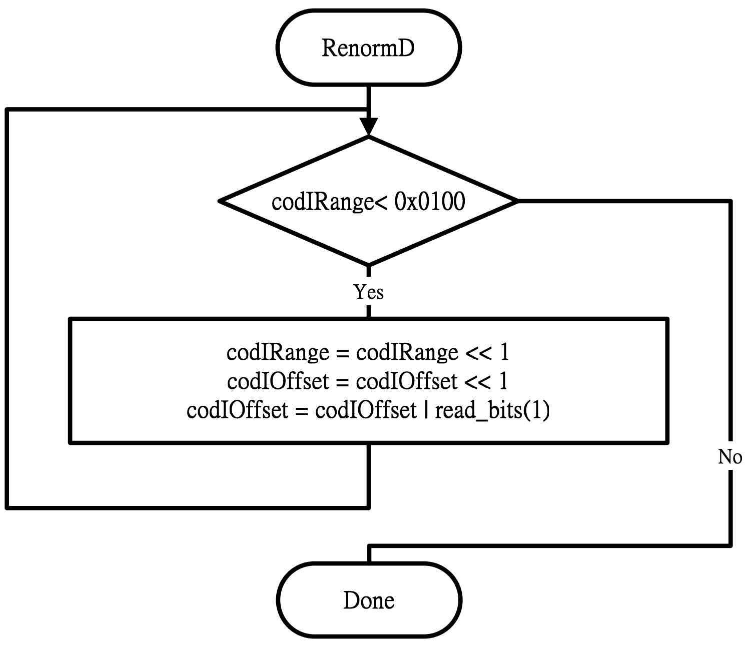
RenormD
codIRange< 0x0100
Yes
codIRange = codIRange << 1
codIOffset = codIOffset << 1
codIOffset = codIOffset | read_bits(1)
No
Done

Figure 2.11 Renormalization of decoding engine

and rMPS to get the new offset. The context probability model is updated in every decoding. The transition rule is the same as the rule in encoding engine.

The final operation is renormalization. Figure 2.11 shows the flowchart of renormalization. The range and the offset are getting smaller after encoding. For integer implementation, when the value of range is smaller than 256, we shift the value of range. The range is limited from 256 to 511. As the range shift one bit to left, the low also shift

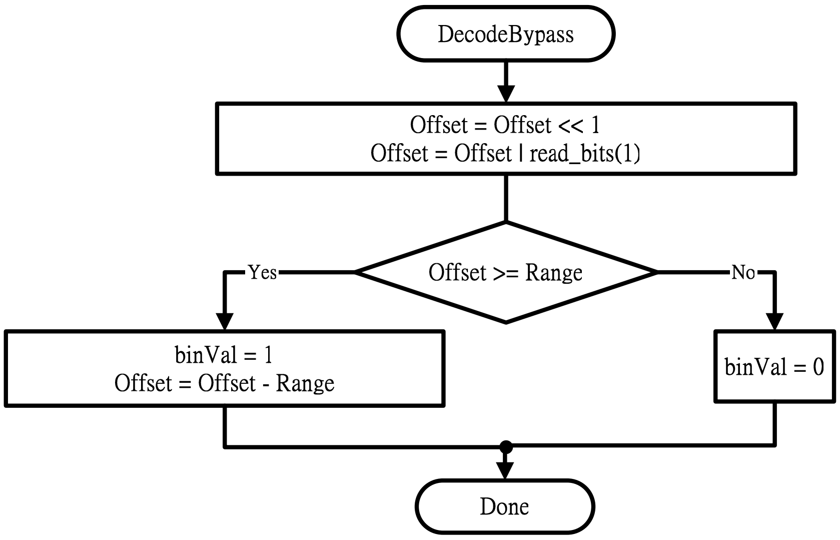

Figure2.12 Flowchart of bypass decoding

one bit to left. Then we insert a new bit from the sequence of encoding results in the least significant bit of low.

The bypass decoding engine is simplified from the regular decoding engine. Figure 2.12 shows the flowchart of the bypass decoding engine. There is no state index used in the bypass decoding engine. In the beginning of the bypass decoding engine, we first shift the offset and read a new bit from the sequence of ending result. Than we decide the bin value according to the comparison result of offset and range. If the value of the offset is bigger than the value of range, the decoding decision is 1 and the new value of the offset is the difference of original offset and range. Otherwise, the decoding decision is 0 and the value of the offset is the same as the original.

2.4 Previous work

In the previous papers, many architecture designs for arithmetic coder and decoder are proposed for different standard. In this section, we introduce the ideas in their architecture.

In [4], they proposed a 4 pipeline stage design for JPEG2000 MQ-coder. Figure 2.13 shows the block diagram of the architecture in [4]. The works of stage 0 and stage 1 are context update and range computation. The context index update in the next cycle. Because the precision of range register are 16 bits and the precision of register is 28 bits in JPEG2000, they pipeline the adder of low into two stages to reduce the path delay. The stage 2 computes the least significant 16 bits of low register. The stage 3 computes the most significant 12 bits of low register and generates the output bits. In [7], they proposed the extension architecture based on [4]. Their architecture can handle more

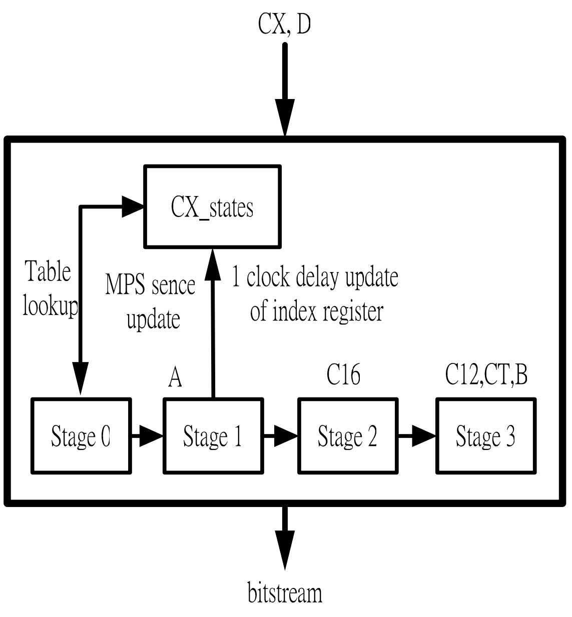

Figure 2.13 4-stage pipeline design of encoder in [4]

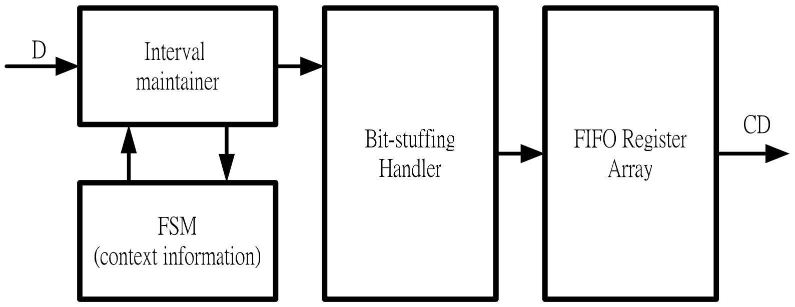

Figure2.14 Pipelined context-based AC encoding flow in [5] & [6]

than one symbol in a cycle. For reducing delay of critical path, they invert the sequence of some elementary operations. As a consequence, their have to use six branches to map the allowable cases. Their method is called the inverse multiple branch selection (IMBS) method.

In [5] and [6], an architecture of CABAC is proposed for multiple standard and JPEG2000. Their architecture is a 3 pipeline stages design. Figure 2.14 shows pipelined context-based AC encoding flow in [5] and [6]. In the stage 0, the interval computation and context information update is completed. In the stage 1, the bit-stuffing handler is in charge of bit-stuffing problem. The bit-stuffing handler uses buffer to detect the sequence "0xFF" in the result bits. If the bit-stuffing handler detects the sequence "0xFF", it will insert a bit after the sequence "0xFF". The inserted bit is the carry bit generated from previous stage. Because the output length of the bit-stuffing handler is 0, 1 or 2 bytes, they add a 4-byte FIFO register to limit the output length in one byte.

In [8], their architecture is based on H.264/AVC. Figure 2.15 shows the block diagram of arithmetic coder in [8]. Their arithmetic coder architecture is a 2 pipeline stage design. The first stage handles encoding iteration. In their design, they divide the original LPS table into four LPS tables which are independent of range. In the encoding iteration, they use a carry-save adder and prefix-adder to reduce the computation time of low and the range. In the renormalization, they use Leading-Zero detection in parallel

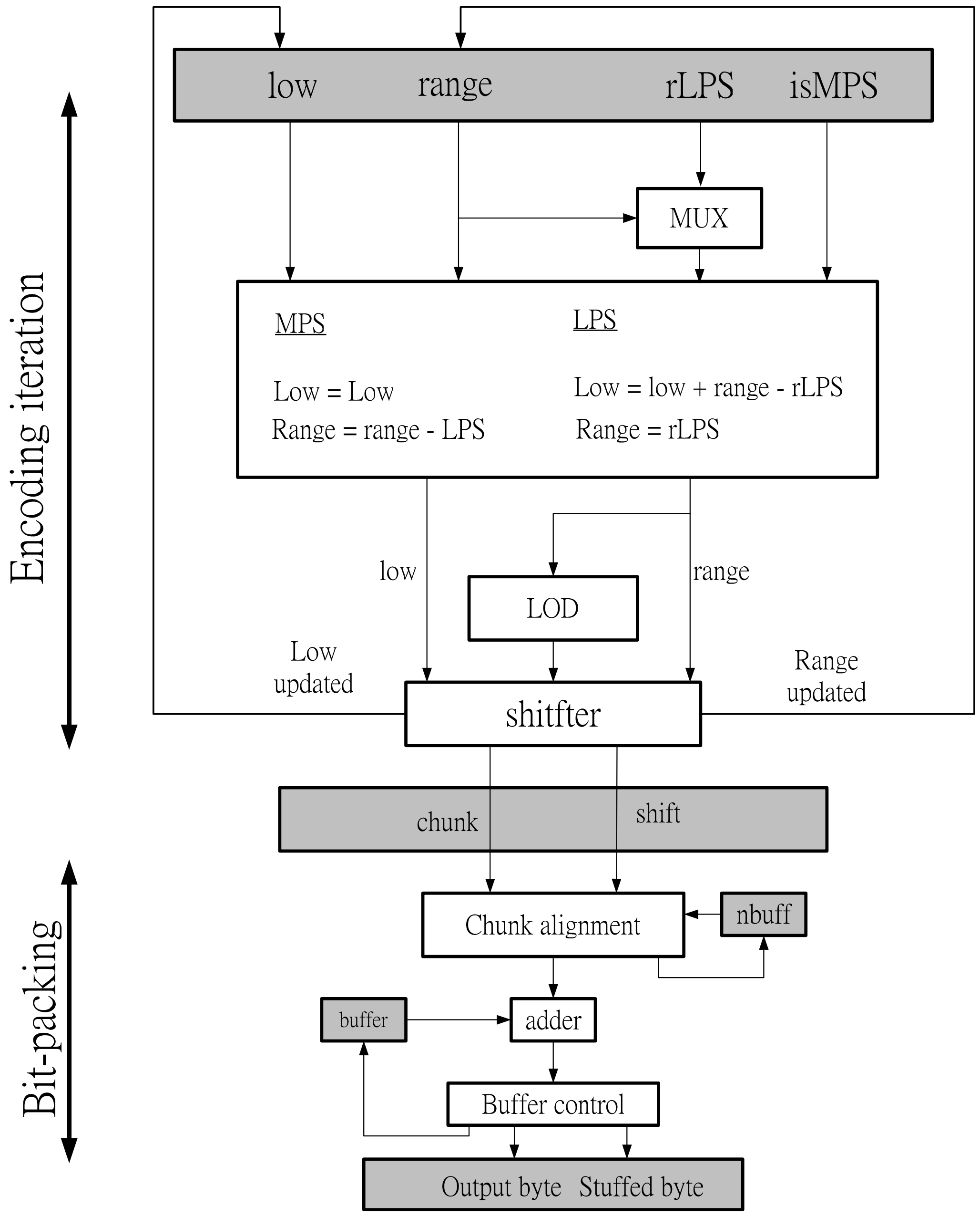

Figure 2.15 Block diagram of arithmetic coder in [8]

with the prefix adder to reduce the time for the renormalization. Stage 1 packs the output bits from stage 0 into a byte. The bit-packing stage also detect the sequence “0xFF” in the result to prevent the result is emitted by a carry propagation. If the bit-packing stage detects the sequence “0xFF”, they suspend the output process. Then count the total length of “0xFF” until further operations discard carry propagation.

In [10], they propose co-processor architecture on SoC platform. In the coder design, they use the MZ-coder instead of the M-coder in H.264/AVC. The MZ-coder provides equivalent bit rate comparison with M-coder. Furthermore, the MZ-coder eliminate the multiple renormalization cycles in M-coder. The coprocessor achieves a constant throughput for both encoding and decoding processes of 1 symbol per cycle.

In [11], they improve the bypass coding of CABAC in H.264/AVC. They use different hardware to handle the regular and the bypass mode coding. If a regular mode follows a bypass mode, their architecture can encode two symbols in the cycle. The probability of this situation is about 10%. Therefore their can increase 10% performance in average.

In the next chapter, we propose a novel architecture of arithmetic encoder and decoder for H.264/AVC. The architecture of encoder is a 3 pipeline stages design and the architecture of decoder is non-pipeline design. For reducing the path delay in the architecture, we rearrange the operations of range operations in the encoder and the decoder. Furthermore, in encoder we extend the architecture to encode more than one symbol per cycle.

Chapter 3

Architecture of Arithmetic Encoder and Decoder

In this chapter, we present hardware architectures for arithmetic encoder and decoder. First, we show a basic encoder architecture, which can encode one symbol per cycle. Then, based on the basic architecture, we further extend the design to support the encoding of multiple symbols per cycle. In the second section, we show the architecture of arithmetic decoder.

3.1 Encoding Architecture

Figure 3.1 shows the block diagram of the basic architecture, which includes 3 pipeline stages. In our architecture, we separate the operations of range and low into stage 0 and stage 1. In stage 2, the byte packing unit can pack the results into the format of byte. Furthermore, the architecture can support two encoding modes. Specifically, as illustrated in Figure 3.1, the stage 0 computes the value of range and update context

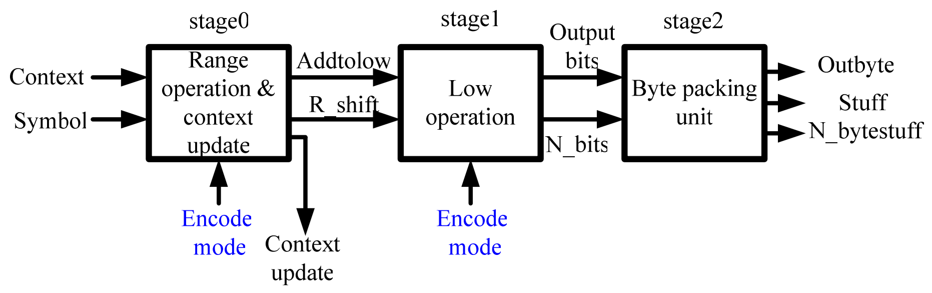

Figure 3.1 Pipeline stage of one-symbol encoding architecture

probability model. The stage 1 computes the value of low and generates output bits. The stage 2 groups the bits from the output of the stage 2 and packs them in a byte-by-byte manner. In addition, the bit-stuffing is also done in the stage 2.

To improve encoding throughput, we propose an extended architecture, which can encode multiple symbols per cycle. To achieve this, the operation of range is first reordered to reduce critical path. Then we duplicate the one-symbol encoding architecture and add additional hardware in each pipeline stage. The detail information will be shown in section 3.2.

3.1.1 One-symbol Encoding Architecture

In this section, we detail the design of each pipeline stage. For the stage 0, the operation includes two parts, which are the computation of range and the update of context probability model. Particularly, the critical path of the stage 0 is the computation of range, as shown in Figure 3.2 (a). We summarize the operations in the critical path as follows:

1. The table look-up of rLPS (range of LPS).
2. The subtraction for getting rMPS (range of MPS).
3. The renormalization.

For reducing the delay in the critical path, we rearrange the order of these operations. Originally, to produce the rLPS, the look-up table takes both the range and the probability of LPS, i.e., Qe., as input. For eliminating the data dependency between the rLPS and the range, we produce 4 sub-tables by unrolling the cases of range. After the unrolling, each sub-table simply takes Qe as input, as shown in Figure 3.2 (b). Then, the renormalization of previous iteration and the table-lookup of rLPS can be done in parallel, as shown in Figure 3.2 (c). Lastly, we can reorganize the operations in the iteration, as shown in Figure 3.2 (d).

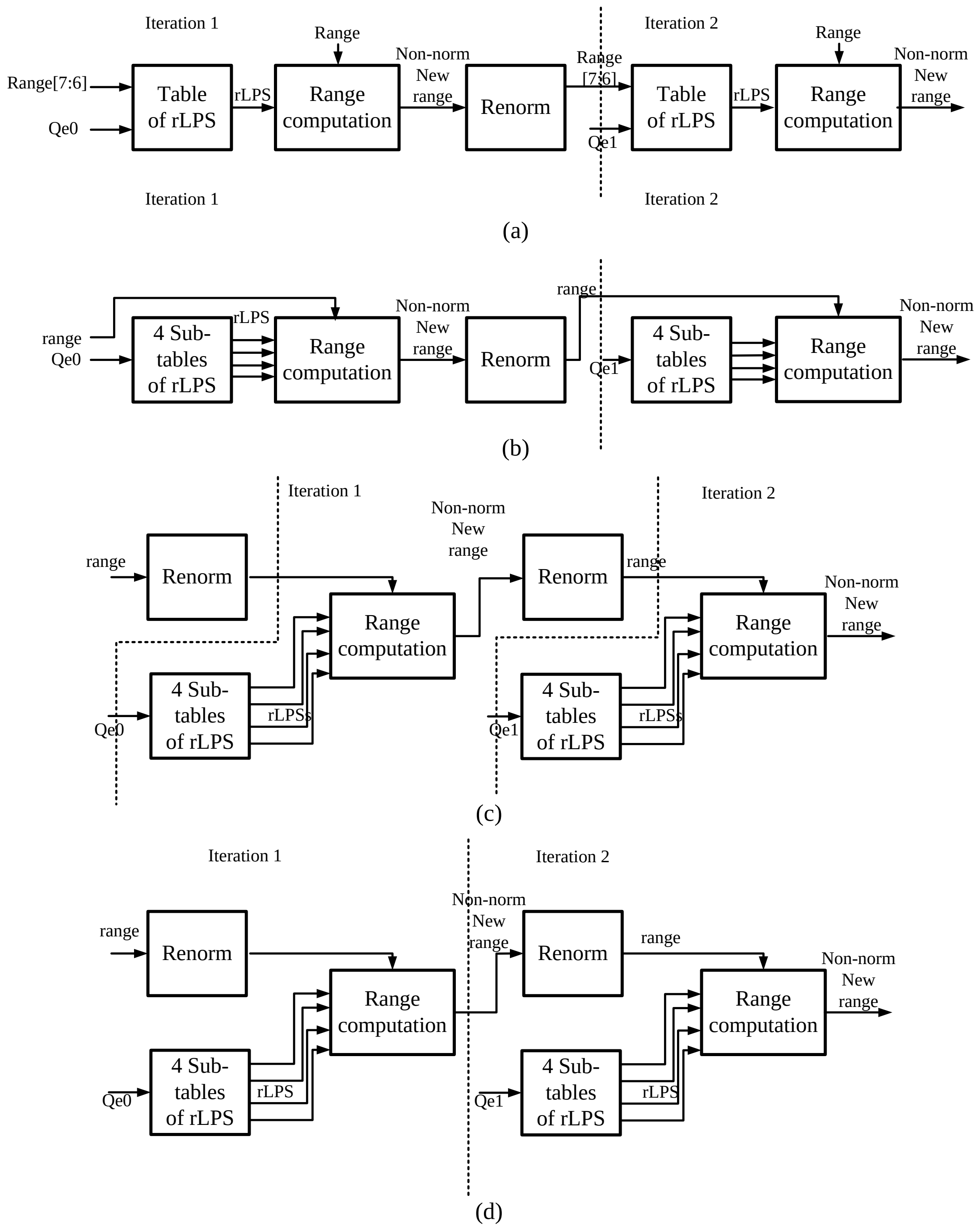

Figure 3.2 Path of range in encoding iteration.

The detailed block diagram of the stage 0 is shown in Figure 3.3. As shown, the stage 0 has three input signals, one output signal, and three intermediate signals for the next stage. The meaning of each symbol is elaborated as follows:

- The signal “Symbol” means the encode symbol.

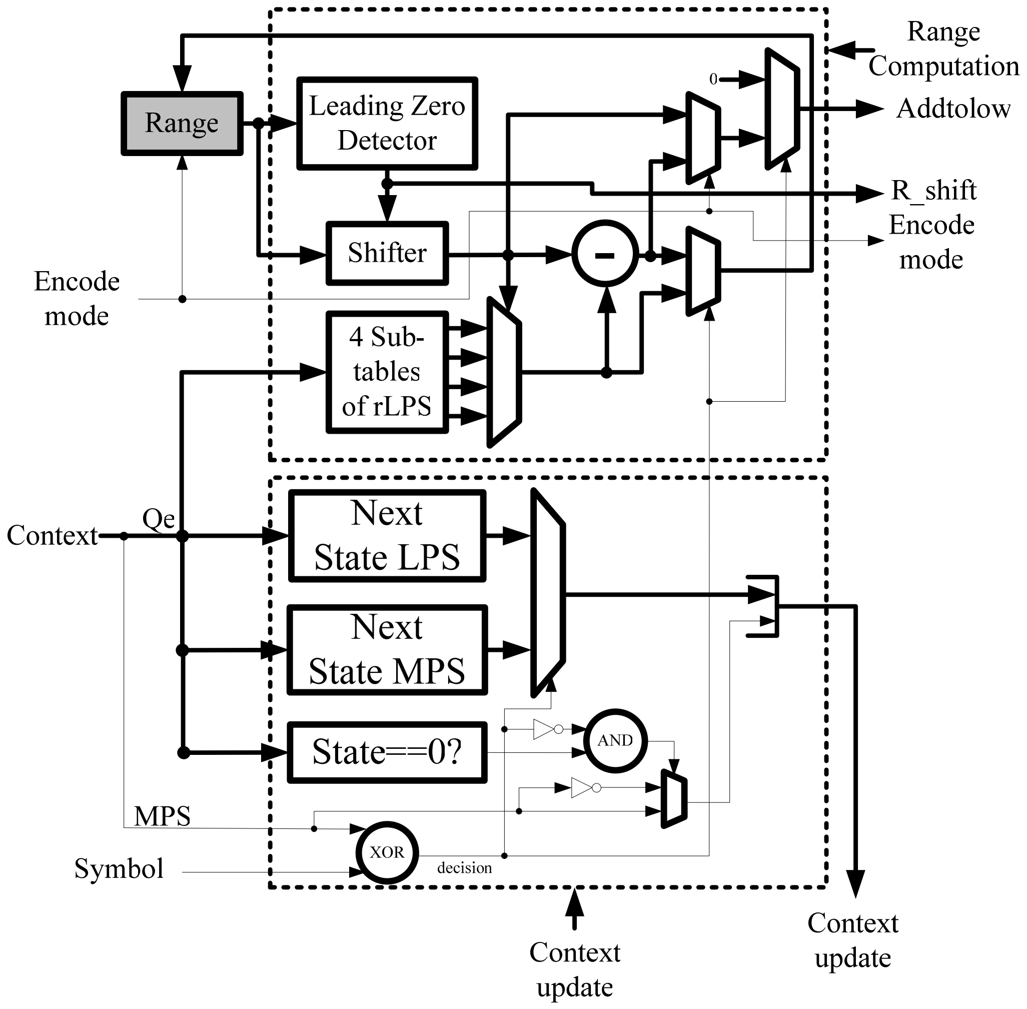

Figure 3.3 Detailed block diagram of stage 0.

- The signal "Context" means the context probability model that includes the Qe and the MPS.
- The signal "Encode mode" specifies whether the coding is in the regular mode or the bypass mode.
- The other two signal passing to the next stage means the value that will add with low and the numbers of output bits in this encode process.
- The output signal "Context update" is used to update the context information.

For supporting the bypass mode in the stage 0, the register of range and the signal "Addtolow" is controlled by the signal "Encode mode". When the encoding is in bypass mode, the range will remain un-changed and the signal "Addtolow" will take the value

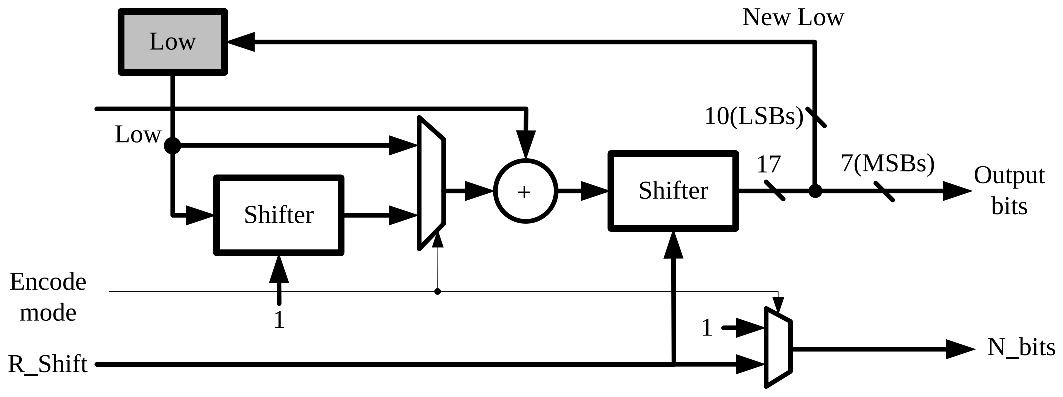

Figure 3.4 Block diagram of the stage1 in basic architecture.

of range. Then, the signal "encode mode" will be passed to the next stage.

In the stage 1, the main operations include the computation and the renormalization of low. Figure 3.4 shows the detailed block diagram of the stage 1. As shown, the stage 1 takes the intermediate signals produced by the stage 0 as input and produces "Output bits" and "N_bits". Respectively, the signals "Output bits" and "N_bits" stand for the encoding result of one symbol and the associated number of coded bits. Particularly, when the coding is in the bypassing mode, the value of low will be firstly shifted to the left by one bit and the total shift value will be set to 1.

In the stage 2, the encoding results, i.e., the compressed bits, from the stage 1 are packed in a byte-by-byte manner. In [7], the byte-stuffing technology is used for packing. To detect the occurrence of 0xFF sequence, a 16-bit buffer is used to buffer the compressed bits. When the value of 0xFF is detected and identified, there are possibilities that the carry propagation will affect the byte that has been outputted. Therefore, we need to hold the output byte and use a register to store the length of stuffed bytes. The operation in packing buffer is shown in Figure 3.5. In the beginning, the second byte in Pbuffer is 0xFF. Then we store the length of the byte 0xFF in the register "N_bytestuff" and the bit value of the byte 0xFF in the register "Stuff". After storing the information of stuff situation, we continue the process of byte packing. If the

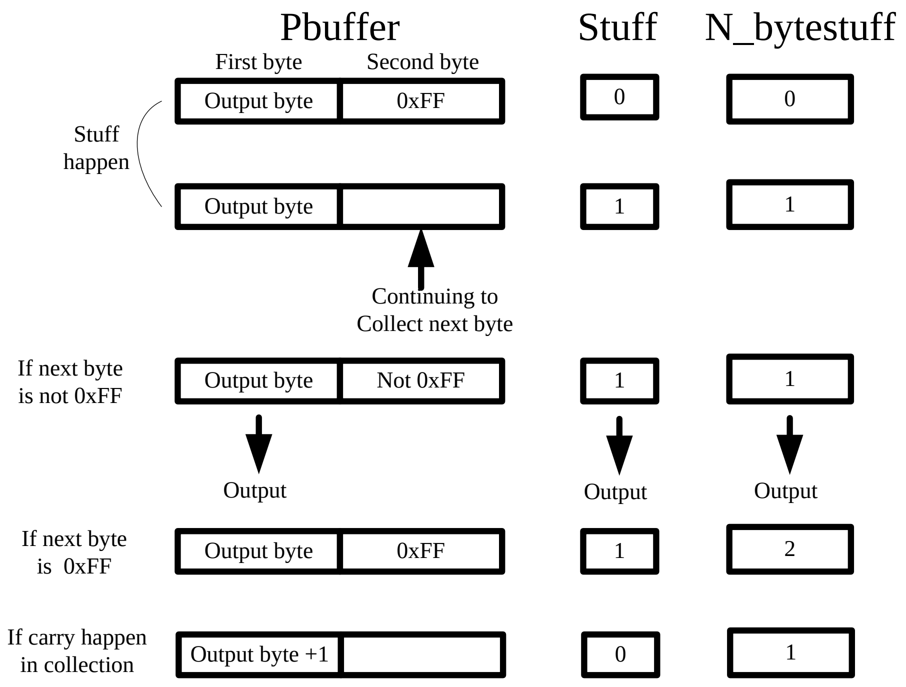

Figure 3.5 Operation in packing buffer.

next byte is not 0xff, stage 2 will output the first byte in the packing buffer, the value in the register "Stuff" and the value in the register "N_bytestuff". On the other hand, if the next byte is 0xff, the register "N_bytestuff" will be increased by 1. If the following operations produce a carry signal, the register "Stuff" will be turned to 0 and the first byte of packing buffer will be increased by 1.

Figure 3.6 shows the block diagram of the stage 2. The register "Pbuffer" stores the encoding results. The register "N_Pbuffer" records the number bit of results in the "Pbuffer". The combination of the registers "Stuff" and "N_bytestuff" specifies the information of byte stuff. Upon the detection of a "0xFF" byte, the register "stuff" records the content of stuffing bits and the "N_bytestuff" specifies the number of the bytes that have the value of "0xFF". Until the next byte is not "0xFF", the stuffing information is output with the signal "Outputbyte".

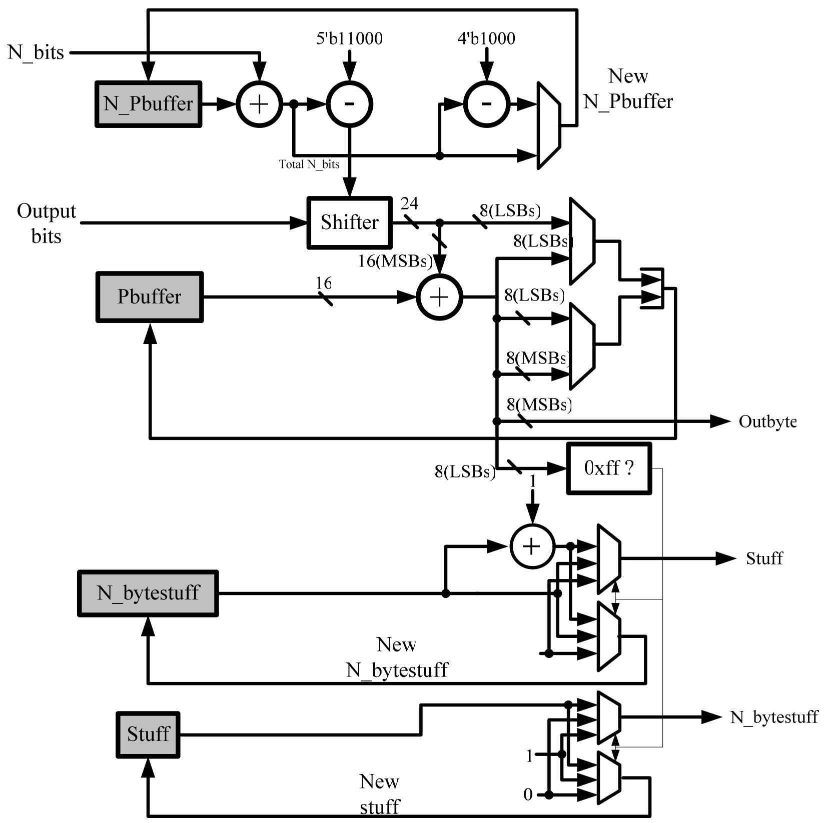

Figure 3.6 Block diagram of the stage2 in the basic architecture.

3.1.2 Multiple Symbols Encoding Architecture

For improving the performance of AC encoding, we propose an encoding architecture that is capable of coding multiple symbols per cycle. While maintaining similar or higher coding performance, our scalable architecture provides the flexibility to adjust clock rate by changing the number of coding symbols per cycle.

Figure 3.7 shows the block diagram of scalable architecture. To encode more than one symbol per cycle, we duplicate the one-symbol encoding architecture and add additional hardware in each pipeline stage. For encoding ***n*** symbols per cycle, we duplicate the one-symbol encoding architecture by ***n*** times in the stages 0 and 1. As shown in Figure 3.7, the one-symbol encoding unit in the stages 0 and 1 includes the range operation, the context update, and the low operation. For multi-symbol encoding, these functional units are duplicated. After encoding the symbol, the values of range and low are passed to the next functional unit. In the stage 0, if more than two encoding symbols use the same context probability model, the later encoding symbols will use the context probability model after the update. Therefore the context information needs a multiplexer to choose the correct one. In the stage 1, the number of result produced by low operation is variable. For reducing the workload and complexity of the stage 2, we combine all the results before passing the data to the stage2.

In the stage 2, we insert a small input buffer to support the multi-symbol encoding. The basic byte packing unit can process 8 bits in one cycle. For coding one symbol, the average number of results from the stage 1 is less than 1. As we extend the design for multiple-symbol encoding, the probability for the total input number being greater than 8 is very small. Such an exception only occurs a few times for each video frame. Thus, we insert a small buffer in front of the stage 2. The input buffer limits the number of input bits to 8 bits. As a result, using an input buffer can maintain the same structure of byte packing unit in the stage 2.

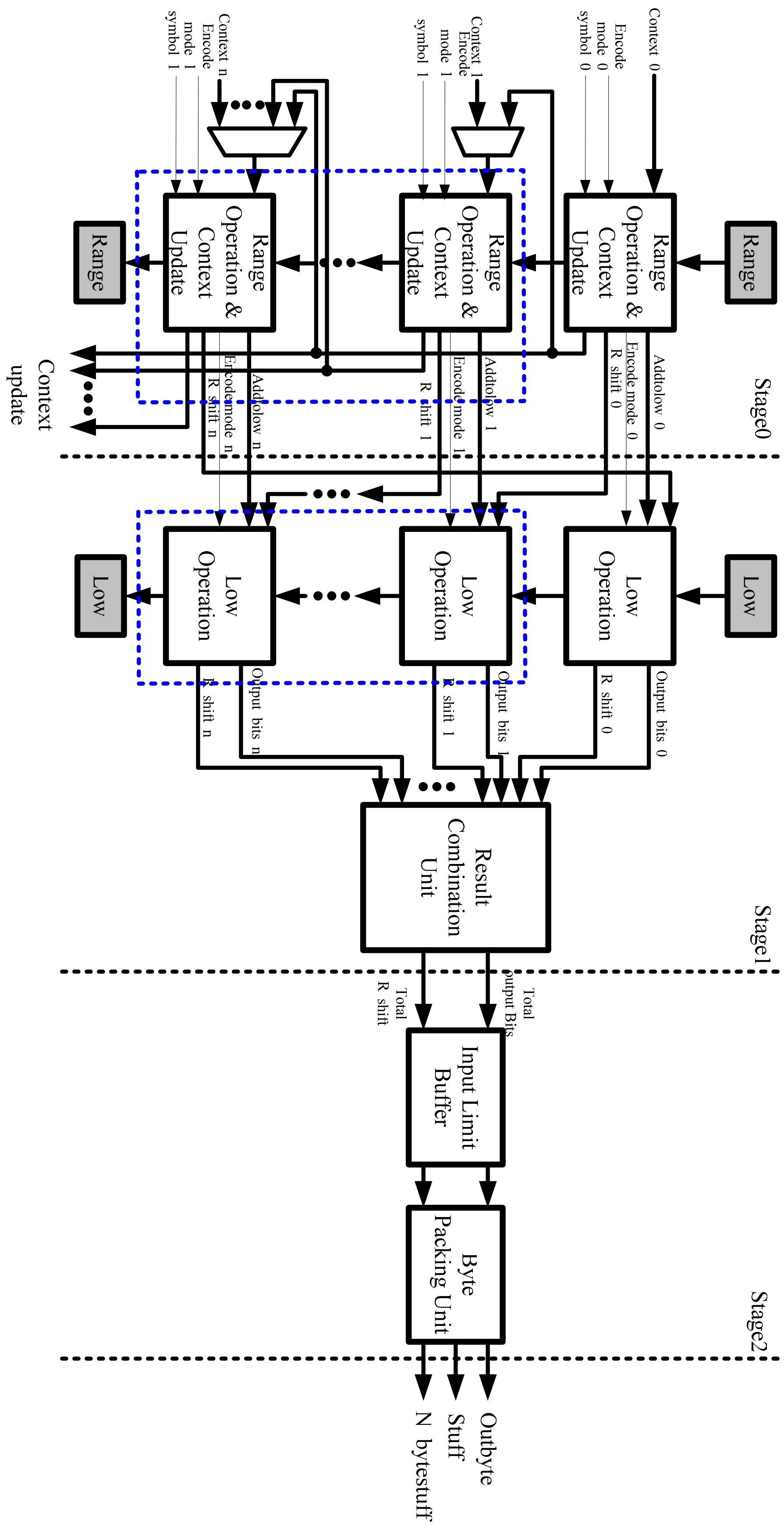

Figure 3.7 Block diagram of the multiple symbols encoding architecture

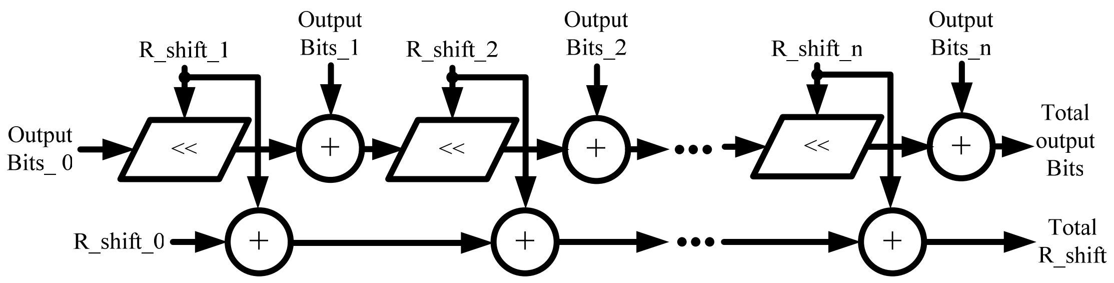

Figure 3.8 Block diagram of result combination unit

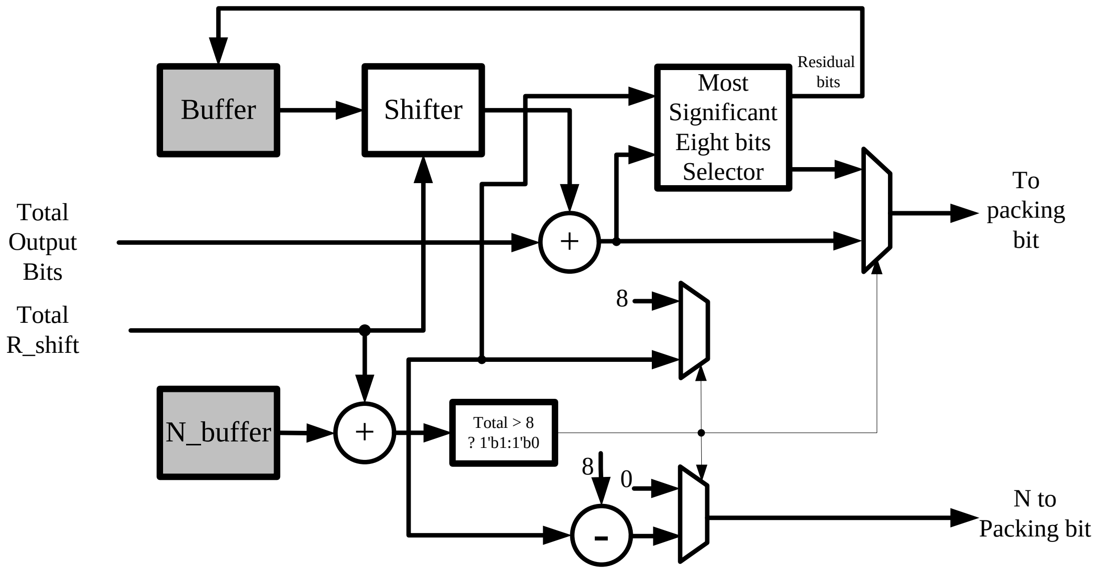

Figure 3.9 Block diagram of input limit buffer

Figure 3.8 details the unit for result combination. The result combination unit consists of shifters and adders. There are two kinds of input signals in Figure 3.8. The signal “Output bits_i” (i=0,1,…,n) means the encoding result of one symbol and the signal “R_shift_i” (i=0,1,…,n) denotes the length of encoding result. For combining the results produced by different low operations, we first shift the previous encoding result to the correct position. Then we use adders to combine all the result bits. By this way, the result combination unit can output the total number of result bits and a sequence of result bits.

Figure 3.9 shows the block diagram of the input limit buffer. There are two input signals, two output signals, and two local registers. The register “Buffer” temporarily stores the residual bits if the length of previous packing bits is greater than 8. The register “N_buffer” records the number of bits that are stored in the register “buffer”. If the buffer is not empty, we combine the bits in the buffer and the input bits. Then we check if the total number of bits is greater than 8. As the total number of bits is greater than 8, we will select the first 8 MSB bits of the combined result as the output and keep the residual bits in the buffer. In the opposite case, we will pass the bits directly to the byte packing unit.

3.1.3 Multiple Standard Support

In addition to supporting multi-symbol encoding, our structure can also be easily tailored to support the arithmetic encoding in JPEG2000. Table3.1 summarizes the difference of the arithmetic coder in H.264/AVC and JPEG2000. There are three major differences, which are (1) the method for getting rLPS, (2) the operations of low and range, and (3) the precision for representing range and low. In JPEG2000, rLPS simply depends on Qe. However, in H264/AVC, rLPS is from both Qe and range. For the operations of low and range, JPEG2000 updates the low by adding the value of rLPS, as

Table 3.1 Differences of arithmetic coder in H.264/AVC and JPEG2000

	H.264/AVC	JPEG2000
rLPS	table[Qe][range[7:6]]	Qe
rMPS	range - rLPS	range - rLPS
operations when symbol == MPS	low_{new} = low $range_{new}$ = rMPS	low_{new} = low + rLPS $range_{new}$ = rMPS
operations when symbol == LPS	low_{new} = low + rMPS $range_{new}$ = rLPS	low_{new} = low $range_{new}$ = rLPS
range register precision(bits)	9	16
low register precision(bits)	10	28

the input symbol is MPS. On the other hand, in H.264/AVC, the low is updated by adding the value of rMPS, as the input symbol is LPS. Lastly, the precision for representing the range and low is different. Specifically, JPEG 2000 requires higher precision for the range and low.

To support JPEG2000, our design is modified to adopt these differences. More specifically, when the coding is for JPEG2000, we remove the 4 sub-tables of LPS and directly connect the Qe to the range compute unit. In the encoding operation, we change the value, which we prepare to add to low, from rMPS to rLPS. Then the timing of adding will be change from that symbol equals LPS to that symbol equals MPS. Lastly, we use high-precision registers and adders to fulfill the need of JPEG2000. Without changing the architecture significantly, our design can be slightly extended to support JPEG2000.

3.2 Decoder Architecture

In this section, we illustrate the architecture of binary arithmetic decoder. For the decoding, our architecture can decode only one symbol per cycle. Different from the case of encoder, at the decoder, the context index for a symbol can only be certain when the previous symbol is decoded. Because of strong data dependency and insufficient context information, it is more difficult to decode multiple symbols per cycle. Thus, our proposed decoder architecture is not pipelined.

To reduce the delay of computation, we apply the same reordering technique in the encoder. Figure 3.10 (a) shows the critical path in the straightforward implementation. As shown, the longest delay is for the computation of new offset value. The offset value is determined after the range. For completing the computation of range, it needs four steps. The first two steps in the decoding process are similar to those in the encoding

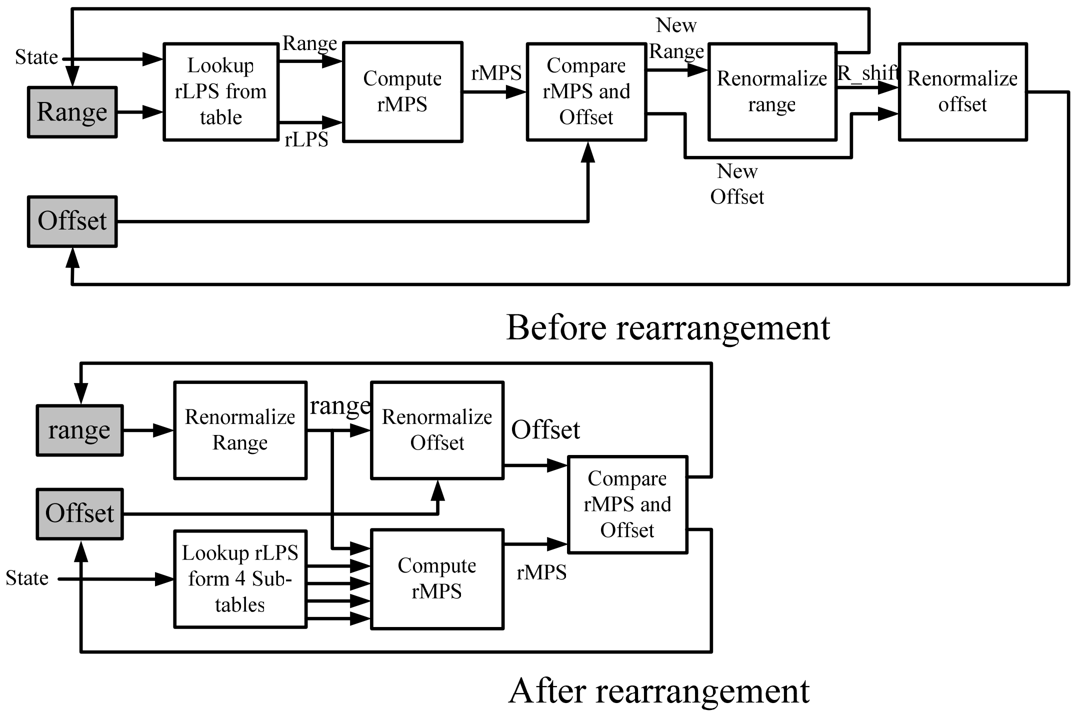

Figure 3.10 Data path of range in decoder

process. The rMPS and rLPS are known in the first two steps. When rMPS is known, we can compare the value of rMPS with the offset to make the decision. After the comparison, new range and offset are determined. Thus, we renormalize the range and the offset. For reducing the critical path, we employ the same reordering technique used for the encoder. As shown in Figure 3.10 (b), the registers of range and the offset store the value without renormalization. Also, we use 4 sub-tables to eliminate the data dependency between rLPS and range. After the reordering, Figure 3.10 (b) illustrates the proposed decoder architecture.

Figure 3.11 shows the detail architecture of decoder. Basically, the decoder has four input signals and three output signals. The signal “Bits_in” replenishes the least significant bits of the offset value during the renormalization. The encode mode indicates whether the decoding process is in regular or bypass mode. The state index and MPS form the context information. The signal “Output bit” is the decoding result and the signal “Qe update” and the signal “MPS update” update the context probability

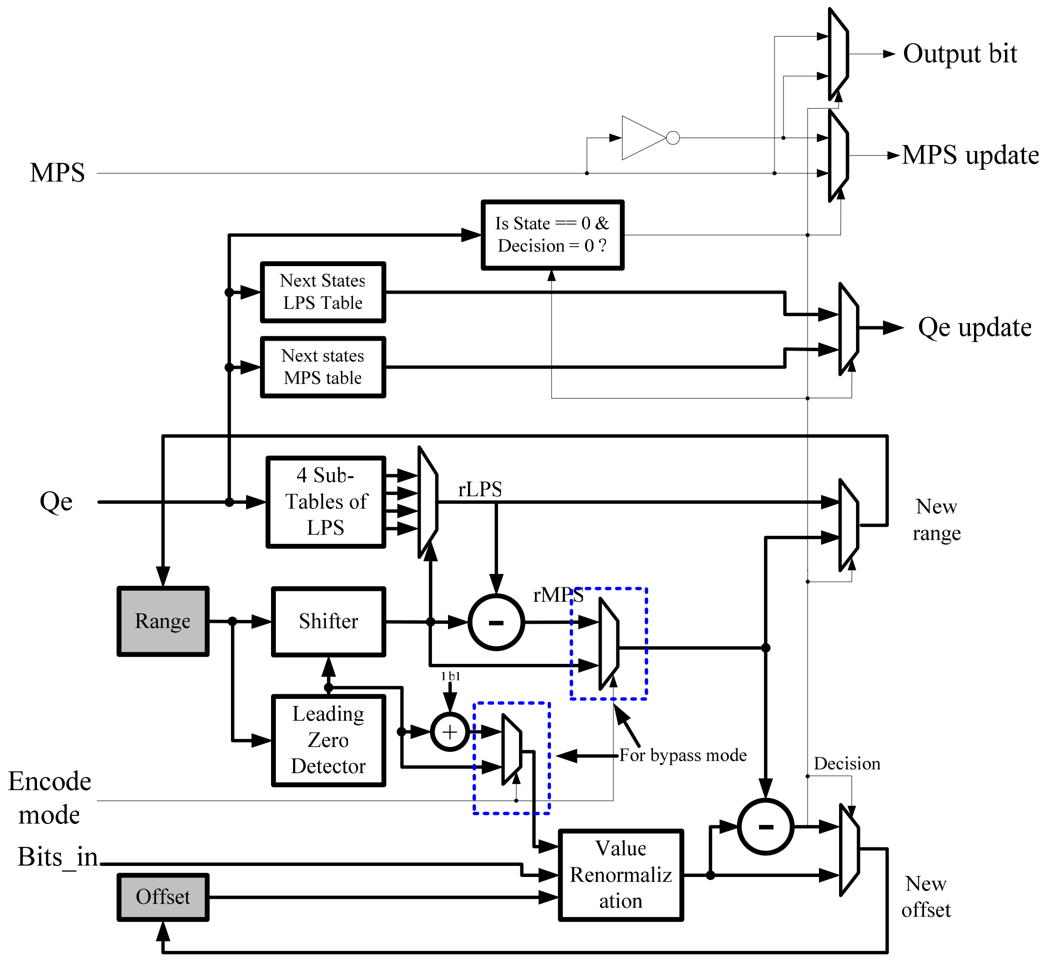

Figure 3.11 Architecture of the AC decoder

model. To support the bypass decoding, two additional multiplexers are deployed in this design, as shown with the dash blocks in Figure 3.11. In the bypass mode, we make the decision according to the range and double offset. The first multiplexer choose rMPS or range according to encoding model. The second multiplexer chooses the double values of offset or the value of offset to decode. But, in our design, the renormalization is done in the beginning. Therefore, we can combine the shift of renormalization and the shift of double low. When bypass decoding, the shift value of offset will be increased by 1.

Chapter 4

Implementation

4.1 Design flow and verification

In this section, we detail the design flow and the verification environment. In the beginning of the design, we build the C model for the proposed architecture. The C model is used not only for analysis but also for debugging in RTL-level design. After the C model design and verification, the design and the simulation of the register transfer level (RTL) level start. The hardware design in RTL is represented by verilog hardware description language (HDL). After the RTL simulation, the RTL verilog code of the design will be synthesized and optimized into gate level. Then the gate level simulation starts. After the gate level simulation, the verification move to the FPGA environment.

Figure 4.1 shows our environment of FPGA. The multi ICE connects the FPGA board with the PC. On the board, there are logic and core modules. The ARM 966 CPU, the SDRAM controller and embedded SRAM is covered by the core module. The proposed design is downloaded to logic module which is made of FPGA. We download our designs through the multi ICE to the FPGA. During the simulation, the compiled code is put in the embedded SDRAM and the image data is put in the external SDRAM. Our designs act as slaves on the bus. The ARM 966 CPU, which is a master on AHB, can access the slaves via AHB. The ARM966 CPU can use the accelerator of CABAC through the AHB.

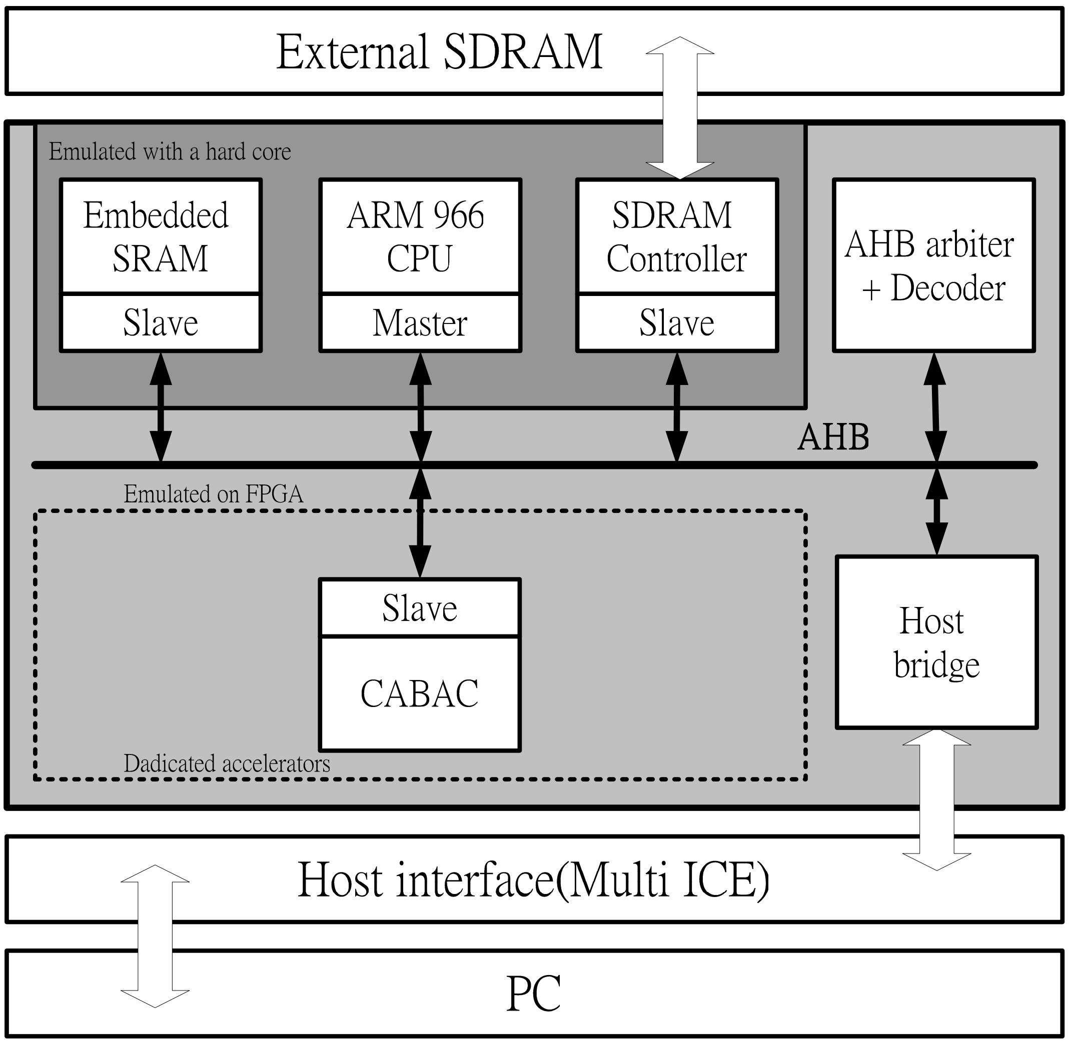

Figure 4.1 FPGA environment

In the FPGA environment, we run the codec of H.264 to verify our CABAC encoder/decoder design. The architecture of CABAC encoder and decoder has passed the verification.

4.2 Implementation Results

Table 4.1 shows the synthesis results of the encoder and decoder. The data of the results is from Design Analyzer. The one-symbol encoding architecture can achieve 370M symbols per second. As the architecture extends to encode two and three symbols per cycle, the performance achieves 526M and 546M symbols per second. Furthermore, the performance of decoder can achieve 333M symbols per second. Although the gate

Table 4.1 Results of encoder and decoder design

Designs	Encoder			Decoder
Number of Encoding/Decoding bits per Cycle	1	2	3	1
Maximum Frequency	370	263	182	333
Gate Count (0.18um)	3.7k	7.8k	9.3k	3.5k
Maximum Throughput (Mega-symbols/s)	370	526	546	333

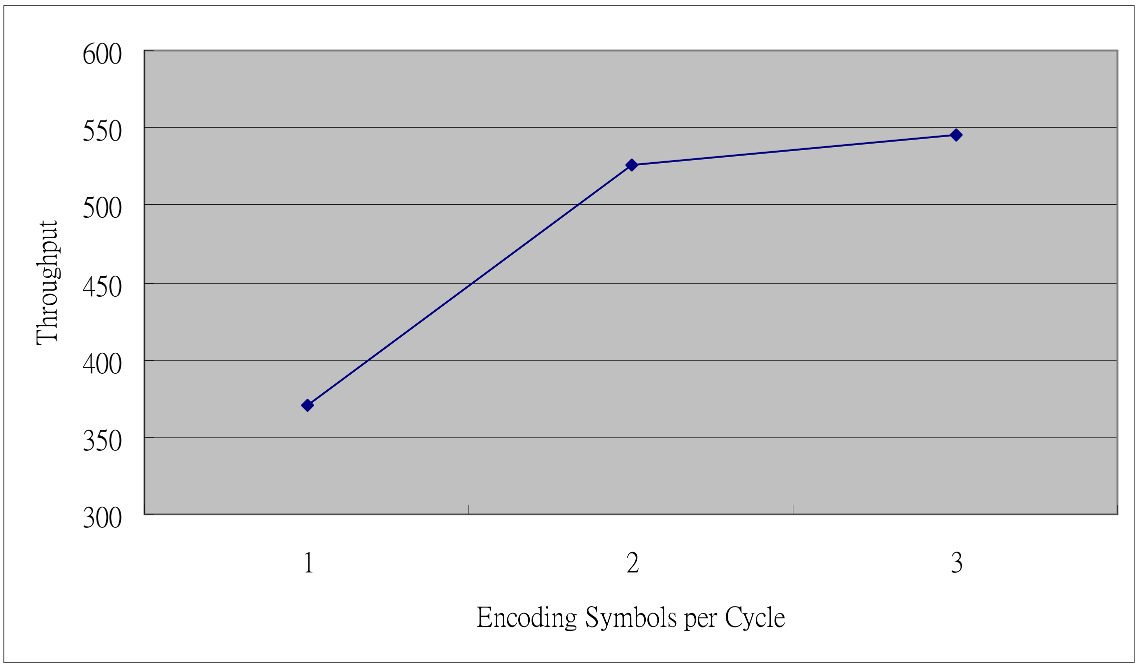

Figure 4.2 Encoding Symbols per Cycles VS Throughput

counts of two-symbol encoding architecture is 2.1 times than that of one-symbol encoding architecture, the performance of two-symbol encoding architecture is 1.4 times faster than one-symbol encoding architecture. When the one-symbol encoding architecture extends to encode three symbols per cycle, the performance achieves the maximum and the gate counts of three-symbol encoding architecture is only 2.5 times faster than one-symbol encoding architecture.

The maximum throughput saturates when the encoding symbols per cycle increase. As figure 4.2 shows, we can find that the maximum throughput is limited in 550M symbols per cycle. The critical path increases with the encoding number per cycle. The increase of critical path causes the saturation of maximum throughput.

4.3 Comparison

Table 4.2 lists the results previous arithmetic encoder design for H.264/AVC with ours. The designs of [4], [5], [6], and [7] are designed for different standard and implemented in old technology. For fair comparison we do not list the results of their designs in the table 4.2. As the table 4.2 shows, our maximum throughput is 1.65 to 2.87 times than the designs of [8], [9]. [10].

Table 4.3 lists the results of our decoder designs and prevous arithmetic deocder designs for H.264/AVC. In [2] and [3], they have decoder designs, but the designs are used for different standard and implement in old technology. In table 4.3, although the maximum throughput of our design is the same as the design of [10], our technology is behind theirs. Therefore the result in our design is better than the result in [10].

Table 4.2: AC encoder performance comparison

Design	Standard	Number of Encoding Symbol per Cycle	Gate Count	Throughput (Mega-symbols/s)
My design	H.264/AVC	1,2,3	3.7k,7.8k,9.3k (0.18um)	370,526,546
[8]	H.264/AVC	1	N/A Estimation(0.18um)	200
[10]	H.264/AVC	1	N/A (0.13um)	330
[11]	H.264/AVC	1	N/A (0.18um)	190

Table 4.3: AC decoder performance comparison

Design	Standard	Gate Count	Throughput (Mega-symbols/s)
My design	H.264/AVC	3.5k(0.18um)	330
[10]	H.264/AVC	N/A (0.13um)	330

Chapter 5

Conclusion

In this thesis, we propose a scalable architecture for the encoding and decoding of the context-adaptive binary arithmetic coder (CABAC) in H.264/AVC.

There is strong data dependency in the algorithm of CABAC. In the encoding and decoding process, the value of range cannot be used until the process of previous symbol complete. Therefore, the critical path will determine the performance of the designs. For reducing the delay of critical path, we reorganize the range operations in encoding and decoding iteration. The new order of range operations can parallel the process.

For high performance requirement, the architecture can be extended to encode more than one symbol per cycle. For extension, we duplicate the basic functions in the stages 0 and 1. Then we respectively add result combination unit and an input limit buffer in the stages 1 and 2. The result combination unit can combine the result of different encoding symbols. The input limit buffer can increase the capacity of byte packing unit. Then the byte packing unit will not be changed. Moreover the basic operations of Arithmetic coder in JPEG2000 and H.264/AVC are similar. The design can be modified to support JPEP2000 easily.

We use 0.18 CMOS technology to synthesize our designs. The maximum throughput of the three-symbol arithmetic coder is 545M symbols per second and the maximum throughput of decoder is 330M symbols per second. Besides, we verify our designs on the environment of FPGA. All the designs work with H.264 encoder or decoder correctly.

The context memory design is future work. As the proposed architecture encodes multiple symbols per cycle, it will need a multi-port memory to support. A context memory with low cost and high performance will be an important problem.

Bibliography

[1] http://www.packetizer.com/codecs/h264/trev_293-schaefer.pdf

[2] D. Marpe, H. Schwarz, T. Wiegand; "Context-Based Adaptive Binary Arithmetic Coding in the H.264/ABC Video Compression Standard", *IEEE Trans on Circuits and Systems for Video Technology*, July 2003

[3] Text of ISO/IEC 14496-10:2004 Advanced Video Coding (second edition).

[4] H.-H Chen, C.-J Lian, K.-F Chen, L.-G Chen, "Context-based Adaptive Arithmetic Encoder Design for JPEG 2000", *VLSI Design/CAD Symposium*, 2001.

[5] K.-K Ong, W.-H Chang, Y.-C Tseng, Y.-S Lee, C.-Y Lee "A High Throughput Low Cost Context-based Adaptive Arithmetic codec for Multiple Standards", *Image Processing*, 2002

[6] K.-K Ong, W.-H Chang, Y.-C Tseng, Y.-S Lee, C.-Y Lee,"A high throughput context-based adaptive arithmetic coder for JPEG2000", *Circuits and Systems, 2002. ISCAS* 2002

[7] Grzegorz Pastuszak, "A novel architecture of arithmetic coder in JPEG2000 based on parallel symbol encoding", *Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'04)*

[8] Osorio, R.R.; Bruguera, J.D,"Arithmetic coding architecture for H.264/AVC CABAC compression system", *Digital System Design, 2004. DSD* 2004

[9] C,-J. Lian, K-F. Chen, H.-H. Chen, and L.-G Chen, "Analysis and architecture design of block-coding engine for EBCOT in JPEG2000," *IEEE Trans. Circuits and System for Video Technology*, March 2003.

[10] J. L. Núñez, V. A. Chouliaras, "High-performance Arithmetic coding VLSI Macro for the H.264 Video Compression Standard", Consumer Electronics, IEEE

Transactions, Vol. 51, ISSUE 1,pp. 144-152Feb. 2005

[11] Hassan Shojania, Subramania Sudharasanan, “A VLSI Architecture for High Performance CABAC Encoding”, Visual Communications and Image Processing, pp. 1444-1454, 2005