國立交通大學

資訊科學與工程研究所

碩士論文

H.264 CAVLC/CABAC 熵解碼整合型 IP 設計

Design of an Unified Entropy IP for H.264 CAVLC/CABAC Decoding

研究生：陳奕岑

指導教授：蔡淳仁 教授

中華民國九十七年七月

H.264 CAVLC/CABAC 熵解碼整合型 IP 設計

Design of an Unified Entropy IP for H.264 CAVLC/CABAC Decoding

研究生：陳奕岑　　Student：Yi-Tsen Chen

指導教授：蔡淳仁　　Advisor：Chun-Jen Tsai

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年七月

Acknowledge

在研究所2年學習的時間裡, 很感謝我的指導教授蔡淳仁博士, 在老師的身上我學習到做研究應有的嚴謹態度, 以及對各種小細節的注意, 很感謝老師的耐心指導, 讓我不管是在學術上, 生活上都學習到做任何事情都應該要全力以赴認真去執行. 也謝謝實驗室的同學們大家一起吃飯玩樂, 讓我們的研究生活不至於過於苦悶, 最後我想謝謝我的家人, 因為有家人的支持, 我才能無憂的念書, 最後很高興能成為MMESLAB的一員, 真的很感謝大家.

E S A 1896

Abstract

In this thesis, we designed a synthesizable RTL model of the entropy decoder (CAVLC and CABAC) for the AVC (a.k.a. H.264) video coding standard. The design has been verified on the Xilinx Vertex 5-based FPGA development board, ML506, using full system verification with the AVC/H.264 reference software JM 12.2. The size of the combined CAVLD and CABAD logic is reasonably small. It only occupies about 7000 slices (21% logic resource of the target device). At a clock rate of 50MHz, the performance of the design can achieve decoding of bitrates over 11 mbps for CAVLD and 8 mbps for CABAC.

Table of Contents

Chapter 1. Introduction 1

1.1. Motivation 1

1.2. Target Platform 2

1.3. Outline of the Thesis 4

Chapter 2. Previous Work 5

2.1. CAVLD 5

2.2. CABAD 6

Chapter 3. Design of the CAVLD Logic 9

3.1. Introduction to the CAVLD Entropy Decoder 9

3.1.1. Algorithm of CAVLD 9

3.2. Overall System Architecture 11

3.2.1. AMBA AHB BUS Wrapper for CAVLD 11

3.2.2. CAVLD Top Module 13

3.3. State Controller 15

3.3.1. Interface of State Controller 15

3.3.2. States of the Controller 18

3.3.3. Barrel Shifter 20

3.4. Bitstream Preprocessing Logics 22

3.4.1. First One Detector 22

3.4.1.1. Interface of First One Detector 22

3.4.1.2. Architecture of First One Detector 23

3.4.2. Switch Input Module 24

3.4.2.1. Switch_input_1 Module 24

3.4.2.1.1. Interface of Switch_input_1 Module 24

3.4.2.1.2. Architecture of switch_input_1 Module 25

3.4.2.2. Switch_input_2 Module 26

3.4.2.2.1. Interface of switch_input_2 Module 26

3.4.2.2.2. Architecture of switch_input_2 Module 27

3.5. Coeff_token Decoder 27

3.5.1. Interface of Coeff_token Decoder 27

3.5.2. Architecture of Coeff_token Decoder 29

3.6. Trailing_ones Calculator 31

3.6.1. Interface of Trailing_ones Calculator 31

3.6.2. Architecture of Trailing_ones Calculator 32

3.7. Level Decoder 33

3.7.1. Interface of Level Decoder 33

3.7.2. Architecture of Level Decoder 35
3.8. Total_zeros Decoder 40
3.8.1. Interface of Total_zeros Decoder.. 40
3.8.2. Architecture of Total_zeros Decoder 41
3.9. Run_before Decoder.. 43
3.9.1. Interface of Run_before Decoder 43
3.9.2. Architecture of Run_before Decoder....................................... 45
3.10. A CAVLD Decoding Example 46
Chapter 4. Design of the CABAD Logic 53
4.1. Introduction to the CABAD Entropy Decoder.................................. 53
4.1.1. Algorithm of CABAD.. 53
4.2. Overall System Architecture 54
4.2.1. AMBA AHB BUS Wrapper for CABAD................................. 54
4.2.2. CABAD Top Module 55
4.3. Binary Arithmetic Coding 58
4.3.1. Normal Decoding Process (AC_Regular_mode)....................... 59
4.3.1.1. Interface of Normal Decoding Process...................... 60
4.3.1.2. Flow Chart of Normal Decoding Process 62
4.3.1.3. Architecture of Normal Decoding Process 62
4.3.2. Bypass Decoding Process (AC_Bypass_mode) 67
4.3.2.1. Interface of Bypass Decoding Process 68
4.3.2.2. Flow Chart of Bypass Decoding Process................... 68
4.3.2.3. Architecture of Bypass Decoding Process................. 69
4.3.3. Final Decoding Process (AC_Final_mode)............................... 71
4.3.3.1. Interface of Final Decoding Process.......................... 71
4.3.3.2. Flow Chart of Final Decoding Process...................... 72
4.3.3.3. Architecture of Final Decoding Process 73
4.4. Initialization for context variables.. 76
4.4.1. Interface of Initialization for context variables 76
4.4.2. Architecture of Initialization for context variables.................... 77
4.5. State Controller for each Syntax Element Controller 79
4.6. MB Skip Flag 81
4.6.1. Interface of MB Skip Flag.. 81
4.6.2. Flow Chart of MB Skip Flag 84
4.7. MB Type.. 84
4.7.1. Interface of MB Type 84
4.7.2. Flow Chart of MB Type in I_slice 85
4.7.3. Flow Chart of MB Type in P_slice 87

4.7.4. Flow Chart of MB Type in B_slice 88

4.8. Sub MB Type 91

4.8.1. Interface of Sub MB Type 92

4.8.2. Flow Chart of Sub MB Type in P_slice 92

4.8.3. Flow Chart of Sub MB Type in B_slice 93

4.9. Intra Prediction Mode for Luma4x4 95

4.9.1. Interface of Intra Prediction Mode for Luma4x4 95

4.9.2. Flow Chart of Intra Prediction Mode for Luma4x4 96

4.10. Intra Prediction Mode for Chroma 96

4.10.1. Interface of Intra Prediction Mode for Chroma 96

4.10.2. Flow Chart of Intra Prediction Mode for Chroma 97

4.11. Reference Frame Index 98

4.11.1. Interface of Reference Frame Index 98

4.11.2. Flow Chart of Reference Frame Index 99

4.12. Motion Vector Difference 99

4.12.1. Interface of Motion Vector Difference 100

4.12.2. Flow chart of Motion Vector Difference 101

4.13. Coded Block Pattern 101

4.13.1. Interface of Coded Block Pattern 101

4.13.2. Flow Chart of Coded Block Pattern 102

4.14. MB Based Quantization Parameter 103

4.14.1. Interface of MB Based Quantization Parameter 103

4.14.2. Flow Chart of MB Based Quantization Parameter 103

4.15. Coded Block Flag 104

4.15.1. Interface of Coded Block Flag 104

4.15.2. Flow Chart of Coded Block Flag 105

4.16. Significance Map 106

4.16.1. Interface of Significance Map 106

4.16.2. Flow Chart of Significance Map 107

4.17. Level Information 108

4.17.1. Interface of Level Information 108

4.17.2. Flow Chart of Level Information 109

4.18. A CABAD Decoding Example 110

Chapter 5. Experimental Results 115

5.1. Synthesis Results of the Proposed Design 115

5.2. Performance of the Proposed Design 116

5.2.1. Performance of the CAVLD Logic 116

5.2.2. Performance of the CABAD Logic 119

5.2.3. Comparisons with Previous Work 122
5.2.3.1. Comparison of CAVLD Logic with the Design in [8] 122
5.2.3.2. Performance of CABAD Logic 123
5.2.4. Performance Analysis on Target Platform 124
Chapter 6. Conclusions and Future Work 126
References 127

List of Figures

Fig 1. Bit-rate savings provided by CABAC relative to CAVLC of h.264/AVC (from [11])........2

Fig 2. ML-506 development board........3

Fig 3. Target platform system architecture........4

Fig 4. Architecture of the bus wrapper module........12

Fig 5. Select input logic in Fig 4........13

Fig 6 Select output logic in Fig 4........13

Fig 7. Overall architecture of the CAVLD top module........15

Fig 8. Interface of CAVLD controller........18

Fig 9. Finite state machine of CAVLD........20

Fig 10. Architecture of Shift_count and carry signal........20

Fig 11. Architecture of forward signal........21

Fig 12. Architecture of Input_data signal........21

Fig 13. interface of first one detector........22

Fig 14. Architecture of first one detector........23

Fig 15. Architecture of Priority_1 ~ Priority_5........24

Fig 16. Interface of switch_input_1 module........25

Fig 17. Architecture of sil_ok signal........25

Fig 18. Architecture of switch_output_1 signal........26

Fig 19. interface of switch_input_2........27

Fig 20. Architecture of switch_output_2 signal........27

Fig 21. Interface of Coeff_token Decoder........29

Fig 22. vlcnum logic in Fig 21........29

Fig 23 output logic1 in Fig 21........30

Fig 24. output logic2 in Fig 21........31

Fig 25. Interface of Trailing_ones Calculator........32

Fig 26. Architecture of Output_trailing_ones........32

Fig 27. Architecture of Trailing_ones_ok and Trailing_ones_total_ok...33

Fig 28. Interface of Level Decoder........35

Fig 29. level prefix logic in Fig 28........35

Fig 30. num of bits logic in Fig 28........36

Fig 31. level suffix logic in Fig 28........36

Fig 32. output level logic in Fig 28........37

Fig 33. suffix length logic in Fig 28........38

Fig 34. level ok logic in Fig 28 39

Fig 35. Interface of Total_zeros Decoder 41

Fig 36. max coeff logic in Fig 35 41

Fig 37. output logic1 in Fig 35 42

Fig 38. output logic2 in Fig 35 42

Fig 39. Interface of Run_before Decoder 44

Fig 40. zero left logic in Fig 39 45

Fig 41. run before logic in Fig 39 45

Fig 42. total ok logic in Fig 39 46

Fig 43. zero left out logic in Fig 39 46

Fig 44. example for Coeff_token decoder 48

Fig 45. example for Trailing_ones decoder 48

Fig 46. example for Level decoder (1/2) 49

Fig 47. example for Level decoder (2/2) 49

Fig 48. example for Total_zeros decoder 50

Fig 49. example for Run_before decoder (1/4) 50

Fig 50. example for Run_before decoder (2/4) 51

Fig 51. example for Run_before decoder (3/4) 51

Fig 52. example for Run_before decoder (4/4) 51

Fig 53. Architecture of the bus wrapper module 55

Fig 54. Overall architecture of the CABAD top module 58

Fig 55. Interface of normal decoding process 61

Fig 56. Flow chart of normal decoding process 62

Fig 57. decide MPS or LPS logic in Fig 55 62

Fig 58. state logic in Fig 55 63

Fig 59. range logic in Fig 55 63

Fig 60. dbitsleft logic in Fig 55 64

Fig 61. MPS logic in Fig 55 65

Fig 62. Bin logic in Fig 55 65

Fig 63. value logic in Fig 55 66

Fig 64. Next logic in Fig 55 67

Fig 65. Interface of bypass decoding process 68

Fig 66. Flow chart of bypass decoding process 68

Fig 67. dbitsleft logic in Fig 65 69

Fig 68. next logic in Fig 65 69

Fig 69. value logic in Fig 65 70

Fig 70. Bin logic in Fig 65 71

Fig 71. Interface of final decoding process 72

Fig 72. Flow chart of the final decoding process 72
Fig 73. Bin logic in Fig 71 73
Fig 74. range logic in Fig 71 73
Fig 75. Next logic in Fig 71 74
Fig 76. value logic in Fig 71 75
Fig 77. dbitsleft logic in Fig 71 75
Fig 78. Interface of initialization for context variables 77
Fig 79. ok logic in Fig 78 77
Fig 80. MPS state logic in Fig 78 78
Fig 81. ram logic in Fig 78 79
Fig 82. State controllers for each syntax element controller 79
Fig 83. Interface of MB Skip Flag 84
Fig 84. Flow chart of MB Skip Flag 84
Fig 85. Interface of MB type 85
Fig 86. Flow chart of MB Type in I_slice 86
Fig 87. Flow chart of MB Type in P_slice 88
Fig 88. Flow chart part1 of MB Type in B_slice 89
Fig 89. Flow chart part2 of MB Type in B_slice 90
Fig 90. Interface of Sub MB Type 92
Fig 91. Flow chart of Sub MB Type in P_slice 93
Fig 92. Flow chart of Sub MB Type in B_slice 94
Fig 93. Interface of Intra Prediction Mode for Luma4x4 96
Fig 94. Flow chart of Intra Prediction Mode for Luma4x4 96
Fig 95. Interface of Intra Prediction Mode for Chroma 97
Fig 96. Flow chart of Intra Prediction Mode for Chroma 98
Fig 97. Interface of Reference Frame Index 99
Fig 98. Flow chart of Reference Frame Index 99
Fig 99. Interface of Motion Vector Difference 100
Fig 100. Flow chart of Motion Vector Difference 101
Fig 101. Interface of Coded Block Pattern 102
Fig 102. Flow chart of Coded Block Pattern 103
Fig 103. Interface of MB Based Quantization Parameter 103
Fig 104. Flow chart of MB Based Quantization Parameter 104
Fig 105. Interface of Coded Block Flag 105
Fig 106. Flow chart of Coded Block Flag 106
Fig 107. Interface of Significant Map 107
Fig 108. Flow chart of Significant Map 108
Fig 109. the interface of Level Information 109

Fig 110. Flow chart of Level Information 110

Fig 111. AC_Regular_mode of arithmetic decoding engine for one
bin 112

Fig 112. Example for AC_Regular_mode 113

Fig 113. Example for AC_Bypass_mode 113

Fig 114. Example for AC_Final_mode 113

List of Tables

Table 1. Table in Fig 22 30
Table 2. thresholds for determining whether to increment suffix_Length 39
Table 3. bitstream example for Intra frame.....................................47
Table 4. bitstream example for Inter frame.....................................47
Table 5. Binarization for MB Type in I_slice..................................87
Table 6. Binarization for MB Type in P_slice.................................88
Table 7. Binarization for MB Type in B_slice 91
Table 8. Binarization for Sub MB Type in P_slice...........................93
Table 9. Binarization for Sub MB Type in B_slice 95
Table 10. bitstream example for Intra frame................................... 110
Table 11. bitstream example for Inter frame................................... 111
Table 12. Synthesis settings 115
Table 13. Macro statistics of CAVLD and CABAD logics 116
Table 14. Slice logic utilization of CAVLD and CABAD logics...... 116
Table 15. Encoding parameters of each test sequence...................... 117
Table 16. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of the 4x4 blocks in I_SLICE of each sequence 117
Table 17. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of the 4x4 blocks in P_SLICE of each sequence 117
Table 18. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of the 2x2 blocks in I_SLICE of each sequence 118
Table 19. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of 2x2 blocks in P_SLICE of each sequence 118
Table 20. The number of decoding cycles per each block type of each sequence 118
Table 21. Totals bits and Mbits/sec of each slice type of each sequence 119
Table 22. Encoding parameters of each test sequence...................... 119
Table 23. Total number of decoding cycles of each slice type of each

sequence 119
Table 24. Cycle/MB of each slice type of each sequence................. 120
Table 25. The average number of occurrences of each syntax element in I_SLICEs... 120
Table 26. The average number of occurrences of each syntax element in P_SLICEs.. 121
Table 27. The average number of occurrences of each syntax element in B_SLICEs 122
Table 28. Totals bits and Mbits/sec of each slice type in each sequence 122
Table 29. Comparisons of average cycle / MB with design in [8] ... 122
Table 30. Comparisons of maximum working frequency with design in [8] 123
Table 31 Comparisons of average cycle / MB with design in [18]. . 123
Table 32. Comparisons of maximal working frequency with design in [18] 123
Table 33. Performance of the CAVLD logic on the target platform.. 124
Table 34. Comparisons of average cycles/MB on experimental development board and using estimation of CABAD logic 125

Chapter 1. Introduction

1.1. Motivation

Entropy coding is an important part of today's video and image coding applications. It is often applied together with other lossy image compression techniques to increase the data compression rate. One of the popular entropy coding techniques is the Variable Length Coding (VLC). The main idea of variable length coding is to minimize the average codeword length. Shorter codewords are assigned to frequently occurring data and longer codewords are assigned to infrequently occurring data. In early MPEG video coding standards like MPEG-1, MPEG-2 and MPEG-4, the VLC technique has been used to reduce the amount of video data streams according to a fixed statistical model. In order to further increase the data compression ratio, AVC (a.k.a. H.264) has adopted the Context-based Adaptive Variable Length Coding (CAVLC) technique to encode the residual data organized as 4x4 or 2x2 blocks of transform coefficients. Due to the context-adaptive feature of the H.264 CAVLC, its coding efficiency is higher than that of the traditional MPEG entropy coders at the cost of slightly higher complexity.

Another entropy coding technique adopted in H.264 is the Context-based Adaptive Binary Arithmetic Coding (CABAC) for main profile video. Wiegand et al. [11] compare the coding efficiency of CABAC with the coding efficiency of CAVLC of H.264/AVC. As shown in Fig 1, the bit-rate savings of CABAC relative to CAVLC of H.264/AVC are shown against the average PSNR of the luminance component for the five interlaced sequences of the test set. It can be seen that CABAC outperforms CAVLC of H.264/AVC for the typical area of target applications. For the range of acceptable video quality for broadcast application of about 38–42 dB average bit-rate savings around 8% are achieved, where higher gains are obtained at lower rates.

Name	Resolution	Frame rate	Duration
Canoe	720 × 576	25 Hz	6 sec.
Formula 1	720 × 576	25 Hz	6 sec.
Rugby	720 × 576	25 Hz	6 sec.
Mobile & Calendar	720 × 480	30 Hz	6 sec.
Football	720 × 480	30 Hz	6 sec.

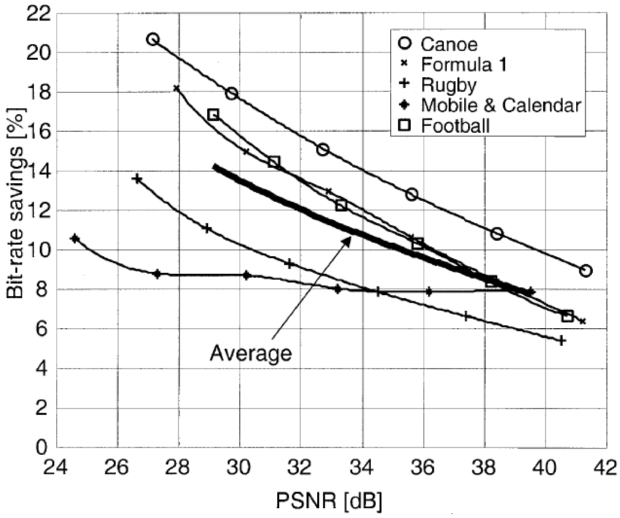


Fig 1. Bit-rate savings provided by CABAC relative to CAVLC of h.264/AVC (from [11])

Although most of the H.264 main profile stream is encoded using CABAC, its decoding algorithm is basically sequential and needs large computation to calculate range, offset and context variables, making it difficult to achieve high decoding performance. The CABAC decoding complexity required to process high definition images in real time is about 3 giga-operations per second. Although this computing complexity is still less than the block processing complexity, the CABAC decoding becomes a major bottleneck in real time processing of high bitrate data due to its sequential nature. In summary, in order to support real-time H.264 video applications, it is necessary to design dedicated hardware for CAVLC and CABAC decoding.

1.2. Target Platform

In this thesis, we used an FPGA development board – ML506 to verify our design. The ML-506 development board is shown in Fig 2, and the key features of the ML-506 are list as follows:

- Virtex 5 XC5VSX50TFFG1136.
- 64 Mbyte flash prom (32Mb x 2).
- 64-bit wide, 256-MB DDR2 Memory.
- JTAG Programming Interface.
- 10/100 Ethernet PHY
- RS-232 - serial port
- USB-2.0 PHY
- Audio in, Audio out.
- Video input, Video (DVI/VGA) output

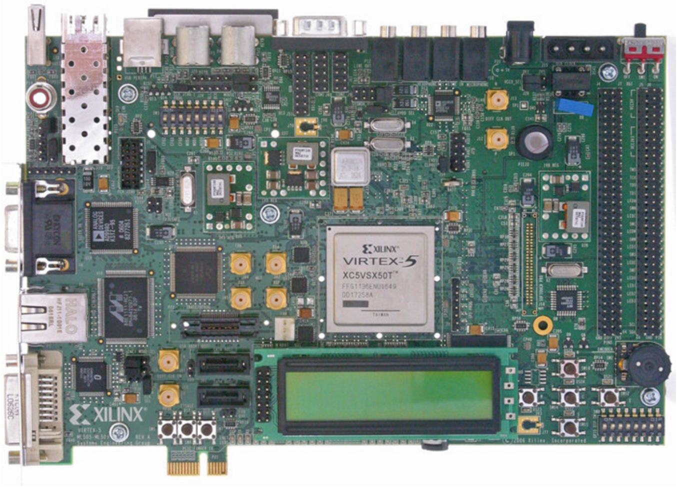


Fig 2. ML-506 development board.

ML-506 has enough logic gate capacity to emulate a complete SoC reference design with a complex 32-bit processor core and some IP's as shown in Fig 3. The central processing unit (CPU) of the target platform is a Sun SPARC v8 compliant soft-core processor, LEON3. The reference hardware platform shown in Fig 3 is constructed using the GRLIB IP library version 1.0.18 developed by Gaisler Research.

The proposed design of the CAVLD and CABAD logics will be implemented on the target platform and a full hardware-software system verification based on the AVC/H.264 reference software implementation JM 12.2 [26] will be conducted to verify the correctness of the proposed entropy decoding hardware logics. Both the CAVLD and CABAD logics will communicate with the rest of the system components (CPU, memory, etc.) via the AMBA AHB bus protocol.

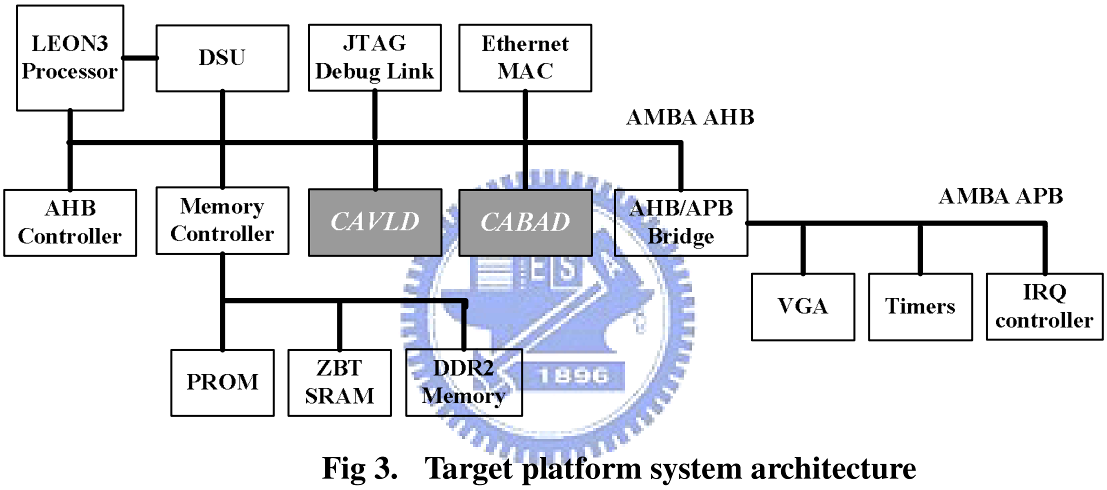


Fig 3. Target platform system architecture

1.3. Outline of the Thesis

The organization of the rest of the thesis is as follows. Chapter 2 introduces some previous work related to CAVLD and CABAD logic design. Chapter 3 describes the algorithm, architecture, and implementation details of the proposed CAVLD logic. Chapter 4 describes the algorithm, architecture, and implementation details of the CABAD logic. Chapter 5 shows the experimental results, and finally, the conclusion and discussions will be given in Chapter 6.

Chapter 2. Previous Work

In this Chapter, we will review some previous designs of the CAVLD and the CABAD architecture.

2.1. CAVLD

Chang et al. [4] published an efficient CAVLD architecture and proposed four different techniques to reduce both the hardware cost and power consumption of CAVLD. These techniques are Partial Combinational Component Freezing (PCCF), Hierarchical Logic for Look-up Tables (HLLT), Zero_left Table Elimination by Arithmetic (ZTEBA), and Zero Codeword Skip (ZCS). As compared to the architecture proposed in [3], the design achieves 23% reduction in hardware cost and 40% improvement in speed.

PCCF is a technique to assign one enable signal to each CAVLC decoder component to freeze the non-operating component of combinational circuits to achieve low-power consumption. HLLT is used to partition the original big LUT into many small LUTs so that the unused parts of the LUTs can be disabled for reducing power consumption. ZTEBA is a technique for more efficient decoding of Run-before syntax elements by finding out the rules among the LUTs. ZCS is used to skip decoding of zero codewords when all coefficients in 4x4 or or 2x2 blocks are zeros.

Lin et al. [7] suggested a power-efficient approach called prefix pre-decoding that can reduce power consumption by 25%. Based on empirical analysis, the lengths of codewords in Coeff_token hardly exceed 8. Therefore, LUTs are divided into two groups: one group contains LUTs with codewords smaller than 8 and the other group contains codewords that can exceed 8.

Yu et al. [8] proposed several techniques to improves the performance of CAVLD, including merging the Coeff_token and T1s processes together to reduce cycles, skipping some decoding process if no coefficient is necessary to be decoded, and decoding multiple symbols in Run_before stage. The proposed design uses 90 cycles for one MB decoding on average, which can meet real time HDTV requirement and saves 64% of cycle count on average when compared with the design in [4].

Tseng et al. [9] proposed an algorithm with a redesign of the LUTs. If a pattern is matched in their look-up table, they can skip the standard CAVLD procedure and reconstruct a block directly. The performance can be improved by 10% compared with the standard CAVLD procedure. In short, the most frequently occurring 4x4-block (or 2x2-block) bitstream patterns are recorded in a table so that a full block decoding can be done quickly. They sample 4,000 frequent patterns and arrange them according to their frequencies. Sum of frequencies of top 4,000 patterns occupies 67.63% of number of decoded block, They re-arrange the order of these 4,000 patterns according to their bit lengths, and there are 81.07% of patterns represented within 8 bits and 96.93% of patterns represented within 12 bits. The pattern-search algorithm is based on a two-pass table look-up method and all the coefficients in a 4x4 (or 2x2) block can be reconstructed directly.

Kim et al. [10] proposed a new CAVLC decoding method using arithmetic hashing operations instead of the conventional table look-up method. Experimental results show that the proposed algorithm is 50% faster and uses 95% less memory access comparing with three conventional search-based table lookup CAVLC algorithms such as Moon's method [5].

2.2. CABAD

Yu et al. [17] divided syntax elements into two classes according to their occurring frequency during the decoding process. For more frequent syntax elements, a new architecture that can decode two regular bins together with one bypass bin in one cycle is proposed. These syntax elements include Motion Vector Difference, Significance Map, and Level Information. And they also divide context models into 18 groups according to their access frequency. With this mechanism, access frequency to the RAM storing context models is greatly reduced. For a typical 4Mbps bitstream at D1 resolution, experimental results show that on average each MB can be decoded within 500 cycles.

Yang et al. [18] proposed several techniques to optimize the CABAC decoding process. At MB information level, previously decoded MB information is packed so that accesses to this information for current MB decoding are more efficient. At slice-data and MB layer level, they perform careful pipeline scheduling, using segmented context tables, adding cache registers, and doing look-ahead codeword parsing to improve decoding performance. In summary, it takes three cycles to generate a bin. When an internal loop occurs, there might be a succession of context memory accesses using the same context values. Therefore, cache registers are used to store the context values and write back to context memory only once at the end of the internal loop. In addition, they propose a look-ahead codeword parsing scheme to detect if the re-normalization on the probability model occurs in CABAD. If the look-ahead condition fits, they can decode two bins in each cycle. Otherwise, it takes one cycle to decode one-bit of CABAD codeword. Furthermore, they partition one context table into multiple segmented context memories. Thus, combining the segmented context memories with cache registers, they can read and write memory in a more flexible way. By exploiting all the proposed design techniques, they can averagely reduce about 53% of cycles count in the CABAD decoding process.

Similarly, Zheng et al. [22], also point out that not all the parameters of a reference MB or all the possible values of an SE need to be stored. They have shown that the bits that are required to store a reference MB are only 142 bits. The proposed architecture and control/decoding strategy have improved the time efficiency by 27.9%, 18.2%, and 48.8% for I frames, P frames, and B frames compared with the architecture in Chen et al. [16].

Chapter 3. Design of the CAVLD Logic

As mentioned in section 1.2, the target platform is a LEON3-based FPGA development board (shown in Fig 3.). In this chapter, we will present our design of the CAVLD logic and how it is integrated into the LEON platform. First, in section 3.1, we present the CAVLD algorithm. In section 3.2, we present the overall architecture of the CAVLD logic and its interface to the LEON processor and the memory subsystem. In section 3.3, we present the state controller of CAVLD. In section 3.4, we discuss some front-end logics, including the first one detector and the switch input module, used to preprocess the input bitstream and extract bit-patterns for the CAVLD logic. Then, we will present the main decoding components of CAVLD which converts the bit patterns to syntax elements, including the Coeff_token decoder (section 3.5), the Trailing_ones calculator (section 3.6), the Level decoder (section 3.7), the Total_zeros decoder (section **3.8**), and the Run_before decoder (section **3.9**). In section 3.10, we will give a CAVLD decoding example.

3.1. Introduction to the CAVLD Entropy Decoder

3.1.1. Algorithm of CAVLD

The algorithm of CAVLD is described as follows:

Step 1. Decoding the number of coefficients and trailing ones:

In this step, CAVLD decodes both the total number of nonzero coefficients (Total_coeff) and the number of trailing ones (trailing ±1 values). Total_coeff can be anything from 0 to 16 for 4x4 blocks, 0 to 4 for 2x2 blocks and Trailing_ones can be anything from 0 to 3. If there are more than three trailing ones, only the last three are counted as trailing ones and others are treated as normal coefficients. When Total_coeff is equal to 0, only this step is required.

Step 2. Decoding the sign of each Trailing_ones coefficient:

Number of trailing ones is calculated in step1. The sign of each trailing one is decoded using one bit from the bitstream. Bit 0 stands for +1 and bit 1 stands for –1. This step is done in reverse order, so that the highest frequency trailing one comes first. If the number of trailing ones is equal to 0, we will skip this step.

Step 3. Decoding the levels of remaining non-zero coefficients:

The level (sign and magnitude) for each remaining non-zero coefficient in the block is decoded in reverse order, starting with the highest frequency. Each level value is coded in Golomb code and can be represented as 0...01xx...xs. The "0...01" is the prefix, and "xx...xs" is the suffix.

Step 4. Decoding the total number of zeros before the last coefficient:

This step is used to decode the total number of zeros before the last non-zero coefficient with the information of Total_coeff at 4x4 or 2x2 blocks. We partition the tables by Total_coeff and looked up these small variable-length tables indexed by the bitstream we will decode. If Total_coeff is equal to 0 or Max_coeff, we will skip this step.

Step 5. Decoding each run of zeros:

The number of zero preceding the highest non-zero coefficient (run_before) is decoded in reverse order. A run_before parameter is decoded for each non-zero coefficient, starting with the highest frequency, with two exceptions:

I. If there are no more zeros left to decode, (i.e. $\sum$[run_before] = Total_zeros), it is not necessary to decode any more run_before values.

II. It is not necessary to decode run_before for the final (lowest frequency) non-zero coefficient.

There are seven sub tables for run_before decoding. These tables are designed based on the Zero_left value, which is calculated by subtracting previous run from previous Zero_left and initialized with Total_zeros. When Zero_left is equal to 0, the remaining runs of rest coefficients is set to 0.

After the introduction of RTL model of CAVLD logic, we will give a example in section 3.10.

3.2. Overall System Architecture

3.2.1. AMBA AHB BUS Wrapper for CAVLD

In this section, we first discuss the bus wrapper module of the CAVLD logic for AMBA AHB bus protocol. In order to integrate the proposed CAVLD logic into the AMBA system bus of the target platform, we designed a bus wrapper for AHB 2.0 protocol. If, in the future, we have to port the CAVLD/CABAD logic to other bus protocols, we only need to modify the bus wrapper.

The interface and the architecture of the bus wrapper module are shown in Fig 4. Most of the interface signals are defined in the AMBA 2.0 bus protocol specification. The interface signals that are not part of the standard AMBA signals are explained as follows:

- hcachei and hcacheo are 1-bit input signals indicate that the data on the data bus is cacheable or non-cacheable.
- hirq is a 32-bit input signal indicates interrupt result bus.
- testen is a 1-bit input signal indicates scanning test enable.
- testrst is a 1-bit input signal indicates scanning test reset.
- scanen is a 1-bit input signal indicates scanning enable.
- testoen is a 1-bit input signal indicates testing output enable.
- hconfig is a set of registers indicates memory access register.

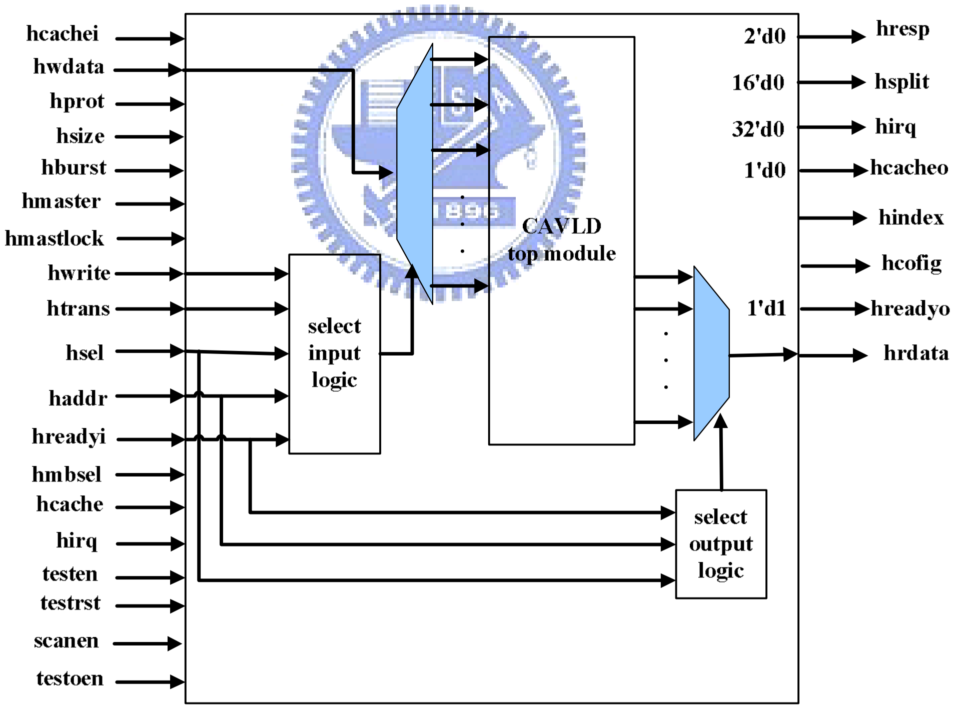


Fig 4. Architecture of the bus wrapper module

We use the select_signal to select the input signals of CAVLD module connecting the hwdata as shown in Fig 5. We use select_signal to select the output signals of CAVLD module connecting to hrdata as shown in Fig 1.

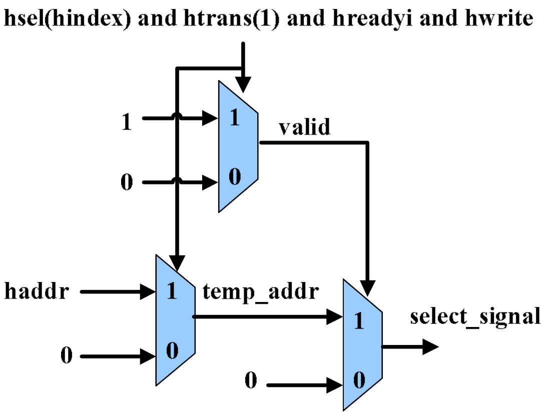


Fig 5. Select input logic in Fig 4

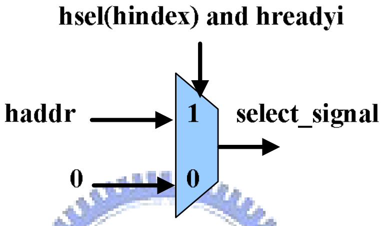


Fig 6. Select output logic in Fig 4

3.2.2. CAVLD Top Module

CAVLD top module includes all of the CAVLD components. The interface and the architecture of the CAVLD top module are shown in Fig 7, and the interface signals are explained as follows.

- nA and nB are each a 5-bit input signal indicates the number of non-zero coefficients in the left and top block of current block.
- nA_valid and nB_valid are each 1-bit input signal indicate whether upper or left block is available or not.
- Decode_start is a 1-bit input signal is used to enable the CAVLD module.
- Decode_type is a 3-bit input signal indicates the type of the current block.
- Input_stream is a 32-bit input signal indicates the bitstream we will decode.
- Update_stream_ok is a 2-bit input signal indicates whether the Input_data

signal is updated or not.

- Output_ok is a 1-bit input signal indicates whether the level and run for the current block are finished outputting or not.
- Output_enable is a 1-bit output signal indicates whether the level and run for the current block can begin outputting or not.
- forward is a 2-bit output signal indicates whether we needed to update Input_data signal or not.
- temp_level_0 ~ temp_level_15 and temp_run_0 ~ temp_run_15 are each 16-bit output signal indicates the value of level and run of the current block.
- Shift_count is a 6-bit output signal indicates the number of bits is used in Input_data signal and which is used to get the unused bitstream of the Input_data signal.
- We use a 53-bit common_input signal to stand for {clk, rst, nA, nB, nA_valid, nB_valid, Decode_start, Decode_type, Input_stream, Update_stream_ok, Output_ok}.
- We use a 521-bit common_output signal to stand for {Shift_count, Output_enable, forward, temp_level_0 ~ temp_level_15, temp_run_0 ~ temp_run_15}.

As shown in Fig 7, all decoding modules communicate with the State Controller to get data and control signals we need, and all decoding modules got input bitstream from the Switch_input_1 or the First_one_detector modules.

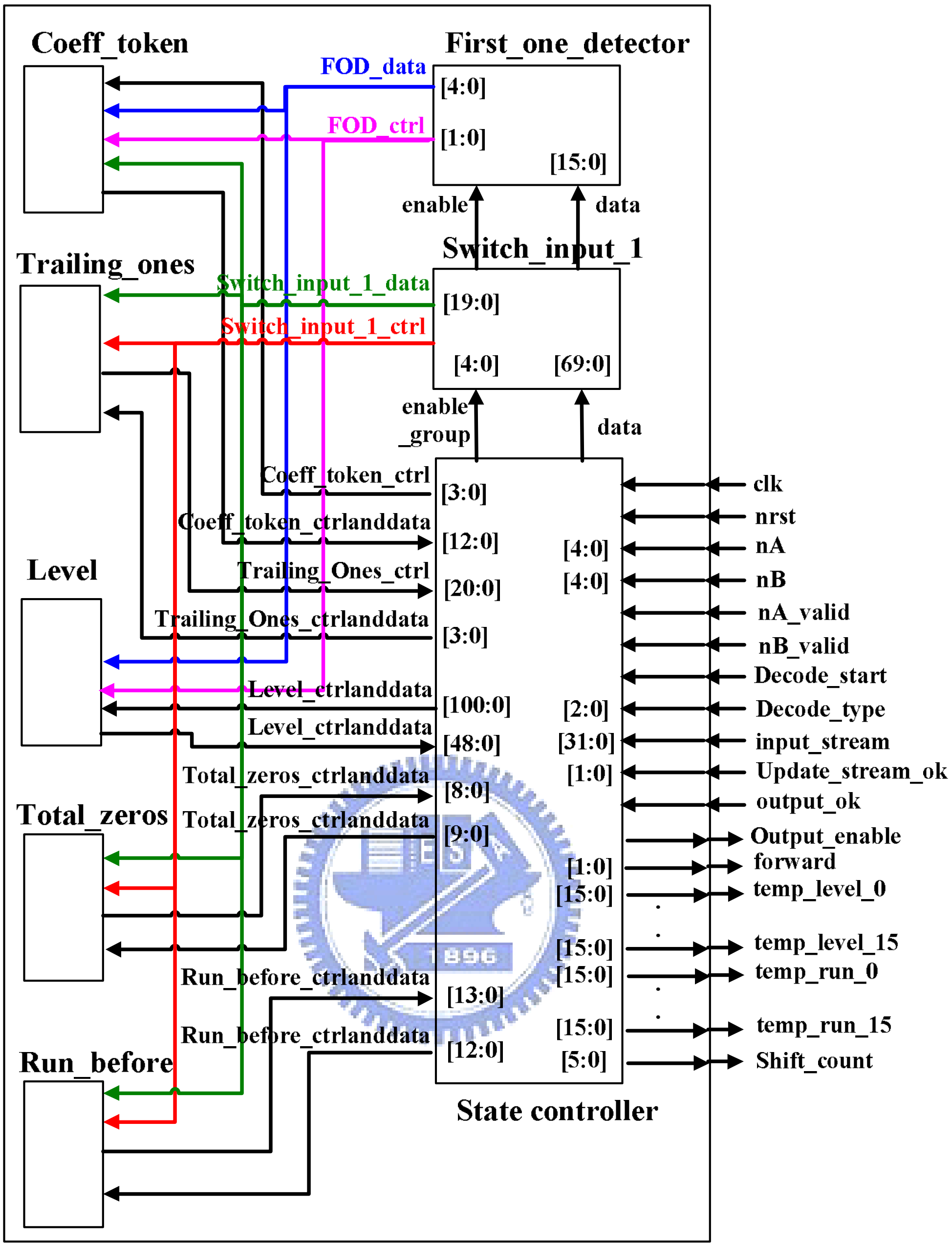


Fig 7. Overall architecture of the CAVLD top module

3.3. State Controller

State controller module includes a state transition controller and a barrel shifter.

3.3.1. Interface of State Controller

The interface of the state controller is shown in Fig 8. The interface signals of the controller are explained as follows:

- common_input and common_output are 53-bit and 515-bit signals which are described in 3.2.2.
- ok_group is a 5-bit input signal which stands for {Coeff_token_ok, Trailing_ones_ok, Level_ok, Total_zeros_ok, Run_before_ok}.
- total_ok_group is a 3-bit input signal which stands for {Trailing_ones_total_ok, Level_total_ok, Run_before_total_ok}.
- Num_of_bits_group is a 20 bit input signal which stands for {Num_of_bits_Coeff_token, Num_of_bits_Trailing_ones, Num_of_bits_Level, Num_of_bits_Total_zeros, Num_of_bits_Run_before}.
- Total_coeff and Trailing_ones are each 5-bit and 2-bit input signal which connects the Total_coeff and Trailing_ones output signal of Coeff_token Decoder.
- Trailing_ones_count_out and Output_Trailing_ones are each 2-bit and 16-bit input signal which connects the Trailing_ones_count_out and Output_Trailing_ones output signal of Trailing_ones Calculator.
- Output_Level, Prefix_greater_than_15, Suffix_Length_out, Level_count_out and Level_Prefix_for_Prefix_greater_than_15_out are each 16-bit, 1-bit, 3-bit, 5-bit and 15-bit input signal which connects the Output_Level, Prefix_greater_than_15, Suffix_Length_out, Level_count_out and Level_Prefix_for_Prefix_greater_than_15_out signal of Level Decoder.
- Output_Total_zeros is a 4-bit put signal which connects the Output_Total_zeros output signal of Total_zeros Decoder.
- Zero_left_out and Run_before are each 4-bit and 4-bit output signal which connects Zero_left_out and Run_before output signal of Run_before

Decoder.

- Input_data is a 64-bit output signal indicates the bitstream we will decode.
- vlcnum is a 3-bit output signal which is used to select the look-up tables for decoding coeff_token for a 4x4 block or a 2x2 block.
- enable_group is a 5-bit output signal which stands for {Coeff_token_enable, Trailing_ones_enable, Level_enable, Total_zeros_enable, Run_before_enable} indicates which decoding module is working on this time.
- Trailing_ones_count_in and Temp_Trailing_ones are each 2-bir and 2-bit output signal which connects the Trailing_ones_count_in and Temp_Trailing_ones input signal of Trailing_ones Calculator.
- Temp_Total_Coeff, Suffix_Length_in, Level_Prefix_in and Level_count_in are each 5-bit, 3-bit, 16-bit and 5-bit output signal which connects the Total_Coeff, Suffix_Length_in, Level_Prefix_in and Level_count_in input signal of Level Decoder.
- Max_coeff is a 5-bit output signal which connects the Max_coeff input signal of Total_zeros Decoder.
- Temp_Total_zeros and Zero_left_in are each 4-bit and 4-bit output signal which connects the Temp_Total_zeros and Zero_left_in input signal of Run_before Decoder.

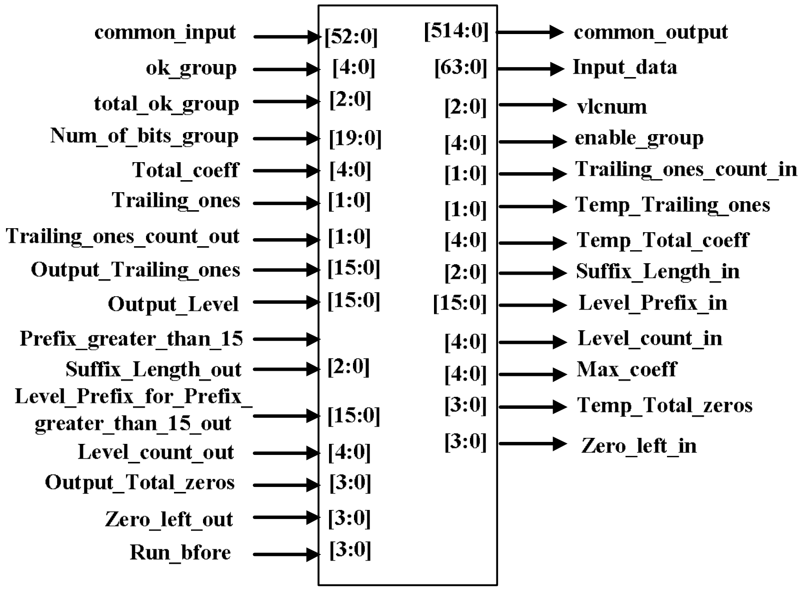


Fig 8. Interface of CAVLD controller

3.3.2. States of the Controller

The controller is a finite state machine as shown in Fig 9. We need 7 states to decode bitstream in CAVLD module. For decoding a 4x4 or 2x2 blocks, we will go from state1 to state7. In Fig 9, 'A' through 'P' are state transition conditions which are described as follows:

- Condition A indicates Coeff_token_ok is equal to 0. On the other words, if coeff_token module is not finished, we will be still in state1.
- Condition B indicates Total_coeff is equal to 0. On the other words, we will skip the current block, and we will transfer to state6.
- Condition C indicates Coeff_token_ok is equal to 1. On the other words, if coeff_token module is finished, we will transfer to state2.
- Condition D indicates Trailing_ones_total_ok is equal to 0. On the other words, if trailing_ones module is not finished, we will be still in state2.
- Condition E indicates Trailing_ones is equal to 0. On the other words, we will skip the trailing_ones module, and we will transfer to state3.

- Condition F indicates Trailing_ones_total_ok is equal to 1. On the other words, if trailing_ones module is finished, we will transfer to state3.
- Condition G indicates Level_total_ok is equal to 0. On the other words, if level module is not finished, we will be still in state3.
- Condition H indicates Total_coeff is equal to Trailing_ones. On the other words, we will skip the level module, and we will transfer to state4.
- Condition I indicates Level_total_ok is equal to 1. On the other words, if level module is finished, we will transfer to state4.
- Condition J indicates Total_zeros_ok is equal to 0. On the other words, if total_zeros module is not finished, we will be still in state4.
- Condition K indicates Total_zeros_ok is equal to 1. On the other words, if total_zeros module is finished, we will transfer to state5.
- Condition L indicates Run_before_ok is equal to 0. On the other words, if run_before module is not finished, we will be still in state5.
- Condition M indicates Total_zeros is equal to 0. On the other words, we will skip the run_before module, and we will transfer to state6.
- Condition N indicates Run_before_ok is equal to 1. On the other words, if run_before module is finished, we will transfer to state6.
- Condition O indicates we started output the level and run signal of the current block.
- Condition P indicates we finished output the level and run signal of the current block and start to decode a new block.

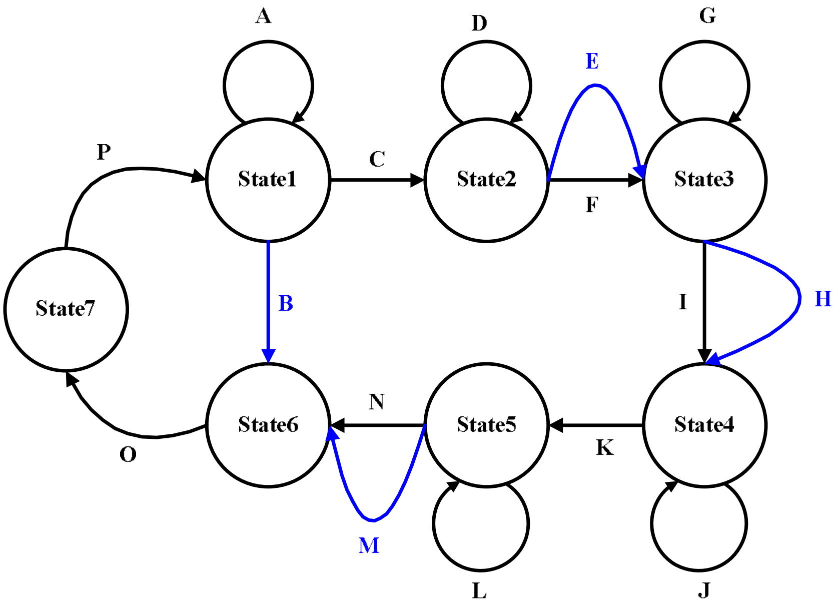


Fig 9. Finite state machine of CAVLD

3.3.3. Barrel Shifter

The Shift_count and carry signal are processed by the logic shown in Fig 10. The carry signal is used to decide the forward signal.

ES 1896

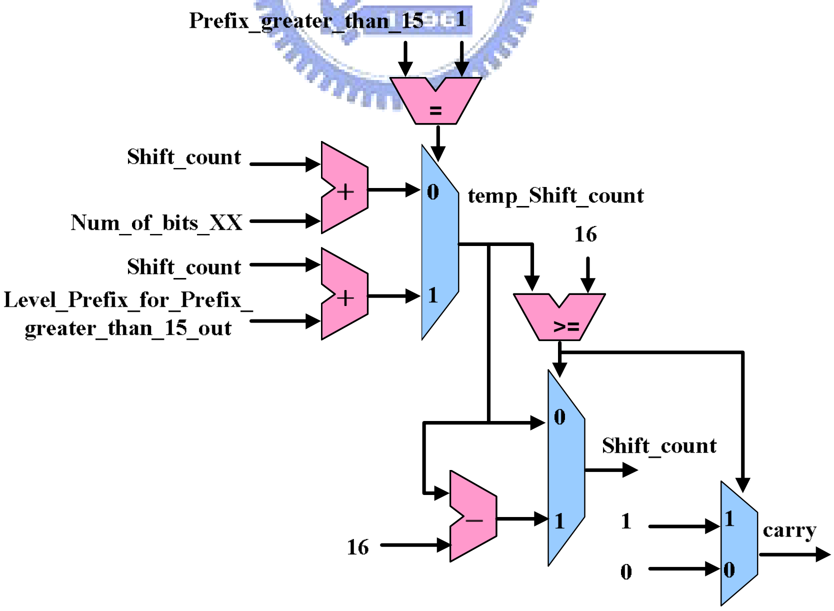


Fig 10. Architecture of Shift_count and carry signal

If Prefix_greather_than_15 is equal to 0, Shift_count is set to Shift_count add Num_of_bits_XX, XX can be Coeff_token, Trailing_ones, Level, Total_zeros or

Run_before. Otherwise, Shift_count is set to (Shift_count + Level_Prefix_for_Prefix_greater_than_15_out). And then if Shift_count is greater than or equal to 16, Shift_count is set to (Shift_count – 16) and carry is set to 1.

The forward signal is generated by the logic shown in Fig 11.

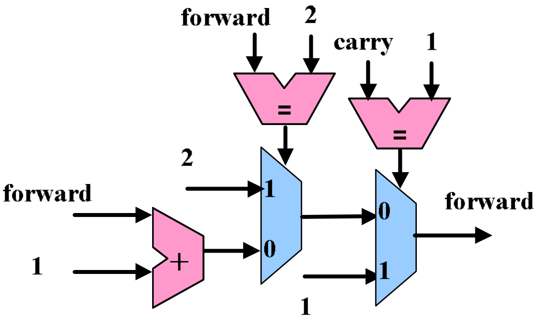


Fig 11. Architecture of forward signal

If carry is equal to 1, forward is set to 1. Otherwise, forward is set to (forward+1). But the maximum value of forward is 2.

The Input_data signal is generated by the logic shown in Fig 12.

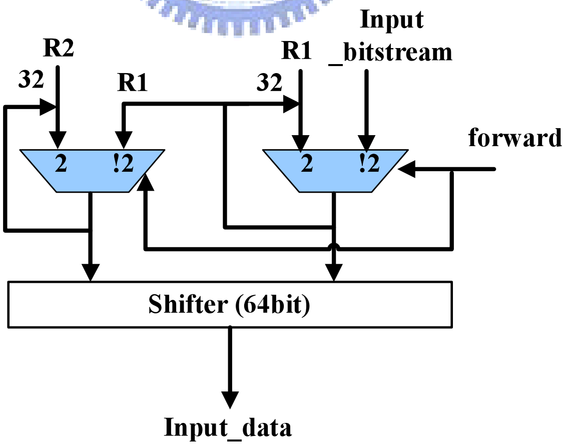


Fig 12. Architecture of Input_data signal

We use a barrel shifter to generate the Input_data signal. Input_data is set to {R2, R1}, where R1 and R2 are 32-bit registers. If forward is not equal to 2, we will update the Input_data signal with 32-bit Input_bitstream signal.

3.4. Bitstream Preprocessing Logics

In this section, we discusses the design of the bitstream parsing front-end logics used to extract the bit patterns from the bitstream to make it easier for the CAVLD logic to convert the bit patterns to syntax elements.

3.4.1. First One Detector

This module computes the number of leading zeros in the input bitstream.

3.4.1.1. Interface of First One Detector

The interface of the first one detector is shown in Fig 13, and the interface signals are explained as follows.

- enable is a 3-bit input signal indicates which module needs leading zeros of input bitstream.
- input_data is a 16-bit input signal indicates the input bitstream.
- Leading_zeros is a 5-bit output signal indicates the number of leading zeros of input_data.
- fod_ok is a 2-bit output signal indicates whether this module is finished or not. If the Coeff_token module used this module, the value of fod_ok is set to 1, else if the level module used this module, the value of fod_ok is set to 2.

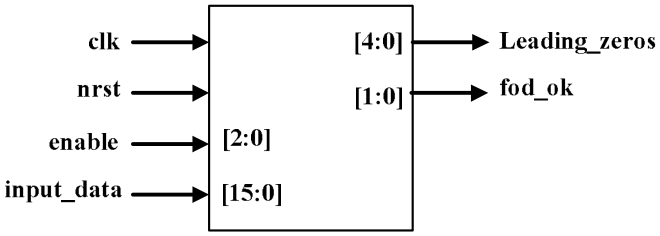


Fig 13. interface of first one detector

3.4.1.2. Architecture of First One Detector

The architecture of the first one detector is shown in Fig 14. Priority_1, Priority_2, Priority_3, Priority_4 encode first ‘1’ position in each four bits of input_data signal. Four non-equal gates test whether there is a ‘1’ among the four bits of input_data signal or not. Priority_5 encodes the first ‘1’ position in the output of four non-equal gates. The output of Priority_1 ~ Priority_5 are the numbers of leading zeros of the each four bits. And then we use the output of Priority_5 to select the output of Priority_1 ~ Priority_4 called temp_output, and we combined the output of Priority_5 and temp_output[1:0] as the number of leading zeros of input_data.

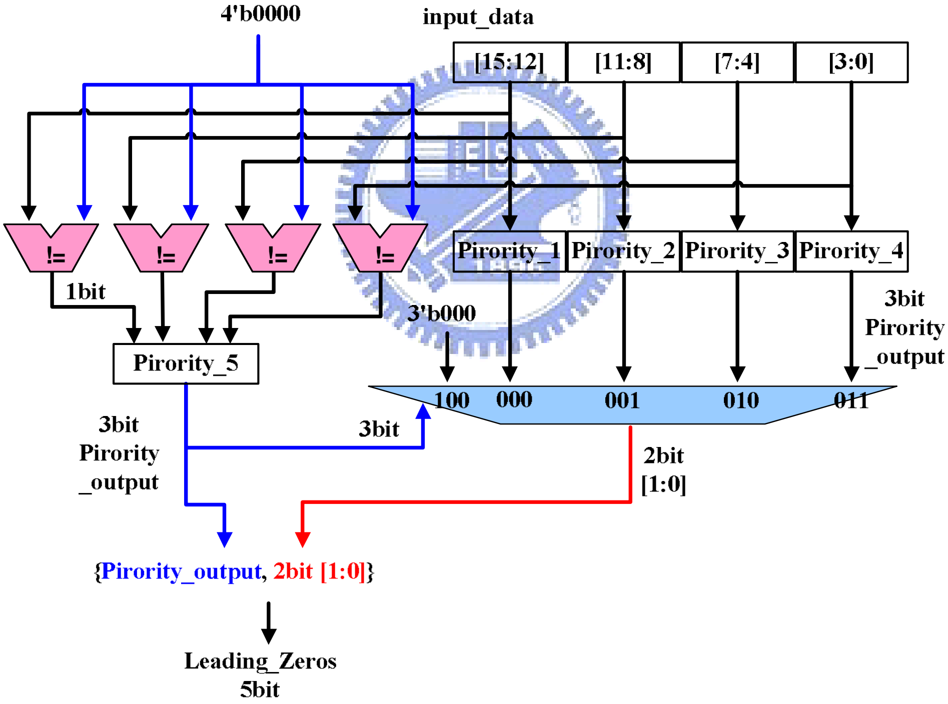


Fig 14. Architecture of first one detector

The architecture of Priority_1 ~ Priority_5 is shown in Fig 15.

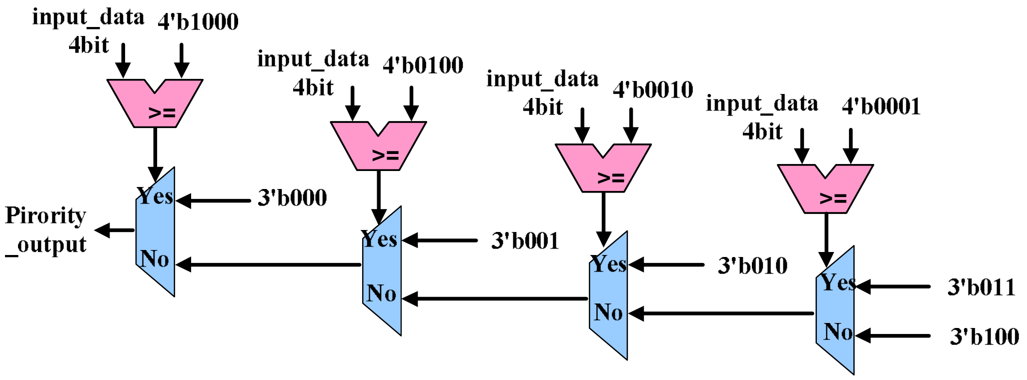


Fig 15. Architecture of Priority_1 ~ Priority_5

The output of Priority_1 ~ Priority_5 are the numbers of leading zeros of the each four bits of input_data.

3.4.2. Switch Input Module

We divide the switch input module into two cases: switch_input_1 and switch_input_2. The switch_input_1 module is used to get the unused bitstream of the input_data, ex. If the input_data is 0011000101…, the 0011 of the input_data is used, so the output of the switch_input_1 module is 000101…. And the switch_input_2 module is used to get the unused bitstream not including leading zeros of the output of the switch_input_1 module, ex. If the output of the switch_input_1 module is 000101…, so the output of the switch_input_2 module is 101….

3.4.2.1. Switch_input_1 Module

3.4.2.1.1. Interface of Switch_input_1 Module

The interface of the switch_input_1 module is shown in Fig 16, and the interface signals are explained as follows.

- input_data is a 64-bit input signal indicates the bitstream we will decode.
- Shift_count is a 6-bit input signal indicates the number of bits is used in input_data signal and which is used to get the unused bitstream of the

input_data signal as the switch_output_1 signal.

- Coeff_token_enable, Trailing_ones_enable, Level_enable, Total_zeros_enable, and Run_before_enable are each 1-bit input signal indicates which decoding module is working on this time.
- switch_output_1 is a 20-bit output signal indicates the first 20-bit unused bitstream of the input_data signal.
- sil_ok is a 3-bit output signal indicates whether this module is finished or not, and this signal is used to enable Coeff_token, Trailing_ones, Level, Total_zeros or Run_before module.

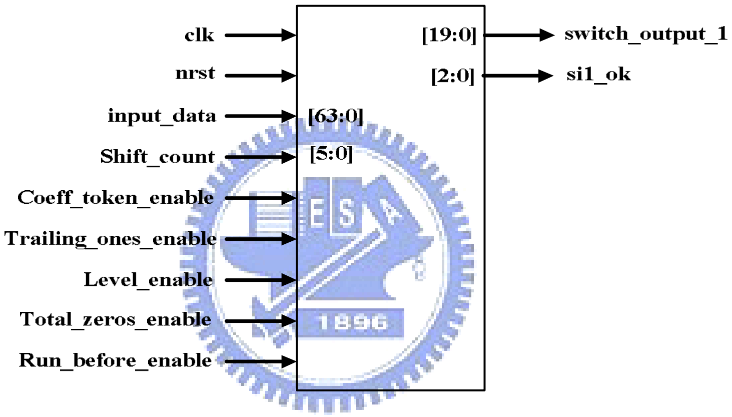


Fig 16. Interface of switch_input_1 module

3.4.2.1.2. Architecture of switch_input_1 Module

The sil_ok signal is generated by the logic shown in Fig 17.

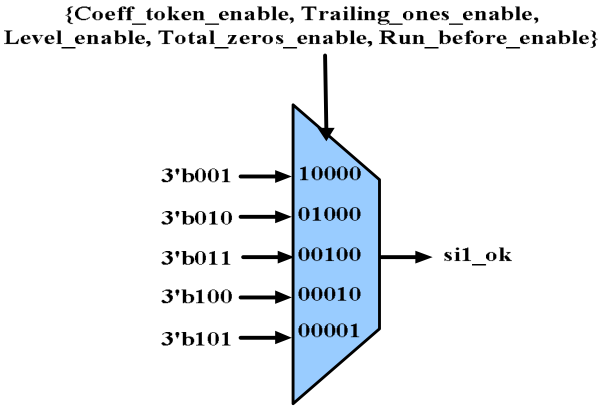

{Coeff_token_enable, Trailing_ones_enable,
Level_enable, Total_zeros_enable, Run_before_enable}
3'b001
3'b010
3'b011
3'b100
3'b101
10000
01000
00100
00010
00001
sil_ok

Fig 17. Architecture of sil_ok signal

The value of the si1_ok signal is depending on {Coeff_token_enable, Trailing_ones_enable, Level_enable, Total_zeros_enable, Run_before_enable}.

The switch_output_1 signal is generated by the logic shown in Fig 18.

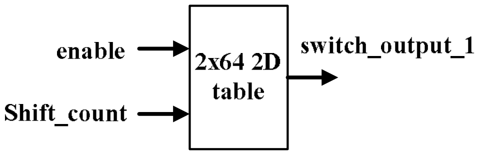


Fig 18. Architecture of switch_output_1 signal

To get the first 20-bit unused bitstream of the input_data signal, we looked up a fixed 2x64 2-D table indexed of enable and Shift_count.

3.4.2.2. Switch_input_2 Module

3.4.2.2.1. Interface of switch_input_2 Module

The interface of the switch_input_2 module is shown in Fig 19, and the interface signals are explained as follows.

- enable is a 2-bit input signal which connects the fod_ok output signal of first one detector module. Only Coeff_token module uses this module.
- switch_output_1 is a 20-bit input signal which connects the switch_output_1 output signal of switch_output_1 module.
- Leading_zeros is a 5-bit input signal which connects the Leading_zeros output signal of first one detector module.
- switch_output_2 is a 4-bit output signal indicates the first 4-bit unused bitstream not including leading zeros of the switch_input_1 signal.
- ok is a 1-bit output signal indicates whether this module is finished or not.

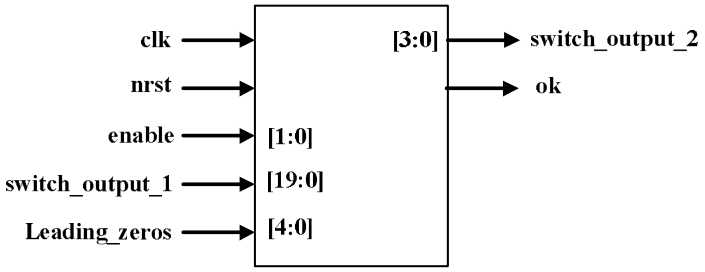


Fig 19. interface of switch_input_2

3.4.2.2.2. Architecture of switch_input_2 Module

The switch_output_2 signal is generated by the logic shown in Fig 20.

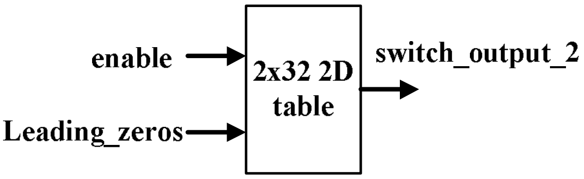


Fig 20. Architecture of switch_output_2 signal

To get the first 4-bit unused bitstream not including leading zeros of the switch_input_1 signal, we looked up a fixed 2x32 2-D table indexed of enable and Leading_zeros.

3.5. Coeff_token Decoder

This module decodes both the total number of nonzero coefficients (Total_coeff) and the number of trailing ±1 values (Trailing_ones). Total_coeff can be anything from 0 to 16 for 4x4 blocks, 0 to 4 for 2x2 blocks and Trailing_ones can be anything from 0 to 3. If there are more than three trailing ±1s, only the last three are treated as trailing ones and others are treated as normal coefficients. When Total_coeff is equal to 0, only this module is required.

3.5.1. Interface of Coeff_token Decoder

The interface of Coeff_token Decoder is shown in Fig 21, and the interface signals are explained as follows:

- vlcnum is a 3-bit input signal which is used to select the look-up tables for decoding coeff_token for a 4x4 block or a 2x2 block.
- Coeff_token_enable is a 1-bit input signal to enable coeff_token module.
- switch_output_1 is a 20-bit input signal which connects the switch_output_1 output signal of switch_output_1 module.
- Leading_zeros is a 5-bit input signal which connects the Leading_zeros output signal of first one detector module.
- fod_ok is a 2-bit input signal which connects the fod_ok output signal of first one detector module.
- Coeff_token_ok is a 1-bit output signal indicates whether this module is finished or not.
- Total_Coeff is a 5-bit output signal indicates the total number of nonzero coefficients.
- Trailing_ones is a 2-bit output signal indicates the number of trailing ±1 values.
- Num_of_bits_coeff_token is a 5-bit output signal indicates the number of bits be consumed of input bitstream in this module.

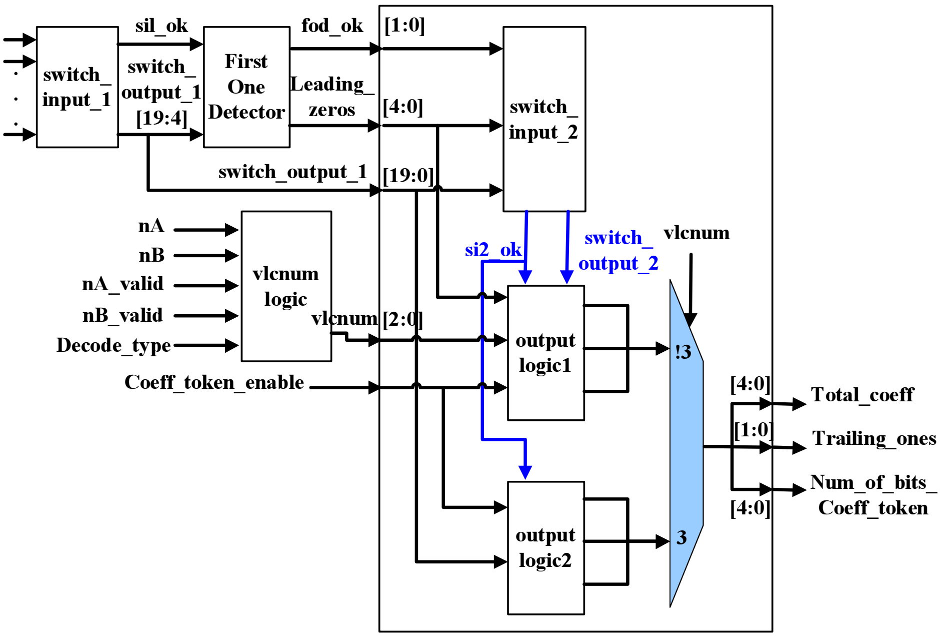


Fig 21. Interface of Coeff_token Decoder

3.5.2. Architecture of Coeff_token Decoder

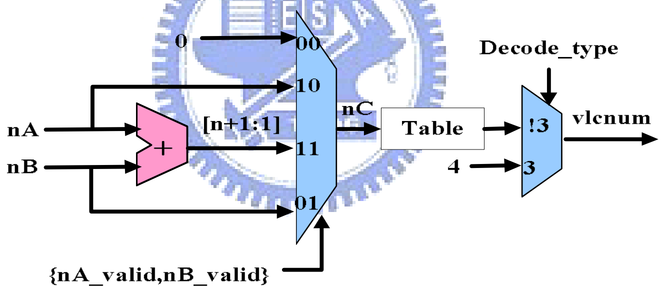


Fig 22. vlcnum logic in Fig 21

The vlcnum signal is generated by the logic shown in Fig 22.

There are four choices of look-up tables for decoding the coeff_token of a 4x4 block, three variable-length code tables (vlcnum = 0, 1, 2) and a fixed-length code table (vlcnum = 3), The choice of table depends on the number of non-zero coefficients in the left and top previously decoded blocks, nA and nB. There is a variable-length code table (vlcnum = 4) to use for decoding coeff_token for a 2x2 block. If upper and left blocks nB and nA are both available, i.e. in the same slice, nC = round ((nA+nB)/2). If only the upper is available, nC = nB; if only the left block is available, nC = nA; if neither is available, nC = 0. And then we looked up tables indexed by nC to decide the vlcnum output signal.

nC	vlcnum
0,1	0
2,3	1
4,5,6,7	2
8 or above	3

Table 1. Table in Fig 22

If there are small than eight non-zero coefficients in neighboring blocks, the Total_coeff, Trailing_ones and Num_of_bits_Coeff_token signal are generated by the logic shown in Fig 23.

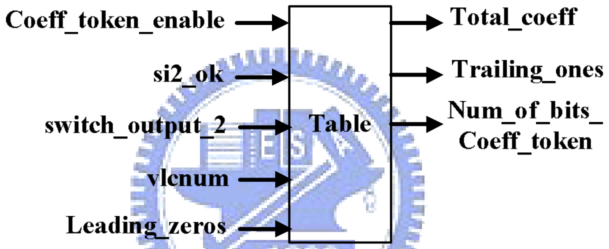


Fig 23. output logic1 in Fig 21

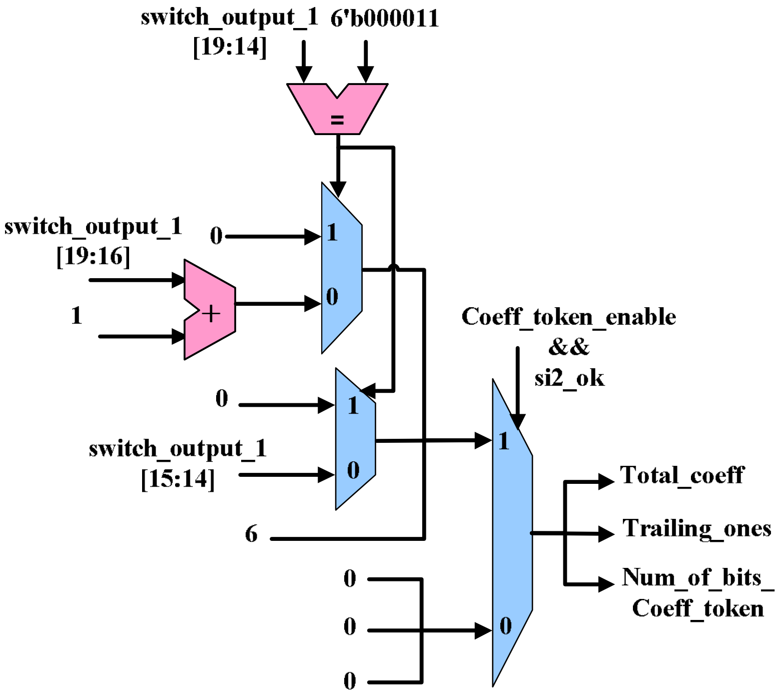


Fig 24. output logic2 in Fig 21

If there are more than eight non-zero coefficients in neighboring blocks, we used a fixed six bit coding. Be different with output logic1, output logic2 used a simple combinational logic to replace looking up a fixed-length table. The Total_coeff, Trailing_ones and Num_of_bits_Coeff_token signal are generated by the logic shown in Fig 24.

3.6. Trailing_ones Calculator

Number of trailing ones is calculated in the Coeff_token module. The sign of each trailing one is decoded using one bit from the bitstream. Bit 0 is assigned for +1 and bit 1 is assigned for -1. This module is starting in reverse order, so that the highest frequency trailing one comes first. If the number of trailing ±1 values is equal to 0, we will skip this process.

3.6.1. Interface of Trailing_ones Calculator

The interface of Trailing_ones Calculator is shown in Fig 25, and the interface signals are explained as follows:

- Trailing_ones_count_in is a 2-bit input signal indicates how many trailing ones are decoded before this module.
- Temp_Trailing_ones is a 2-bit input signal indicates the total number of trailing ±1 values.
- switch_output_1 is a 20-bit input signal which connects the switch_output_1 output signal of switch_output_1 module.
- si1_ok is a 3-bit input signal which connects the si1_ok output signal of switch_input_1 module to enable this module.
- Trailing_ones_count_out is a 2-bit output signal indicates how many trailing

ones are decoded after this module.

- Trailing_ones_ok is a 1-bit output signal indicates a trailing one is decoded.
- Trailing_ones_total_ok is a 1-bit output signal indicates all training ones of 4x4 block or 2x2 block are decoded.
- Output_Trailing_ones is a 16-bit output signal indicates the value of trailing one.
- Num_of_bits_Trailing_ones is a 1-bit output signal indicates the number of bits be consumed of input bitstream in this module.

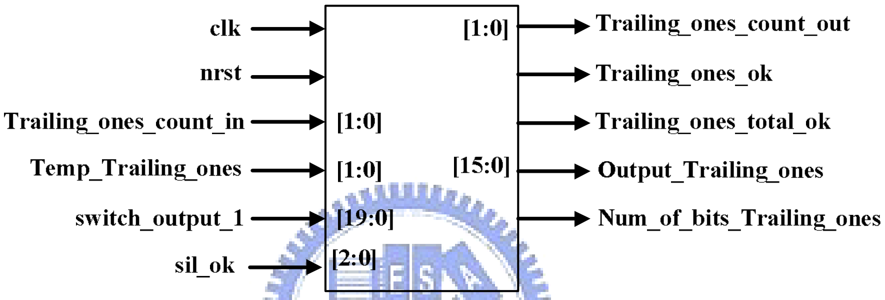


Fig 25. Interface of Trailing_ones Calculator

3.6.2. Architecture of Trailing_ones Calculator

Num_of_bits_Trailing_ones is always setting to 1.

The Output_trailing_ones signal is generated by the logic shown in Fig 26.

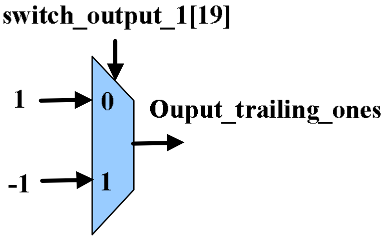


Fig 26. Architecture of Output_trailing_ones

The Trailing_ones_ok and Trailing_ones_total_ok signals are generated by the logic shown in Fig 27.

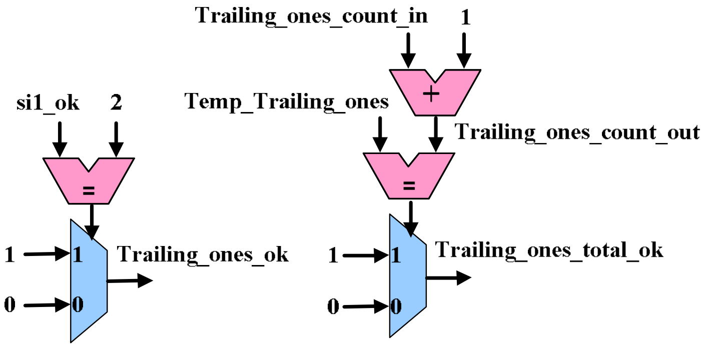


Fig 27. Architecture of Trailing_ones_ok and Trailing_ones_total_ok

If a trailing one is decoded in this time, Trailing_ones_ok is set to 1, otherwise, Trailing_ones_ok is set to 0. And if all training ones of 4x4 block or 2x2 block are decoded, on the other words, Temp_Trailing_ones is equal to Trailing_ones_count_out, Trailing_ones_total_ok is set to 1, otherwise, Trailing_ones_total_ok is set to 0.

3.7. Level Decoder

The level (sign and magnitude) for each remaining non-zero coefficient in the block is decoded in reverse order, starting with the highest frequency. Each level which is Golomb based structured can be represented as 0...01xx...xs. The “0...01” is the prefix, and “xx...xs” is the suffix.

3.7.1. Interface of Level Decoder

The interface of Level Decoder is shown in Fig 28, and the interface signals are explained as follows.

- Input_data is a 63-bit input signal indicates the input bitstream.
- Shift_count is a 6-bit input signal indicates the number of bits is used in Input_data signal and which is used to get the unused bitstream of the input_data signal.
- Total_Coeff is a 5-bit input signal indicates the total number of nonzero

coefficients.

- Trailing_ones is a 2-bit input signal indicates the number of trailing ±1 values.
- Suffix_Length_in is a 3-bit input signal indicates the length of the suffix before this module.
- Level_Prefix_in is a 16-bit input signal indicates the Leading_zeros, which is only used when the leading_zeros is greater than or equak to 15.
- Level_count_in is a 5-bit input signal indicates how many normal coefficients not including trailing ones are decoded before this module.
- Leading_zeros is a 5-bit input signal indicates the number of leading zeros of Input_data.
- fod_ok is a 2-bit input signal which connects the fod_ok output signal of first one detector module.
- Level_ok is a 1-bit output signal indicates a normal coefficient is decoded.
- Level_total_ok is a 1-bit output signal indicates all normal coefficients of 4x4 block or 2x2 block are decoded.
- Num_of_bits_Level is a 6-bit output signal indicates the number of bits be consumed of input bitstream in this module.
- Output_Level is a 16-bit output signal indicates the value of level.
- Prefix_greater_than_15 is a 1-bit output signal indicates whether the value of Leading_zeros is greater than or equal to 15 or not.
- Suffix_Length_out is a 3-bit output signal indicates the length of the suffix after this module.
- Level_Prefix_for_Prefix_greater_than_15_out is a 15-bit output signal indicates the temp Leading_zeros of this normal coefficient we will decode,
- Level_count_out is a 5-bit output signal indicates how many normal

coefficients not including trailing ones are decoded after this module.

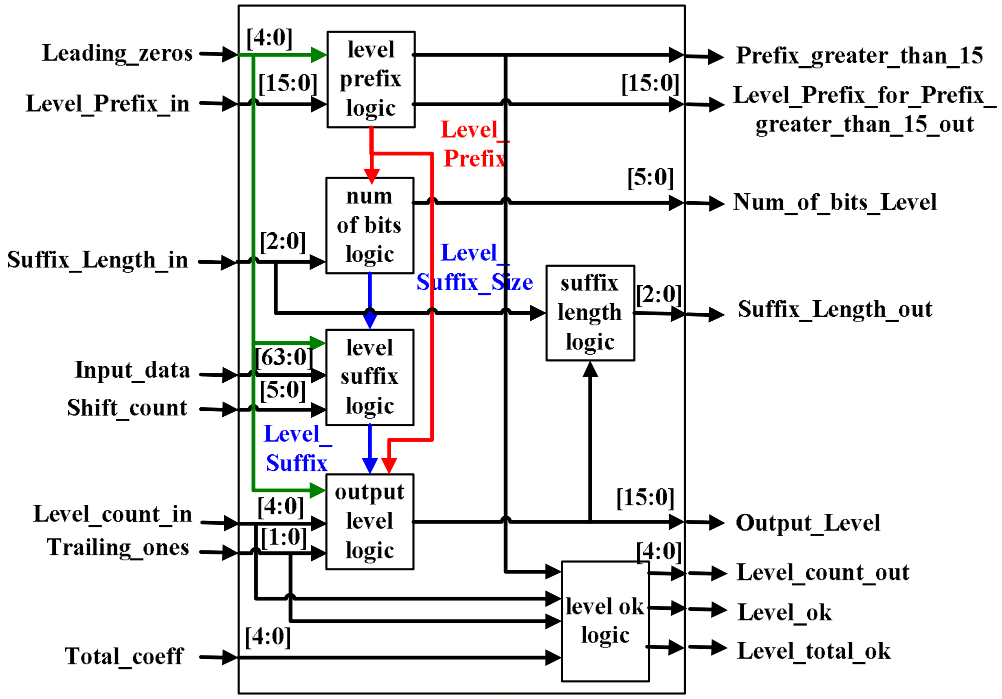


Fig 28. Interface of Level Decoder

3.7.2. Architecture of Level Decoder

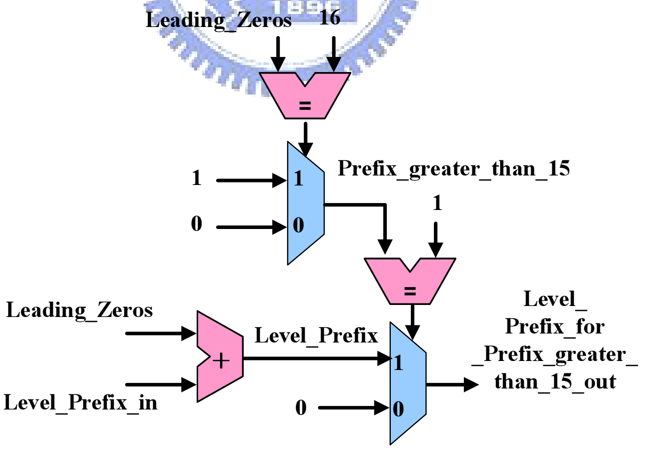


Fig 29. level prefix logic in Fig 28

The Prefix_greater_than_15 and Level_Prefix_for_Prefix_greater_than_15_out signals are generated by the logic shown in Fig 29.

If Leading_zeros is equal to 16, Prefix_greater_than_15 is set to 1 and Level_Prefix_for_Prefix_greater_than_15_out is set to (Leading_zeros +

Level_prefix_.in).

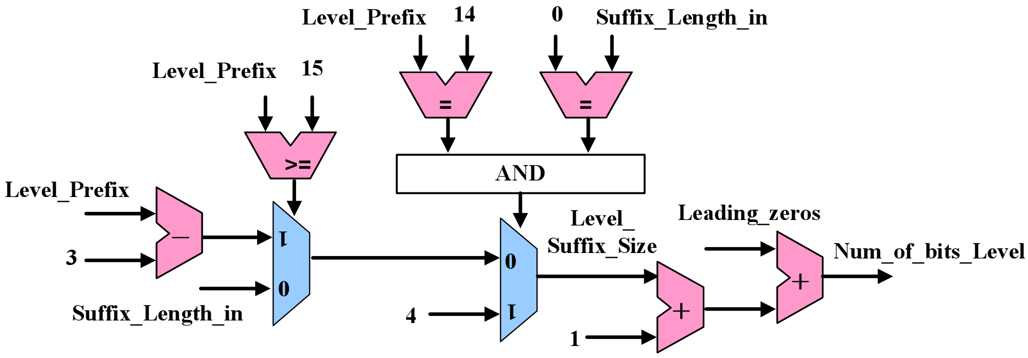


Fig 30. num of bits logic in Fig 28

The Num_of_bits_Level and Level_Suffix_Size signals are generated by the logic shown in Fig 30.

The length of the LevelSuffix, Level_Suffix_Size, is decided by the following rules:

1. If Level_Prefix is small than 15, Level_Suffix_Size is set to suffixLength;

2. If Level_Prefix is equal to 14 and suffixLength is equal to 0, Level_Suffix_Size is set to 4.

3. If Level_Prefix greater than or equal to 15, Level_Suffix_Size is set to (Level_Prefix – 3).

Num_of_bits_Level is set to (Level_Suffix_Size + Leading_zeros + 1).

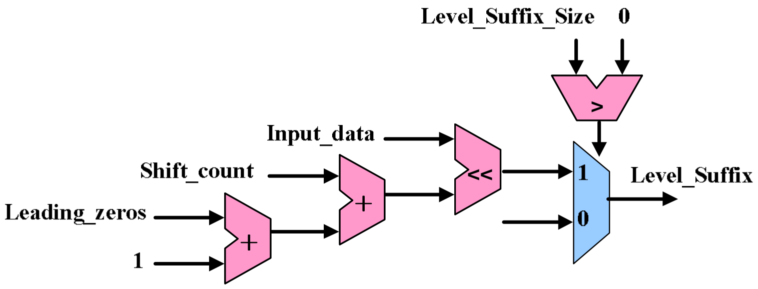


Fig 31. level suffix logic in Fig 28

The Level_Suffix signal is generated by the logic shown in Fig 31.

If Level_suffix_Size is greather than 0, Level_Suffix is set to (Input_data << (Shift_count + Leading_zeros + 1)).

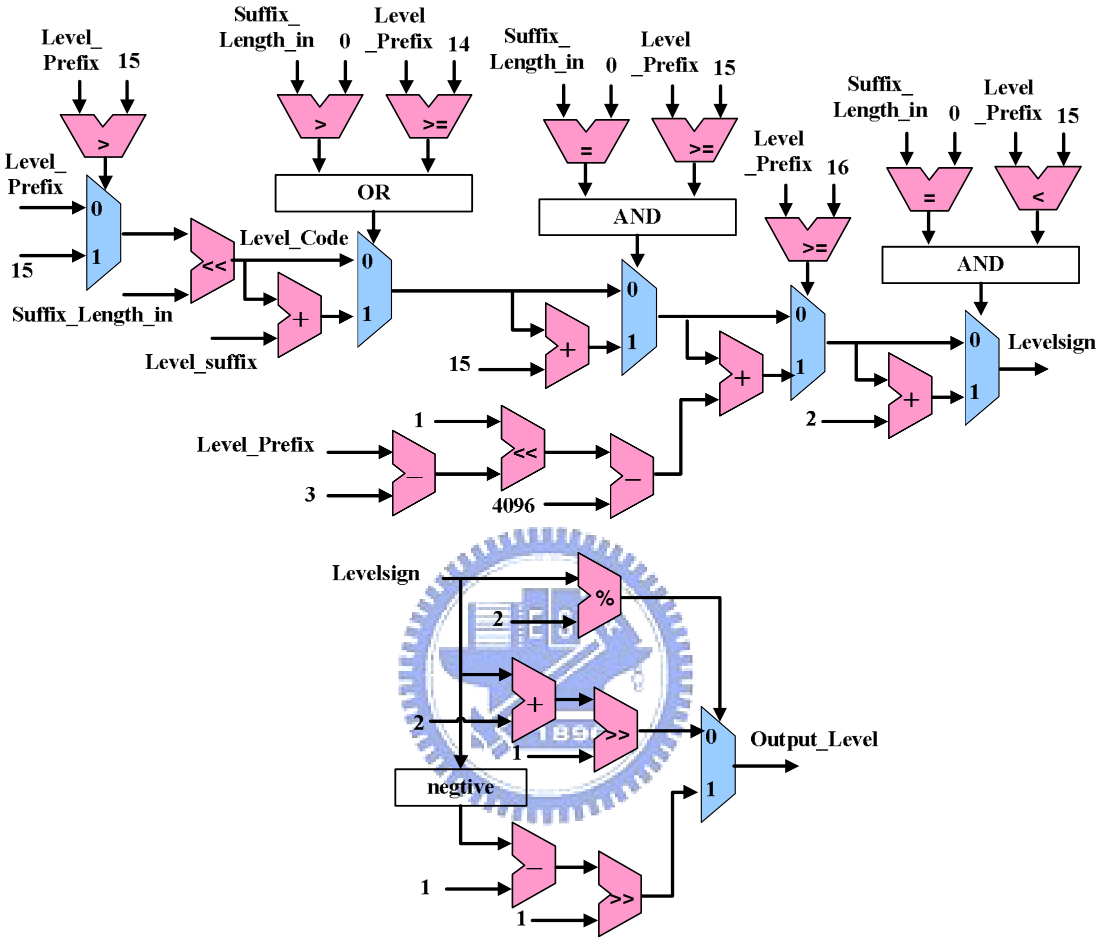


Fig 32. output level logic in Fig 28

The Output_level signal is generated by the logic shown in Fig 32.

The value of level, Level_Code, is reconstructed according to the following procedure:

1. If Level_Prefix is greater than 15, Level_Code is set to (15 << suffixLength), otherwise, Level_Code is set to (Level_Prefix << suffixLength)

2. If Suffix_Length_in is greater than 0 or Level_Prefix is greater than or equal to 14, Level_Code is set to (Level_Code + Level_suffix).

3. If Level_Prefix is greater than or equal to 15 and suffixLength is equal to 0, Level_Code is set to (Level_Code + 15).

4. If Level_Prefix is greater than or equal to 16, Level_Code is set to (Level_Code + (1 << (Level_Prefix - 3) - 4096)).

5. If Level_Prefix is small than 15 and suffixLength is equal to 0, Level_Code is set to (Level_Code + 2).

6. If Level_Code[0] is equal to 0, which means Level_code is positive, Level_Code is set to ((Level_Code + 2) >> 1), otherwise, Level_Code is set to ((-Level_Code - 1) >> 1).

7. Output_Level is set to Level_Code.

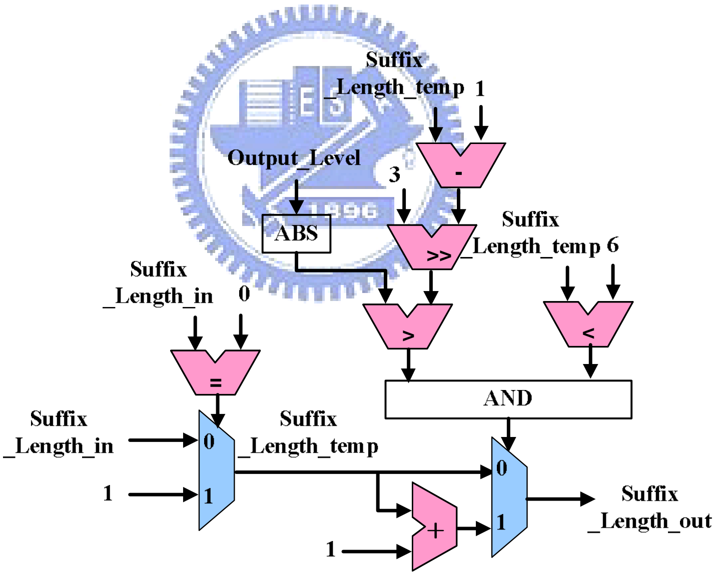


Fig 33. suffix length logic in Fig 28

The Suffix_Length_out signal is generated by the logic shown in Fig 33

The suffix_Length may be between 0 and 6 bits and suffix_Length is adapted depending on the magnitude of each successive coded level. The choice of suffix_Length is adapted as follows:

1. Initialize suffix_length to 0 (unless there are more than 10 non-zero

coefficients and less than three trailing ones, in which case initialize to 1),

2. Decode the highest-frequency non-zero coefficient.

3. If the magnitude of this coefficient is larger than a predefined threshold, increment suffix_Length. (If this is the first level to be decoded and suffix_Length was initialize to 0, set suffix_Length to 2).

In this way, the choice of suffix is matched to the magnitude of the recently-decoded coefficients. The thresholds are shown in Table 2.

Current suffix_Length	Threshold to increment suffix_Length
0	0
1	3
2	6
3	12
4	24
5	48
6	N/A (highest suffix_Length)

Table 2. thresholds for determining whether to increment suffix_Length

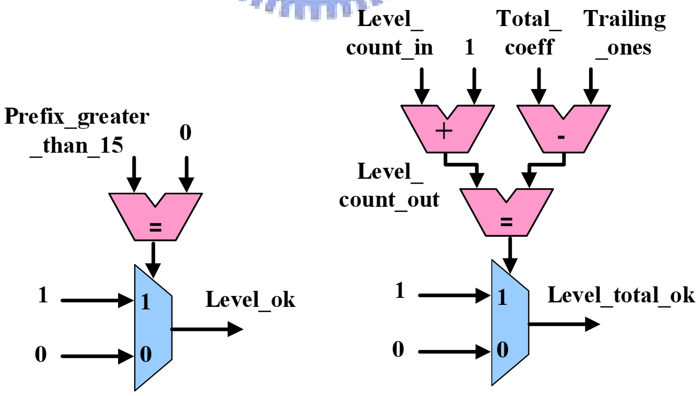


Fig 34. level ok logic in Fig 28

The level_ok and Level_total_ok signal are generated by the logic shown in Fig 34.

If Prefix_greater_than_15 is equal to 0, Level_ok is set to 1. If Level_count_out is equal to (Total_coeff - Trailing_ones), Level_total_ok is set to 1.

3.8. Total_zeros Decoder

The Total_zeros Decoder is used to decode the total number of zeros before the last non-zero coefficient with the information of Total_coeff at 4x4 or 2x2 blocks. We partition the tables by Total_coeff and looked up these small variable-length tables indexed by the bitstream we will decode. If Total_coeff is equal to 0 or Max_coeff, we will skip this process.

3.8.1. Interface of Total_zeros Decoder

The interface of the Total_zeros Decoder is shown in Fig 35, and the interface signals are explained as follows.

- Total_Coeff is a 5-bit input signal indicates the total number of nonzero coefficients.
- Max_coeff is a 5-bit input signal indicates the max number of nonzero coefficients.
- switch_output_1 is a 20-bit input signal which connects the switch_output_1 output signal of switch_output_1 module.
- si1_ok is a 3-bit input signal which connects the si1_ok output signal of switch_input_1 module to enable this module.
- Total_zeros_ok is a 1-bit output signal indicates whether this module is finished or not.
- Output_Total_zeros is a 4-bit output signal indicates the total number of zeros before the last non-zero coefficient in 4x4 or 2x2 blocks.
- Num_of_bits_Total_zeros is a 4-bit output signal indicates the number of bits be consumed of input bitstream in this module.

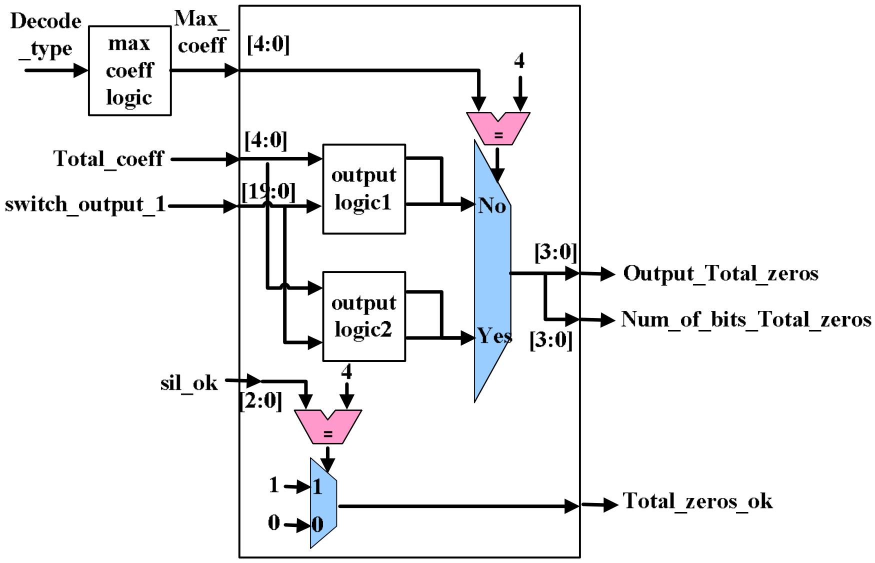


Fig 35. Interface of Total_zeros Decoder

3.8.2. Architecture of Total_zeros Decoder

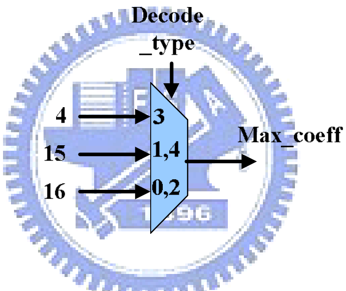


Fig 36. max coeff logic in Fig 35

The Max_coeff signal is generated by the logic shown in Fig 36.

If Decode_type is equal to ChromaDC, Max_coeff is set to 4; otherwise, if Decode_type is equal to Intra16x16AC or ChromaAC, Max_coeff is set to 15; otherwise, if Decode_type is equal to Intra16x16DC or Luma, Max_coeff is set to 16;

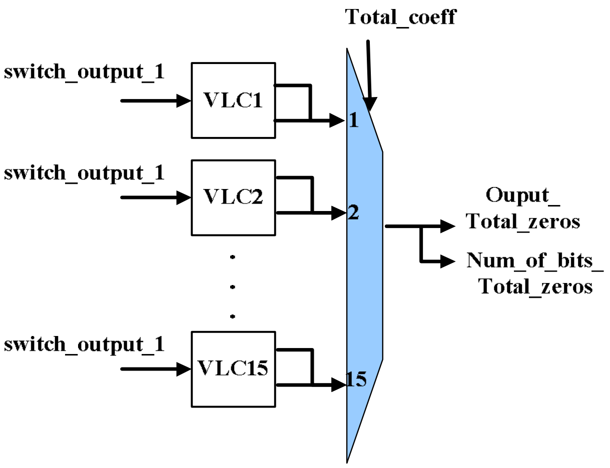


Fig 37. output logic1 in Fig 35

if Decode_type is not ChromaDC. The Output_Total_zeros and Num_of_bits_Total_zeros signals are generated by the logic shown in Fig 37.

We partition the tables to 15 tables by Total_coeff and looked up these small variable-length tables indexed by switch_output_1.

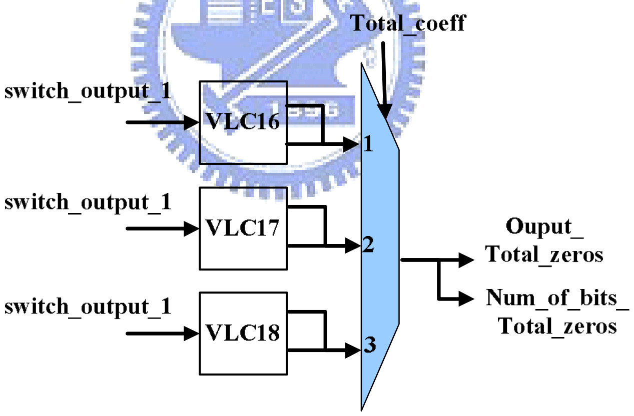


Fig 38. output logic2 in Fig 35

if Decode_type is ChromaDC. The Output_Total_zeros and Num_of_bits_Total_zeros signals are generated by the logic shown in Fig 38.

We partition the tables to 3 tables by Total_coeff and looked up these small variable-length tables indexed by switch_output_1.

3.9. Run_before Decoder

The number of zero preceding the highest non-zero coefficient (run_before) is decoded in reverse order. A run_before parameter is decoded for each non-zero coefficient, starting with the highest frequency, with two exceptions:

- If there are no more zeros left to decode, (i.e. $\sum$[run_before] = Total_zeros), it is not necessary to decode any more run_before values.
- It is not necessary to decode run_before for the final (lowest frwquency) non-zero coefficient.

There are seven sub tables for run_before decoding. These tables are divided based on the Zero_left which is calculated by subtracting previous run from previous Zero_left and initialized with Total_zeros. When Zero_left is equal to 0, the remaining runs of rest coefficients is set to 0.

3.9.1. Interface of Run_before Decoder

The interface of Run_before Decoder is shown in Fig 39, and the interface signals are explained as follows.

- Temp_Total_zeros is a 4-bit input signal which connects Output_Total_zeros output signal of the total_zeros module, and it is used to initialize the value of Zero_left.
- Zero_left_in is a 4-bit input signal indicates how many remaining zeros are not decoded before this module.
- switch_output_1 is a 20-bit input signal which connects the switch_output_1 output signal of switch_output_1 module.
- si1_ok is a 3-bit input signal which connects the si1_ok output signal of switch_input_1 module to enable this module.

- Total_coeff is a 5-bit input signal which is used to initialize the value of Tmep_Total_coeff. And Tmep_Total_coeff is used to decide whether the final (lowest frwquency) non-zero coefficient is happened or not.
- Run_before_ok is a 1-bit output signal indicates a run_before is decoded.
- Run_before_total_ok is a 1-bit output signal indicates all run_before of 4x4 block or 2x2 block are decoded.
- Zero_left_out is a 4-bit output signal indicates how many remaining zeros are not decoded after this module.
- Run_before is a 4-bit output signal indicates the number of zero preceding the current non-zero coefficient
- Num_of_bits_Run_before is a 4-bit output signal indicates the number of bits be consumed of input bitstream in this module.

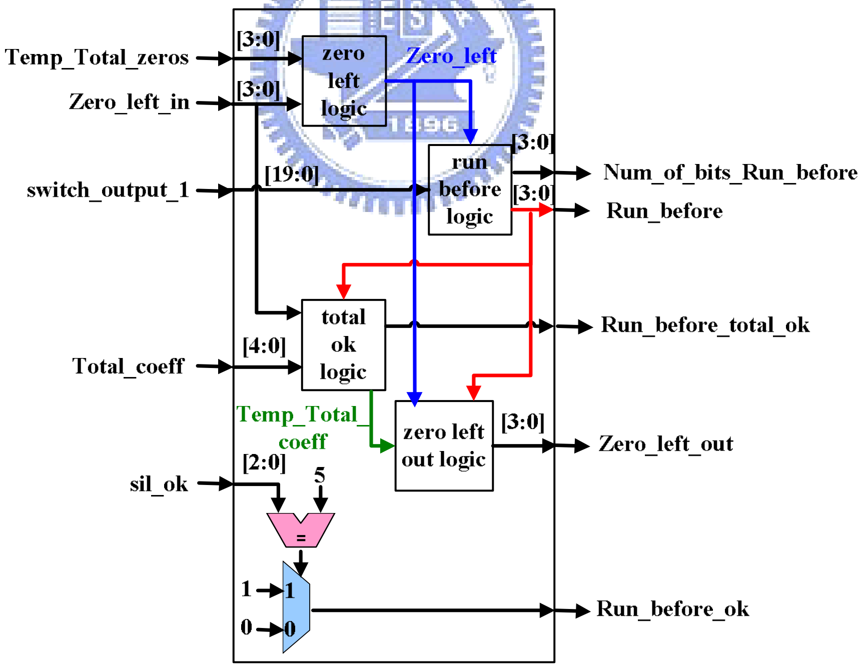


Fig 39. Interface of Run_before Decoder

3.9.2. Architecture of Run_before Decoder

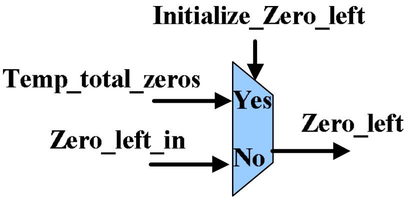


Fig 40. zero left logic in Fig 39

The Zero_left signal is generated by the logic shown in Fig 40.

The Run_before and Num_of_bits_Run_before signals are generated by the logic shown in Fig 41.

We looked up tables depend on Zero_leet and switch_input_1.

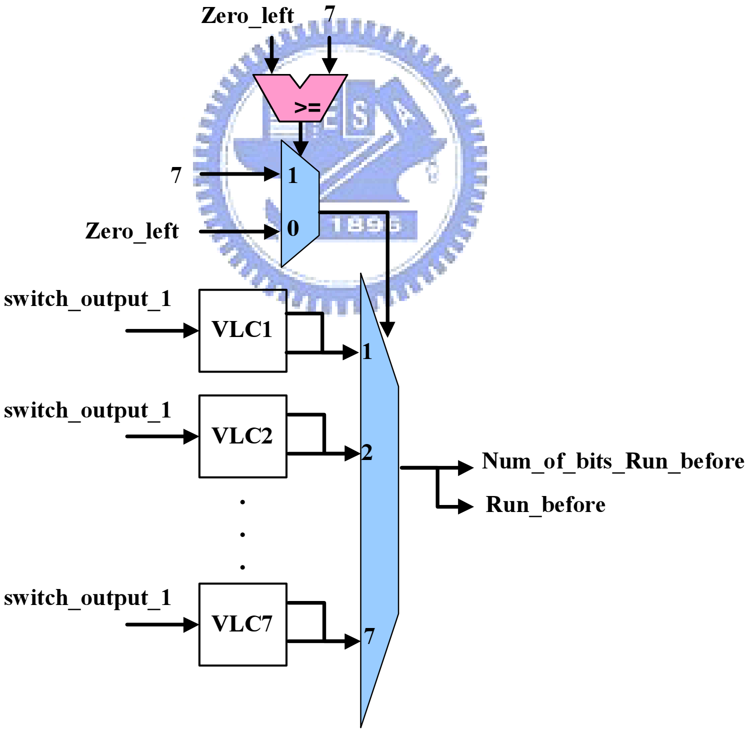


Fig 41. run before logic in Fig 39

The Run_before_total_ok signal is generated by the logic shown in Fig 42.

If there are no more zeros left to decode (i.e. Zero_left_in = Run_before), or run_before for the final (lowest frwquency) non-zero coefficient (i.e.

Temp_Total_coeff = 1), Run_before_total_ok is set to 1.

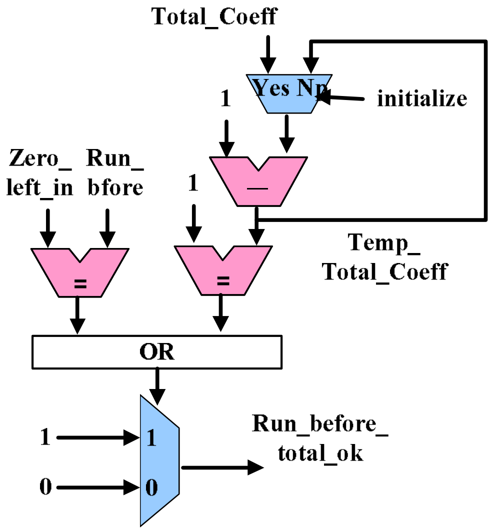


Fig 42. total ok logic in Fig 39

The Zero_left_out signal is generated by the logic shown in Chapter 1.

If Temp_Total_coeff is equal to 1, Zero_left_out is set to 0. Otherwise, Zero_left_out is set to (Zero_left – Run_before).

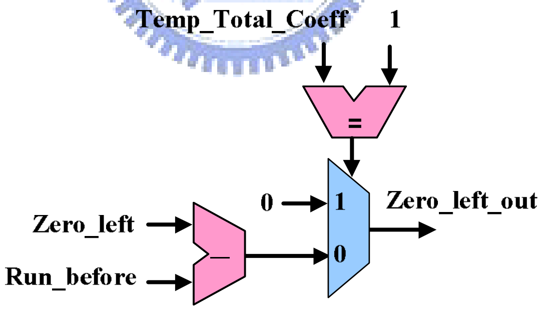
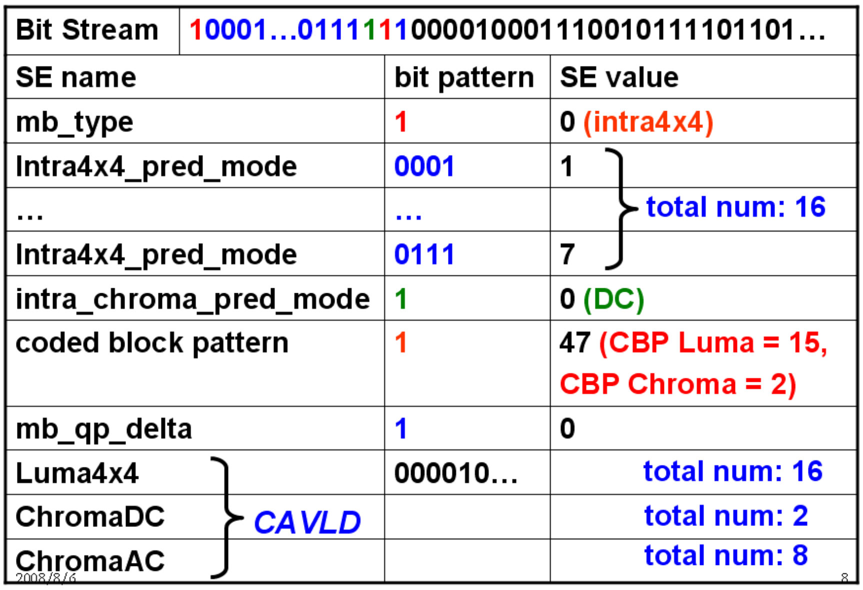


Fig 43. zero left out logic in Fig 39

3.10. A CAVLD Decoding Example

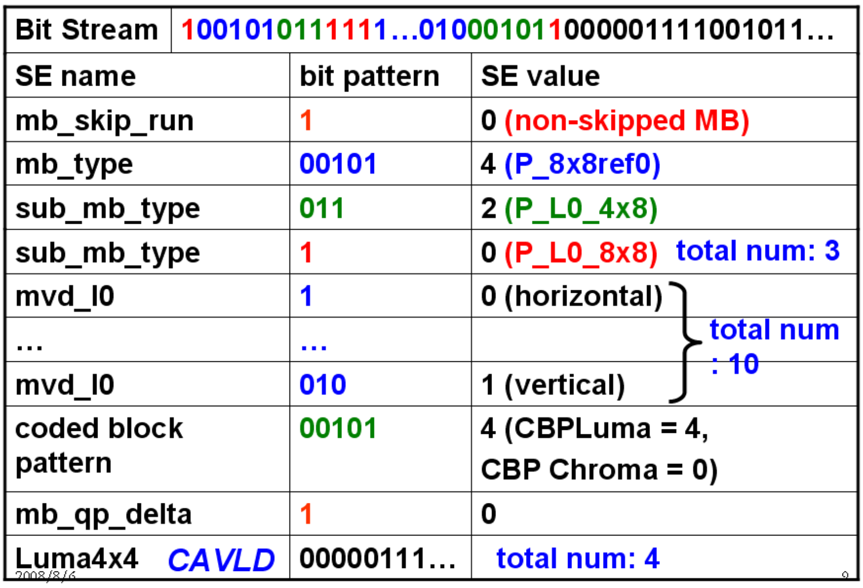
A bitstream example for intra frame and inter frame is shown in Table 3 and Table 4. The entropy encoding method of Luma4x4, ChromaDC and ChromaAC SEs are CAVLC, so we decode these SEs using the proposed CAVLD logic, and the entropy encoding method of other SEs are UVLC which is very simple, so we decode these SEs using software.



Bit Stream	10001…0111111000010001110010111101101…	
SE name	bit pattern	SE value
mb_type	1	0 (intra4x4)
Intra4x4_pred_mode	0001	1
…	…	total num: 16
Intra4x4_pred_mode	0111	7
intra_chroma_pred_mode	1	0 (DC)
coded block pattern	1	47 (CBP Luma = 15, CBP Chroma = 2)
mb_qp_delta	1	0
Luma4x4	000010…	total num: 16
ChromaDC CAVLD		total num: 2
ChromaAC		total num: 8

2008/8/6 8

Table 3. bitstream example for Intra frame



Bit Stream	1001010111111…010001011000001111001011…	
SE name	bit pattern	SE value
mb_skip_run	1	0 (non-skipped MB)
mb_type	00101	4 (P_8x8ref0)
sub_mb_type	011	2 (P_L0_4x8)
sub_mb_type	1	0 (P_L0_8x8) total num: 3
mvd_l0	1	0 (horizontal)
…	…	total num : 10
mvd_l0	010	1 (vertical)
coded block pattern	00101	4 (CBPLuma = 4, CBP Chroma = 0)
mb_qp_delta	1	0
Luma4x4 CAVLD	00000111…	total num: 4

2008/8/6 9

Table 4. bitstream example for Inter frame

An example for Coeff_token decoder is shown in Fig 44. If we have an input bitstream 0000100011100101111101101…, this stream will be processed with switch_input_1 logic, first-one detector logic and switch_input_2 logic, we can get the Si2_out like this: 100011100101111101101…. Then we use nc, the first four bit of Si2_out, 1000, and Leading_zeros as input singals of output logic1 to get the Total_coeff and Trailing_ones of the 4x4 block, and number of bits we used in stream of Coeff_token decoder,

The stream was partitioned into two sub-streams by underscore, the sub-stream before underscore, 0000100, stands for used bitstream, and the other sub-stream after underscore, 01110010111101101…, stands for unsed bitstream.

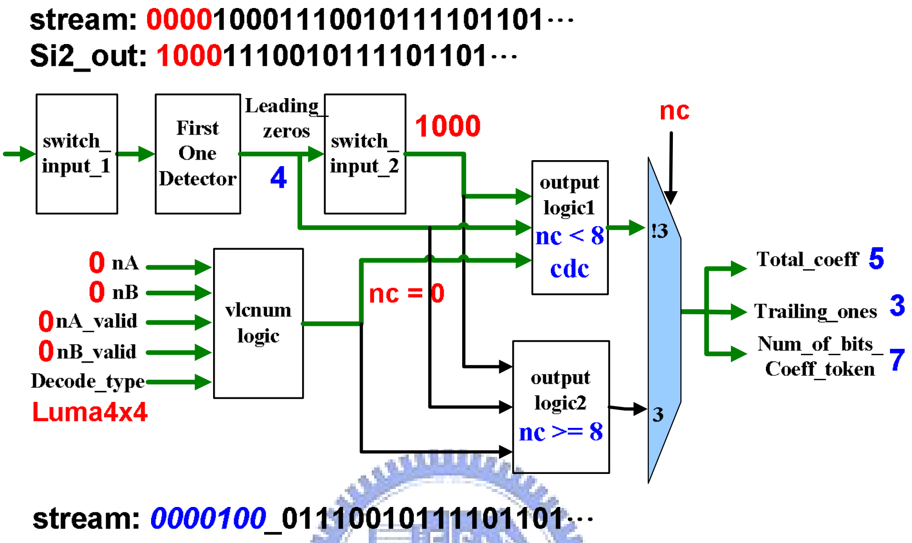


Fig 44. example for Coeff_token decoder

An example for Trailing_ones decoder is shown in Fig 45. Because we get the Trailing_ones of the 4x4 block in Coeff_token decoder, so we need to decide the signs of three trailing ones, and put these trialing ones into output_array.

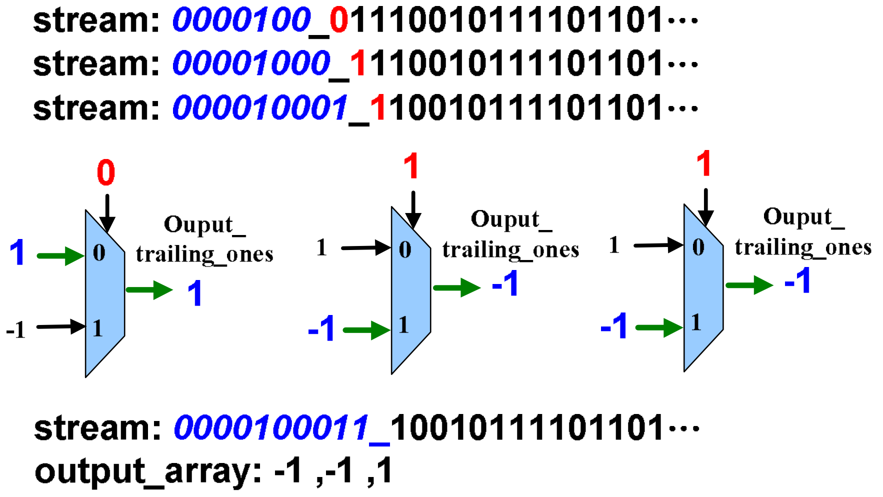

stream: 0000100_01110010111101101···
stream: 00001000_1110010111101101···
stream: 000010001_110010111101101···
0
1
-1
0
1
Ouput_ trailing_ones
1
1
1
-1
0
1
Ouput_ trailing_ones
-1
1
1
-1
0
1
Ouput_ trailing_ones
-1
stream: 0000100011_10010111101101···
output_array: -1 ,-1 ,1

Fig 45. example for Trailing_ones decoder

An example for Level decoder is shown in Fig 46 and Fig 47. We need to decode two levels in this example. First, we use the originl stream as input signal to switch_input_1 logic to get the Si1_out which stands for the unused stream in original stream, and then Si1_out is processed with level prefix, num of bits, level suffix and output level logics to get the Output_Level, and put the Output_Level into output_array.

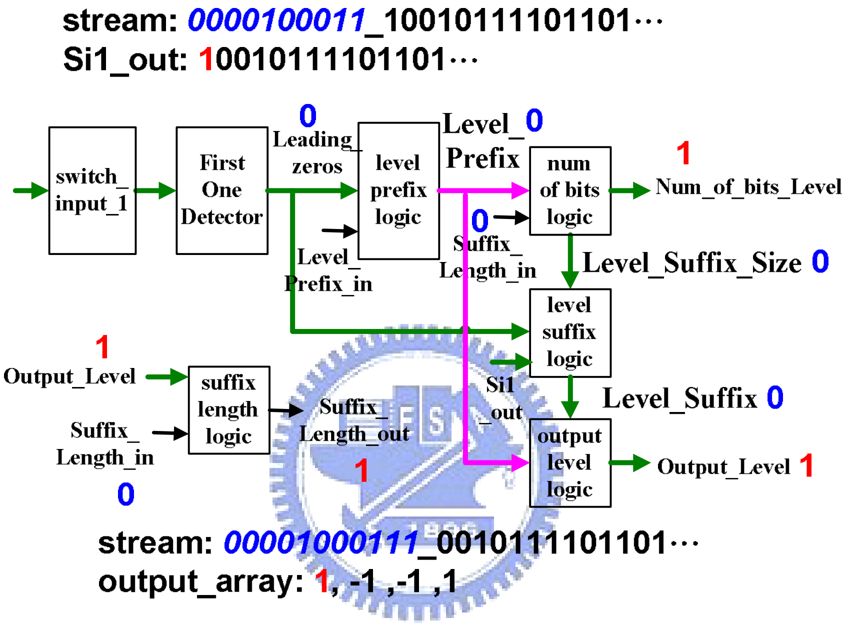


Fig 46. example for Level decoder (1/2)

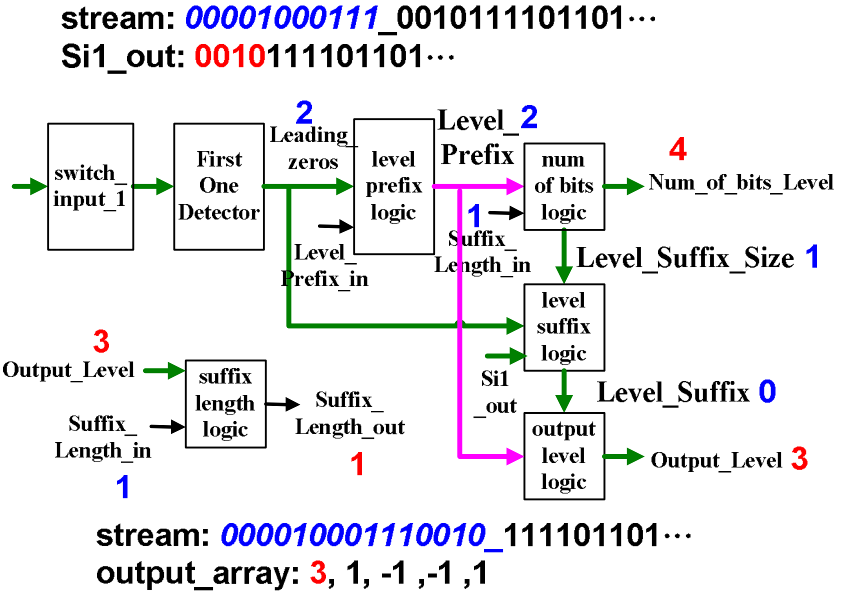


Fig 47. example for Level decoder (2/2)

An example for Total_zeros decoder is shown in Fig 48. We use Total_coeff and Si1_out as input signals to output logic1 to get the total number of zeros before the last coefficient and number of bits we used in Si1_out of Total_zeros decoder,

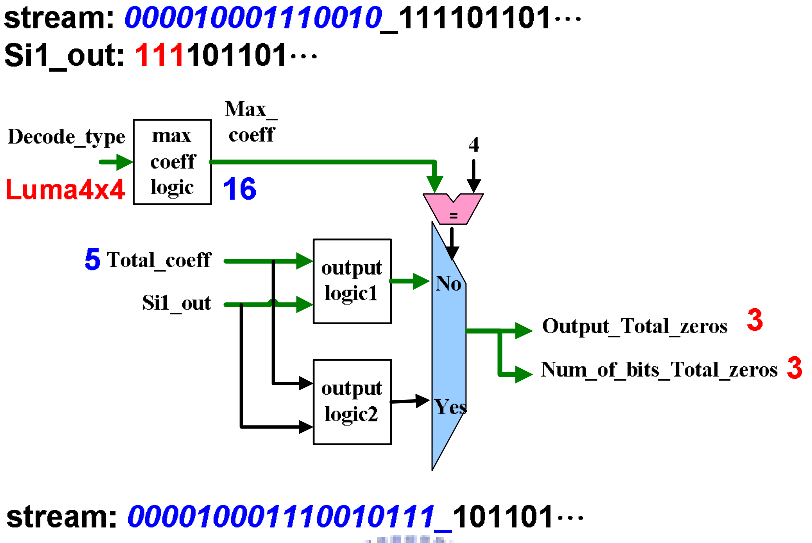


Fig 48. example for Total_zeros decoder

An example for Run_before decoder is shown in Fig 49 ~ Fig 52. We use the Zero_left and Si1_out as input signals to run before logic to get Run_before for each non-zero coefficient and number of bits we used in Si1_out of Run_before decoder. Then we insert the same number of zeros as Run_before of each non-zero coefficient and put them in output_array.

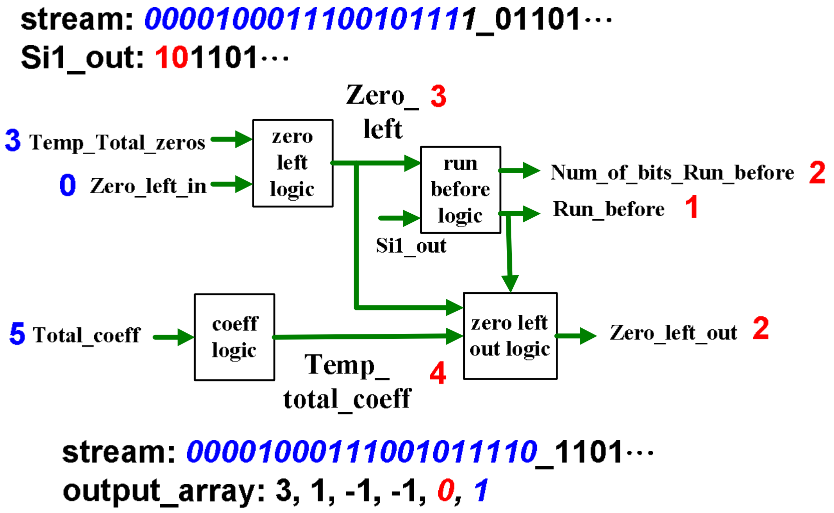
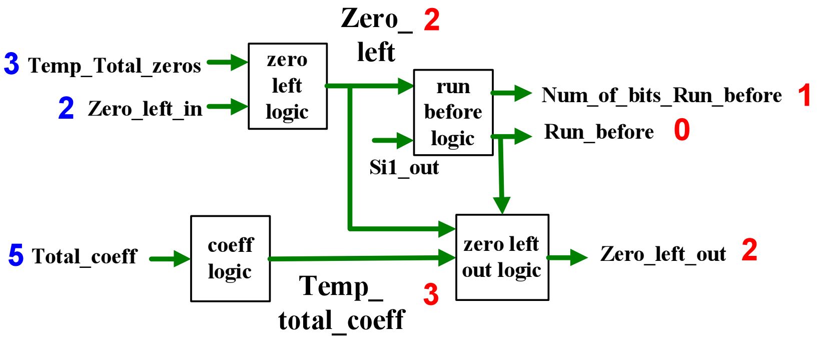


Fig 49. example for Run_before decoder (1/4)

stream: *00001000111001011110_*1101⋯

Si1_out: 1101⋯



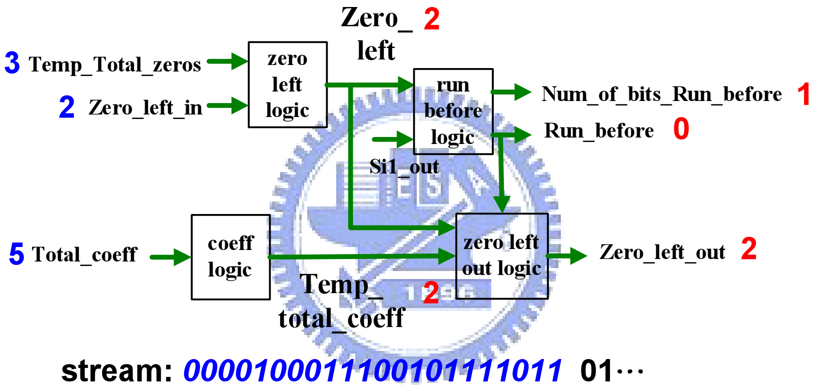
stream: *00001000111001011110*1*_*101⋯

output_array: 3, 1, -1, *-1*, 0, 1

Fig 50. example for Run_before decoder (2/4)

stream: *0000100011100101111101_*101⋯

Si1_out: 101⋯



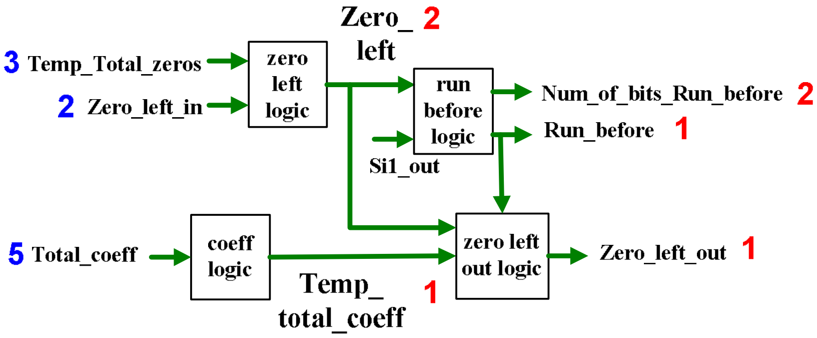
stream: *000010001110010111110*1*1_*01⋯

output_array: 3, 1, *-1*, -1, 0, 1

Fig 51. example for Run_before decoder (3/4)

stream: *00001000111001011111011_*01⋯

Si1_out: 01⋯



stream: *0000100011100101111101101_*⋯

output_array: 3, *0*, *1*, -1, -1, 0, 1

Fig 52. example for Run_before decoder (4/4)

It is not necessary to decode run_before for the final (lowest frequency) non-zero coefficient, so we put the remaining zeros in front of the final (lowest frequency) non-zero coefficient, and the output_array is: 0, 3, 0, 1, -1, -1, 0, 1. The final 4x4 coefficient block we decoded is

0	3	-1	0
0	-1	1	0
1	0	0	0
0	0	0	0

Chapter 4. Design of the CABAD Logic

As mentioned in section 1.2, the target platform is a LEON3-based FPGA development board (shown in Fig 3.). In this chapter, we will present our design of the CABAD logic and how it is integrated into the LEON platform. First, in section 4.1, we present the CABAD algorithm and give a example for this algorithm. In section 4.2, we present the overall architecture of the CABAD logic and its interface to the LEON processor and the memory subsystem. In section 4.3, we will present the binary arithmetic coding including normal decoding process, bypass decoding process and final decoding process. In section 4.4, we will present the initialization of context variables. In section 4.5, we will present the state controller for each syntax element controller. Then, we will present each syntax element decoding component of CABAD which converts the bit patterns to syntax elements, including MB Skip Flag (section 4.6), MB Type (section 4.7), Sub MB Type (section 4.8), Intra Prediction Mode for Luma4x4 (section 4.9), Intra Prediction Mode for Chroma (section 4.10), Reference Frame Index (section 4.11), Motion Vector Difference (section 4.12), Coded Block Pattern (section 4.13), MB Based Quantization Parameter (section 4.14), Coded Block Flag (section 4.15), Significant Map (section 4.16) and Level Information (section 4.17). In section 4.18, we will give a CAVLD decoding example.

4.1. Introduction to the CABAD Entropy Decoder

4.1.1. Algorithm of CABAD

The algorithm of CABAD is described as follows:

Step 1. Context and probability modeling:

The H.264 standard defines an extensive set of context information associated with syntax elements (SEs). In this step, context modeling selects the context index according to which SE is to be decoded and the previously decoded syntax information from top, left, or the current macroblock (MB).

Step 2. Binary arithmetic decoding:

In this step, the CABAC decoder decodes one bin of the bitstream and updates probability model.

Step 3. Binarization:

In this step, binarization is in charge of checking if the successive decoded bin (or bit) is in bin string. If not, the decoder will keep on decoding next bin. If yes, the decoder is prepared to decode the next SE.

After the introduction of RTL model of CAVLD logic, we will give a example in section 4.18.

4.2. Overall System Architecture

4.2.1. AMBA AHB BUS Wrapper for CABAD

In this section, the bus wrapper module of the CABAD logic for AMBA AHB bus protocol is similar to that described in 3.2.1. The main difference is that the bus wrapper module of the CABAD logic includes one block RAM which stores Ctx->MPS and Ctx->state for each symbol during the CABAD decoding process, as shown in Fig 53.

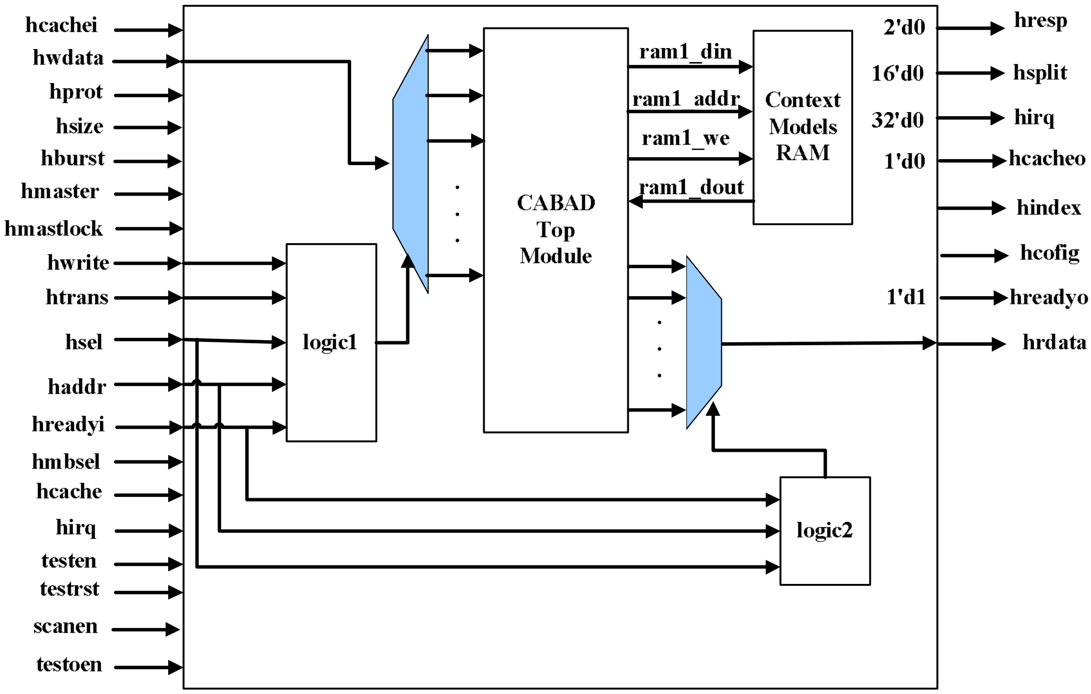


Fig 53. Architecture of the bus wrapper module

4.2.2. CABAD Top Module

CABAD top module includes all of the CABAD components. The interface and the architecture of the CABAD top module are shown in Fig 54, and the interface signals are explained as follows.

- a, b are each 8-bit input signal indicates the information needed for almost syntax element controllers. i.e. information for left and top macro-block of current macro-block.
- MVD_component is a 1-bit input signal indicates deciding the Ctx_id of MVD syntax element.
- top_value_in, top_range_in, and top_Dbitsleft_in are 25-bit, 9-bit and 5-bit input signals using for AC engine.
- which_SE is a 5-bit input signal indicates which syntax element is decoding on this time.
- next_16bits is a 16-bit input signal from the bitstream we will decode. If

AC_Dbitsleft_in is small or equal the number of bits to be consumed in AC engine, we will combine this signal with offset indicates output signal AC_value_out.

- slice_type is a 2-bit input signal indicates the type of the current slice which is decoded. There are four types of slice: I_slice, SI_slice, P_slice and B_slice..
- input_next_16bits_ok is a 1-bit input signal indicates the value of next_16bits is updated or not. If yes, the value of input_next_16bits_ok is equal to 1, otherwise, the value of input_next_16bits_ok is equal to 0.
- initial_ram is a 1-bit input signal to enable Context variable Initialization..
- img_qp is a 8-bit input signal indicates the quantized parameter of the current slice which will be decoded
- model_number is a 2-bit input signal corresponding to fixed decision for inter slices. The value of model_number can be 0, 1 and 2.
- last_dquant_in is a 8-bit input signal which is indicating for the preceding macro-block of current macro-block in decoding order.
- fld is a 1-bit input signal indicates the method of scanning in macro-block is zig-zag scan or interlace scan.
- ram1_dout is a 16-bit input signal indicates the Ctx->state and Ctx->MPS from the Context Models RAM.
- Coeff_in is a 16-bit input signal. Each bit of this signal indicates there is zero or non-zero coefficient in corresponding position in a given scanning order for a block. If there is a non-zero coefficient in corresponding position, the corresponding bit of Coeff_15_to_0 is set to 1, otherwise, the corresponding bit of Coeff_15_to_0 is set to 0.
- use_next_16bits_wire is a 1-bit output signal indicates requesting the

processor to update the value of next_16bits.

- top_value_out_wire, top_range_out_wire and top_Dbitsleft_out_wire are 25-bit, 9-bit and 5-bit output signals from syntax element controllers.
- SE_value1_wire and SE_value2_wire are each 8-bit output signals indicates the value of the syntax element we decoded.
- SE_context_wire is a 8-bit output signal indicates the type of current macro-block.
- initial_ram_ok is a 1-bit output signal indicates end of Context variable Initialization.
- SE_OK_wire is a 1-bit output signal indicates whether we have finished decoding one syntax element or not.
- last_dquant_wire is a output signal indicates the value of dquant.
- ram1_din_wire is a 16-bit output signal indicates the context variables which will be stored in block ram.
- ram1_addr_wire is a 10-bit output signal indicates the location of the context variables which will be stored in block ram.
- ram1_we_wire is a 1-bit output signal to enable the block ram for context variables.
- Coeff_0_out ~ Coeff_15_out are each 16-bit output signal indicating the value of the significant coefficients in corresponding position of a block.

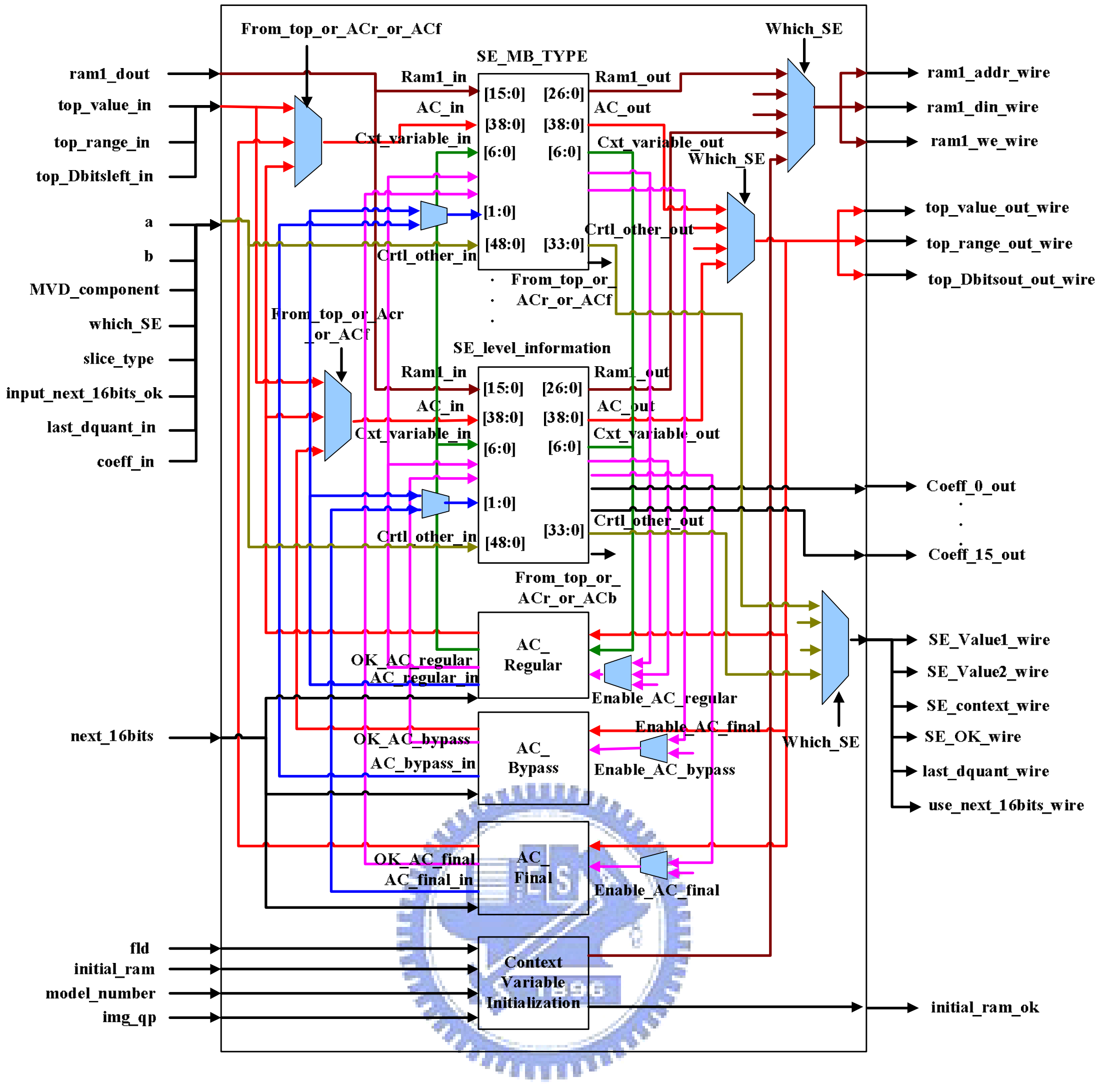


Fig 54. Overall architecture of the CABAD top module

4.3. Binary Arithmetic Coding

The principle of binary arithmetic coding is based on recursive interval subdivision of the interval which is called R (Range). The interval is subdivided into two subintervals according to the given estimation of the probability P_{LPS} of Least Probable Symbol (*LPS*): one interval is $rLPS = R{\cdot}P_{LPS}$ which is assigned to the *LPS,* and the other interval is $rMPS = R - rLPS$, which is assigned to the *MPS.* Depending on offset falls into *rLPS (LPS occur)* or *rMPS (MPS occur),* the corresponding subinterval is chosen as the new interval.

In practical implementation of a binary arithmetic codec, there are two main factors determine the throughput. The first one is the multiplication operation in the expression $rLPS = R \cdot P_{LPS}$, which calculates the interval subdivision. The other one is the update of new probability P_{LPS}. CABAD inherits the table-based mechanism in Q-coder but provide better compression efficiency. In CABAD, in order to decrease the complexity of computing $R \cdot P_{LPS}$, the multiplication results are pre-stored in a fixed table. Range value is approximated by four quantized values using an equal-partition of the whole range $2^8 \le R \le 2^9$ into four cells. The value of P_{LPS} is approximated by 64 quantized values indexed by the 6-bit state value. According to above terms, the approximated subinterval range values *rLPS* is table-based by looking up from a 4x64 2D table. State value is updated after decoding each bin by looking up a fixed table indexed of original state and *MPS_or_LPS_occur* which decides the new subinterval is *rLPS* or *rMPS*. To used fixed precision integer arithmetic, the range and offset must be renormalized after decoding each bin. During the renormalization process, the range is left shift to guarantee the most significant bit of range is always 1.

The binary arithmetic coding in Context-based Adaptive Binary Arithmetic Decoding (CABAD) is divided into three different processes: normal decoding process, bypass decoding process and final decoding process. Each decoding process is explained as follows.

4.3.1. Normal Decoding Process (AC_Regular_mode)

This process is invoked to decode a Bin as output in almost syntax element controllers. It takes dynamically estimating probability P_{LPS} and Most Probable Symbol as inputs to decide the output value of Bin, and it may consume multiple bits of bitstream we will decode in renormalization.

4.3.1.1. Interface of Normal Decoding Process

The interface of normal decoding process is shown in Fig 55, and the interface signals are explained as follows.

- AC_Enable_regular is a 1-bit input signal to enable this normal decoding process.
- AC_state_in is a 6-bit input signal indicates dynamically estimating probability P_{LPS}. The value of P_{LPS} is approximated by 64 quantized value indexed by this signal.
- AC_range_in is a 9-bit input signal indicates the interval of this decoding process, and the value of this signal is must greater and equal than 2^8 and small than 2^9.
- AC_Dbitsleft_in is a 5-bit input signal indicates how many bits are not consumed to be offset in AC_value_in signal.
- AC_value_in is a 25-bit input signal composed of 9-bit offset and some bits from bitstream and the length is AC_Dbitsleft_in. The advantage of doing this is saving lots of memory access to get bits from bitstream in renormalization.
- AC_MPS_in is a 1-bit input signal indicates the Most Probable Symbol.
- AC_bitstream_next_16bits is a 16-bit input signal from the bitstream we will decode. If AC_Dbitsleft_in is small or equal the number of bits to be consumed in decoding process, we will combine this signal with offset indicates output signal AC_value_out.
- OK_AC_regular is a 1-bit output signal indicates end of this process. This process can be done in only one cycles, so when AC_Enable_regular is set to 1 in this cycles, OK_AC_regular is set to 1 in next cycles

- AC_range_out(9-bits), AC_state_out(6-bit), AC_Dbitsleft_out(5-bit), AC_value_out(25-bit), AC_MPS_out(1-bit) and AC_Bin_out(1-bit) are output signals, and the value of these are computed by AC_range_in, AC_state_in, AC_Dbitsleft_in, AC_value_in, AC_MPS_in and AC_MPS_in input signals and some internal signals shown in Fig 55.
- Use_AC_bitstream_next_16bits is a 1-bit output signal indicates whether the AC_bitstream_next_16bits has used or not.

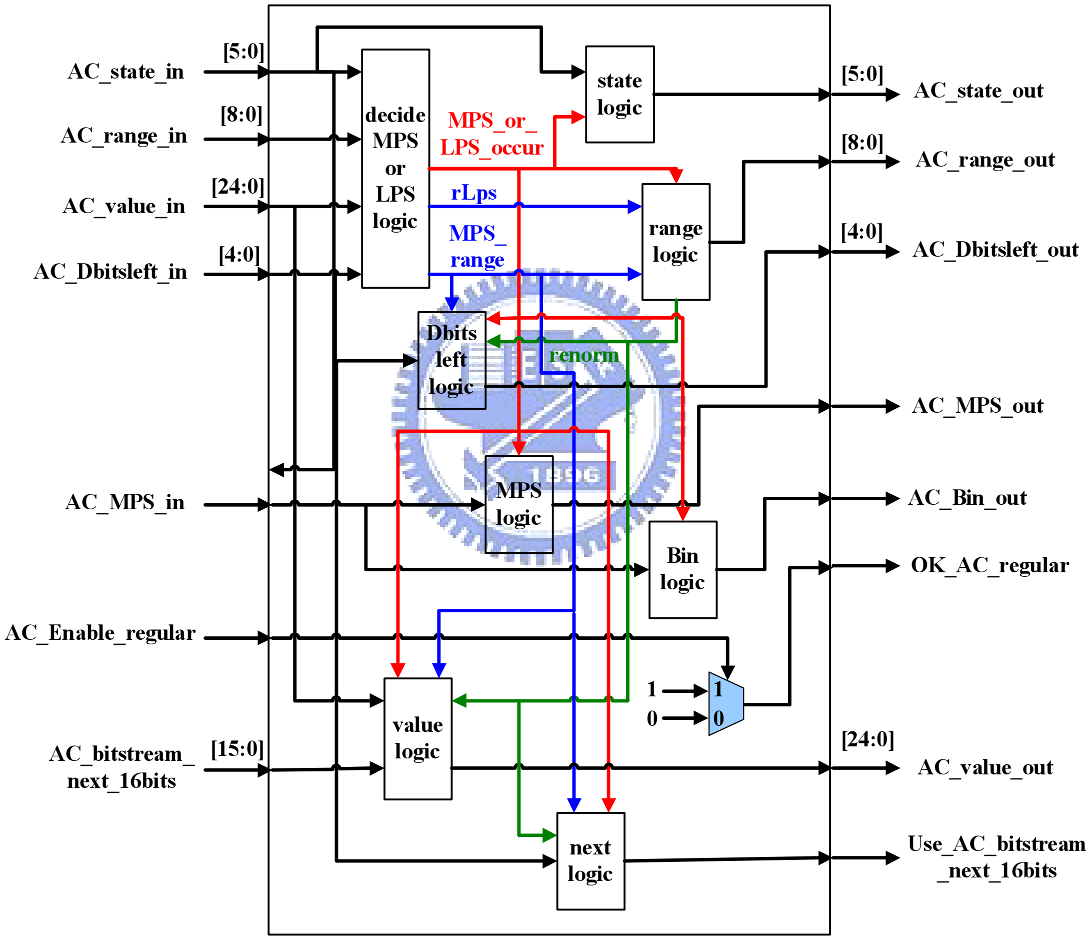
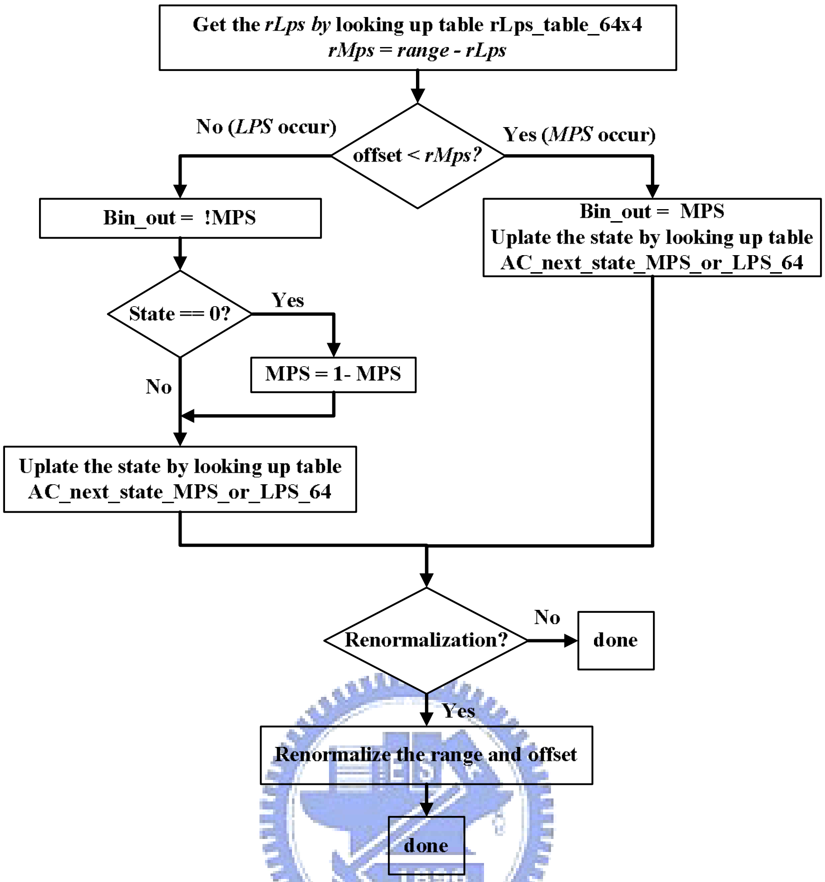


Fig 55. Interface of normal decoding process

4.3.1.2. Flow Chart of Normal Decoding Process



Get the *rLps by* looking up table rLps_table_64x4
rMps = range - rLps
offset < *rMps?*
No (*LPS* occur)
Yes (*MPS* occur)
Bin_out = !MPS
Bin_out = MPS
Uplate the state by looking up table
AC_next_state_MPS_or_LPS_64
State == 0?
Yes
No
MPS = 1- MPS
Uplate the state by looking up table
AC_next_state_MPS_or_LPS_64
Renormalization?
No
done
Yes
Renormalize the range and offset
done

Fig 56. Flow chart of normal decoding process

The flow chart of normal decoding process is shown in Fig 56.

4.3.1.3. Architecture of Normal Decoding Process

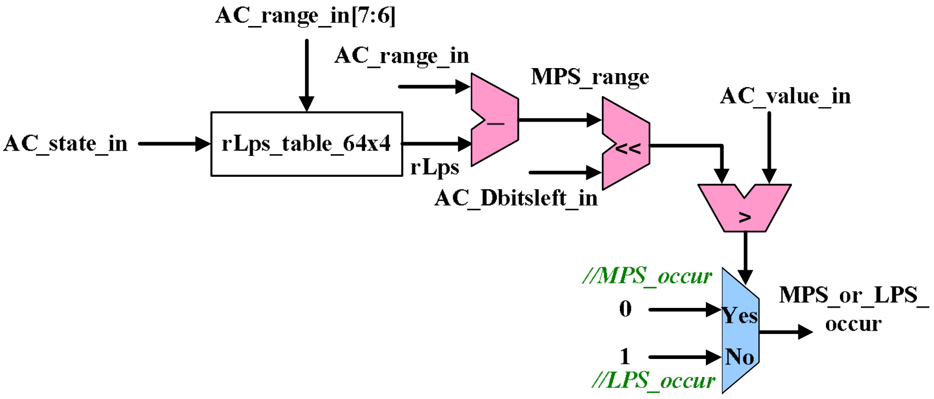


Fig 57. decide MPS or LPS logic in Fig 55

The MPS_or_LPS_occur signal is generated by the logic shown in Fig 57.

To get the value of rLps, we looked up a fixed 64x4 2-D table indexed of AC_state_in and AC_range_in[7:6]. Because AC_range_in is composed of rLps and rMps(MPS_range), so MPS_range is equal to (AC_range_in – rLps). To decide the value of MPS_or_LPS_occur, we compared the value of AC_range_in with offset in AC_value_in. If offset is greater than AC_range_in, MPS_or_LPS_occur is set to 0 indicating MPS-occur. Otherwise, MPS_or_LPS_occur is set to 1 indicates LPS-occur.

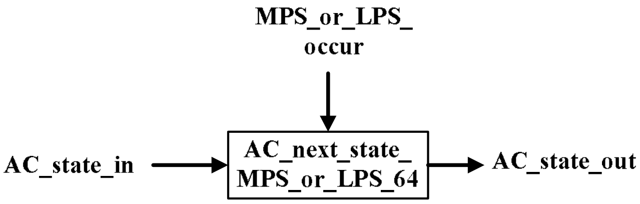


Fig 58. state logic in Fig 55

The AC_state_out signal is generated by the logic shown in Fig 58.

To get the value of AC_state_out, dynamically estimating probability P_{LPS} for next decoding process, we looked up a fixed 64x2 2-D table indexed of AC_state_in and MPS_or_LPS_occur.

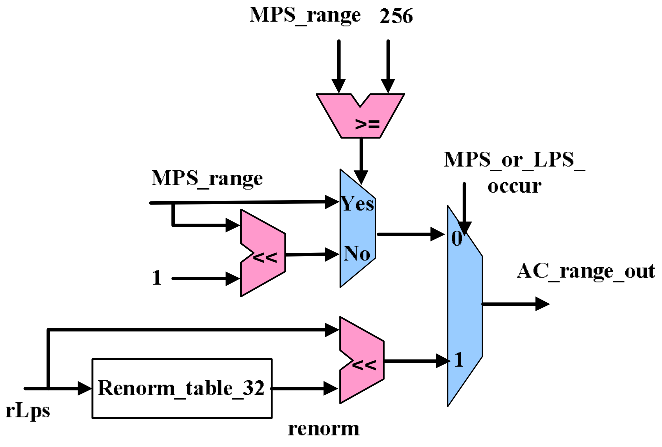


Fig 59. range logic in Fig 55

The AC_range_out signal is generated by the logic shown in Fig 59.

AC_range_out is depending on how many bits are needed in renormalization of MPS_range or rLps.

If MPS occur, we have only zero or one bit in renormalization, so AC_range_out is set to MPS_range or (MPS_range << 1). Otherwise, if LPS-occur, we looked up a fixed table indexed of rLps to get renorm, so AC_range_out is set to (rLps << renorm).

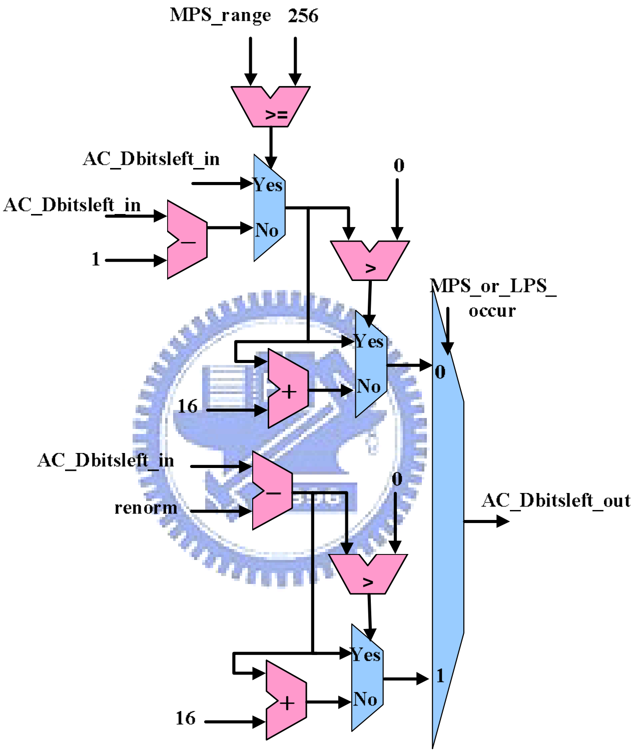


Fig 60. dbitsleft logic in Fig 55

The AC_Dbitsleft_out signal is generated by the logic shown in Fig 60.

If there are not enough bits in AC_value_in signal to consume of next decoding process, the value of AC_Dbitsleft_out will need to add 16.

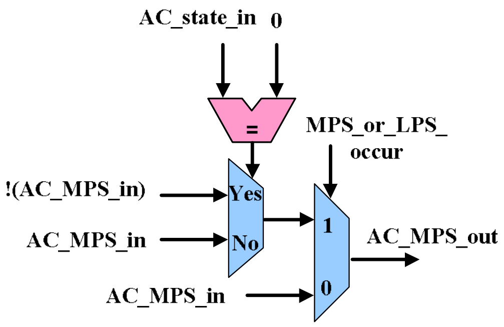


Fig 61. MPS logic in Fig 55

The AC_MPS_out signal is generated by the logic shown in Fig 61.

if LPS-occur and AC_state_in is equal to zero, AC_MPS_out is a inverse of AC_MPS_in. Otherwise, AC_MPS_out is set to AC_MPS_in.

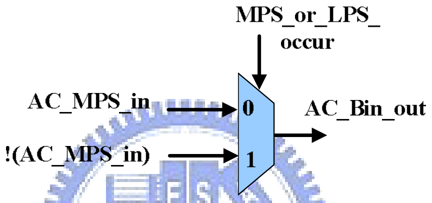


Fig 62. Bin logic in Fig 55

The AC_Bin_out signal is generated by the logic shown in Fig 62.

AC_Bin_out is depending on LPS-occur or MPS-occur, if LPS-occur, AC_Bin_out is a inverse of AC_MPS_in. Otherwise, AC_Bin_out is set to AC_MPS_in.

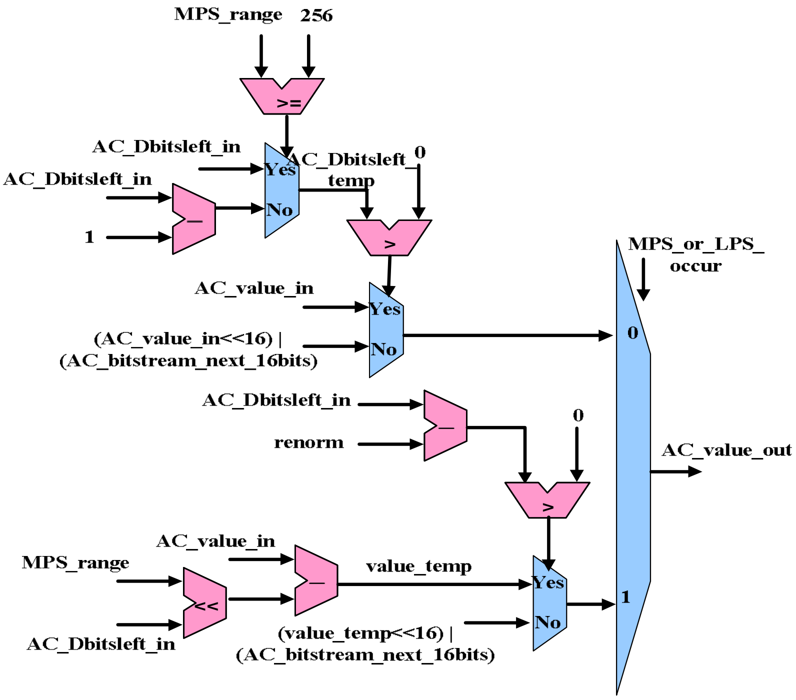


Fig 63. value logic in Fig 55

The AC_value_out signal is generated by the logic shown in Fig 63.

If MPS-occur, AC_value_out is set to AC_value_in or ((AC_value_in << 16) | AC_bitstream_next_16bits) depending on the value of AC_DbitsLeft_temp.

if LPS-occur, AC_value_out is set to value_temp or ((value_temp << 16) | AC_bitstream_next_16bits) depending on the value of (AC_Dbitsleft_in - renorm).

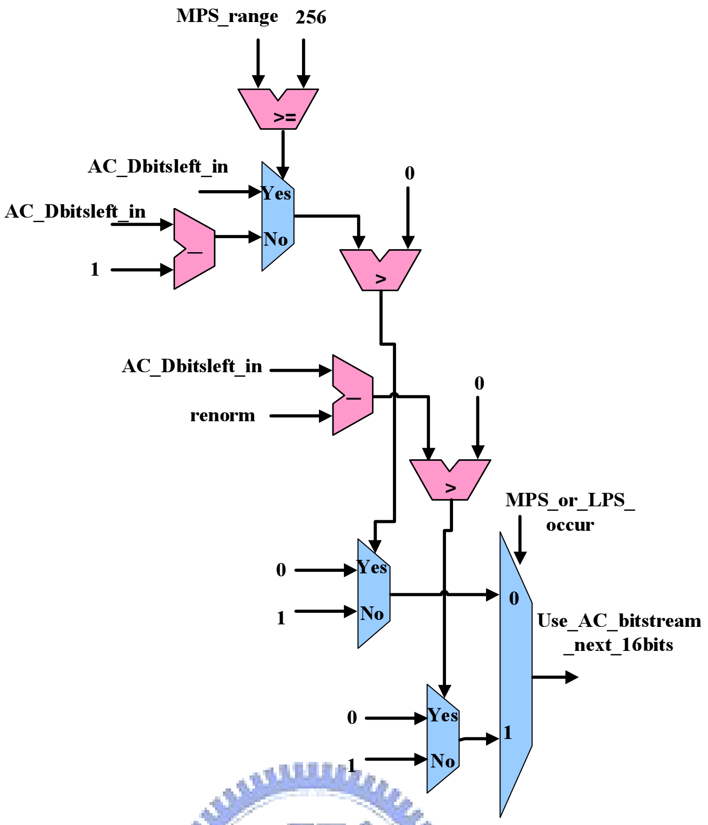


Fig 64. Next logic in Fig 55

The Use_AC_bitstream_next_16bits signal is generated by the logic shown in Fig 64.

If there are not enough bits to consume in AC_value_in signal of next decoding process, Use_AC_bitstream_next_16bits is set to 1. Otherwise, Use_AC_bitstream_next_16bits is set to 0.

4.3.2. Bypass Decoding Process (AC_Bypass_mode)

This process is invoked to decide the sign value of motion vector and level, or it is invoked in unary/k-th order Exp-Golomb (UEGK) binarzation process of motion vector and level information syntax element controllers.

As the different as normal decoding process, the value of dynamically estimating probability P_{LPS} is set to 0.5. It directly read one bit from the bitstream and produces one Bin as output. So we don't need the information about dynamically estimating probability P_{LPS} and Most Probable Symbol.

4.3.2.1. Interface of Bypass Decoding Process

The interface of bypass decoding process is shown in Fig 65, and the interface signals are similar with normal decoding process, so we don't explain these signals here.

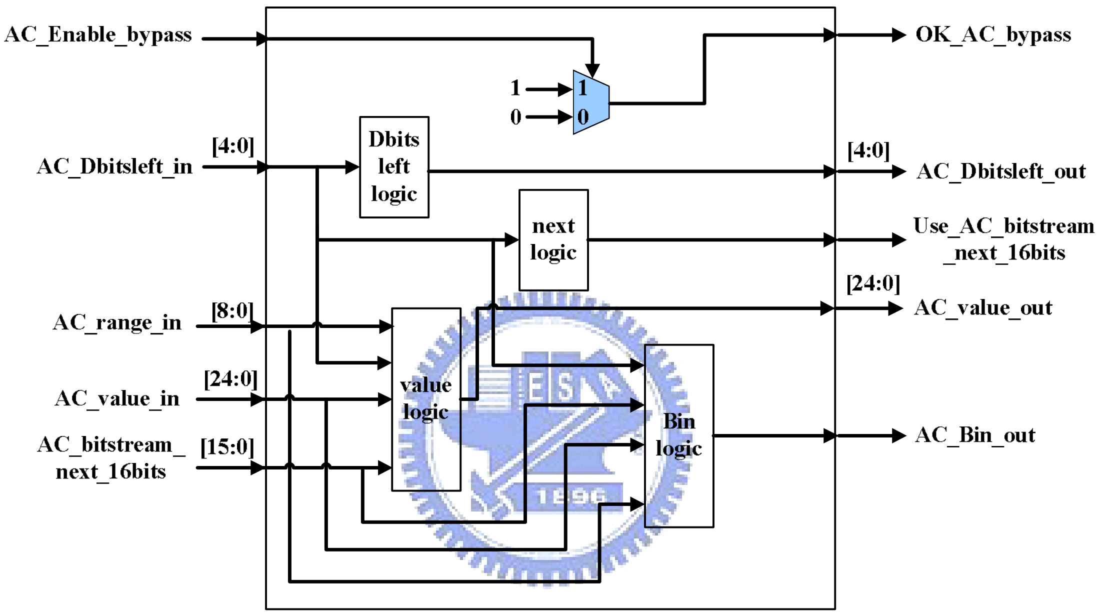


Fig 65. Interface of bypass decoding process

4.3.2.2. Flow Chart of Bypass Decoding Process

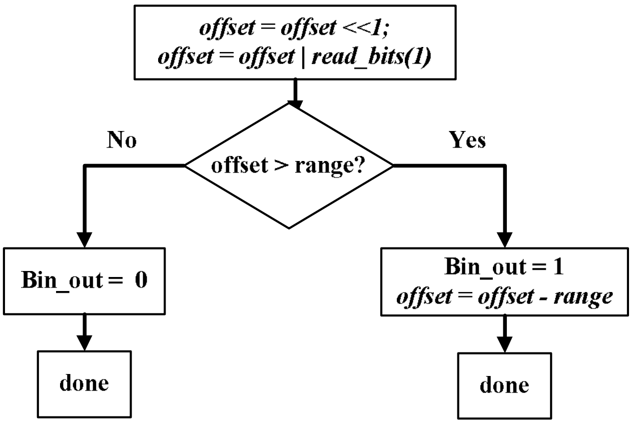


Fig 66. Flow chart of bypass decoding process

The flow chart of bypass decoding process is shown in Fig 66.

4.3.2.3. Architecture of Bypass Decoding Process

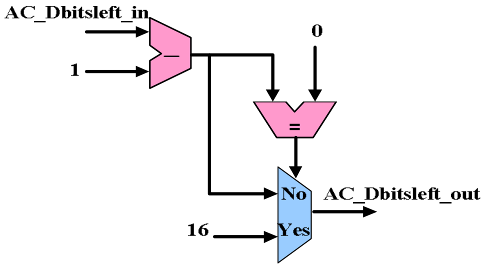


Fig 67. dbitsleft logic in Fig 65

The AC_Dbitsleft_out signal is generated by the logic shown in Fig 67.

If (AC_Dbitsleft_in – 1) is greater than 0, AC_Dbitsleft_out is set to (AC_Dbitsleft_in – 1). Otherwise, AC_Dbitsleft_out is set to 16.

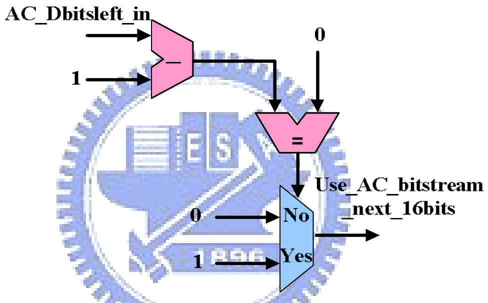


Fig 68. next logic in Fig 65

The Use_AC_bitstream_next_16bits signal is generated by the logic shown in Fig 68.

Use_AC_bitstream_next_16bits is depending on whether there are enough bits to consume in AC_value_in signal of next decoding process or not. If yes, Use_AC_bitstream_next_16bits is set to 0. Otherwise, Use_AC_bitstream_next_16bits is set to 1.

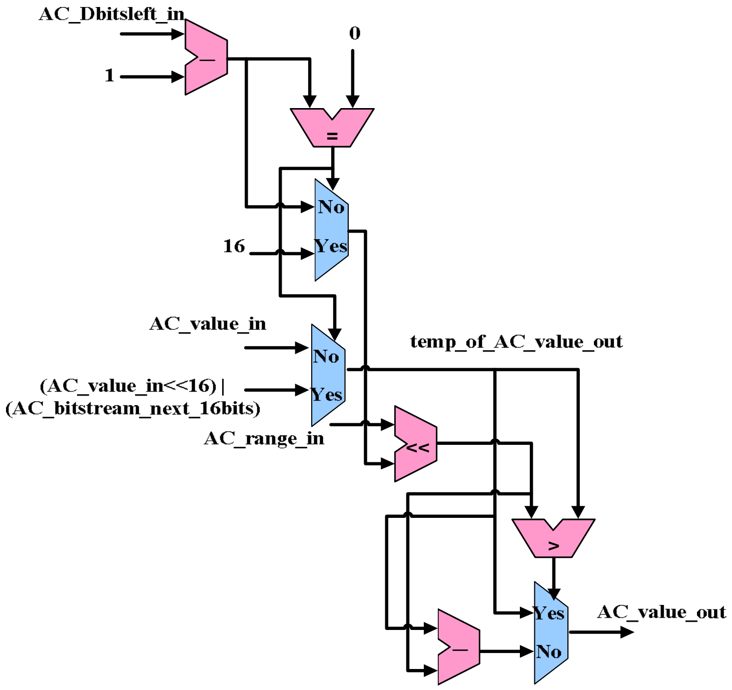


Fig 69. value logic in Fig 65

The AC_value_out signal is generated by the logic shown in Fig 69.

If there are not enough bits to consume in AC_value_in signal of next decoding process, temp_of_AC_value_out is set to ((AC_value_in << 16) | AC_bitstream_next_16bits). Otherwise, temp_of_AC_value_out is set to AC_value_in. And then if AC_range_in is greater than the offset in temp_of_AC_value_out signal, AC_value_out is set to temp_of_AC_value_out. Otherwise, AC_value_out is set to (temp_of_AC_value_out - (AC_range_in << AC_Dbitsleft_out)).

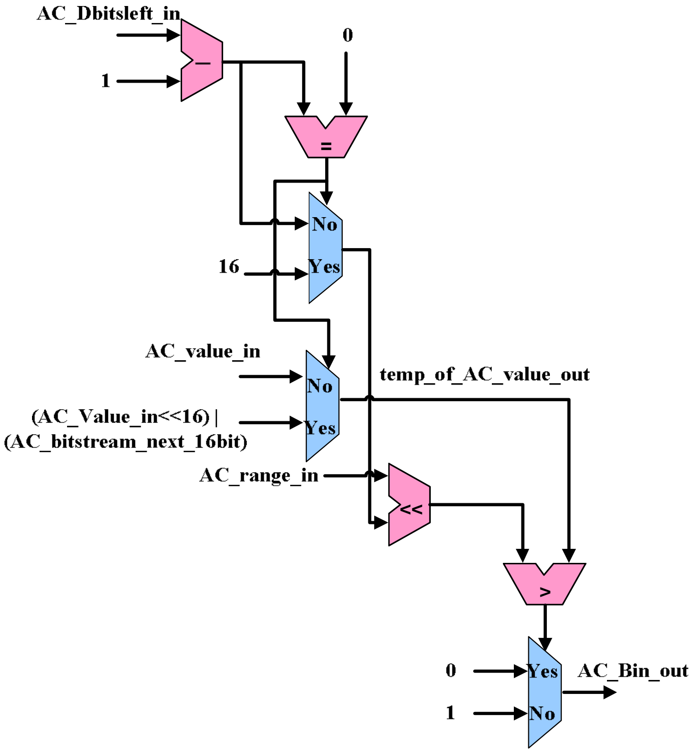


Fig 70. Bin logic in Fig 65

The AC_Bin_out signal is generated by the logic shown in Fig 70.

If AC_range_in is greater than the offset in temp_of_AC_value_out signal, AC_Bin_out is set to 0. Otherwise, AC_Bin_out is set to 1.

4.3.3. Final Decoding Process (AC_Final_mode)

This process is invoked to decode of end_of_slice_flag and the bin indicating the I_PCM mode. It directly read one bit from the bitstream and produces one Bin as output.

As the similar as bypass decoding process, we don't need the information about dynamically estimating probability P_{LPS} and Most Probable Symbol.

4.3.3.1. Interface of Final Decoding Process

The interface of final decoding process is shown in Fig 71, and the interface signals are similar with normal decoding process, so we don't explain these signals here.

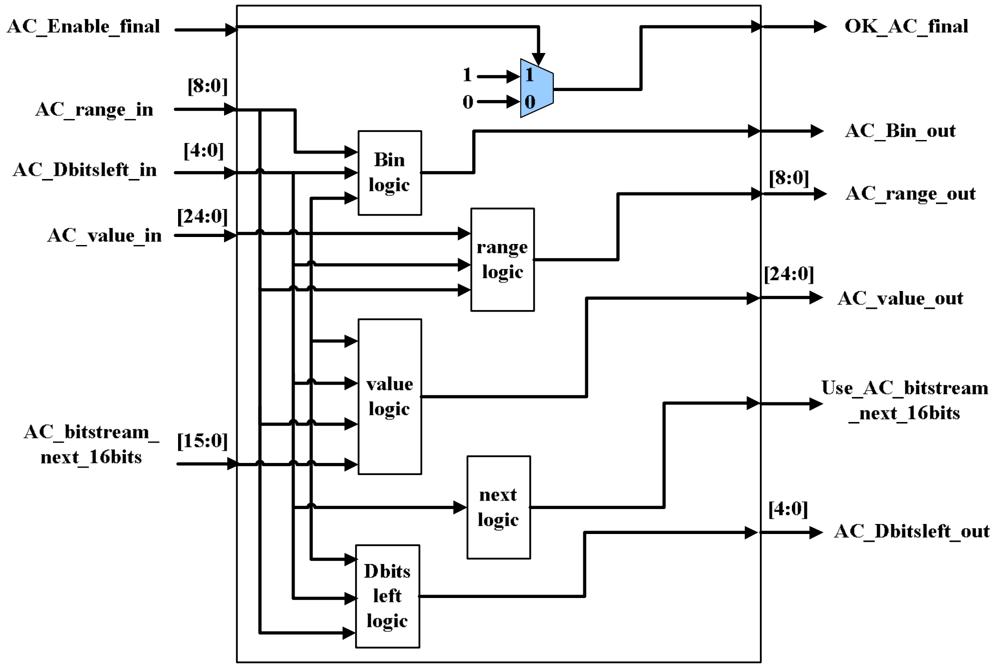


Fig 71. Interface of final decoding process

4.3.3.2. Flow Chart of Final Decoding Process

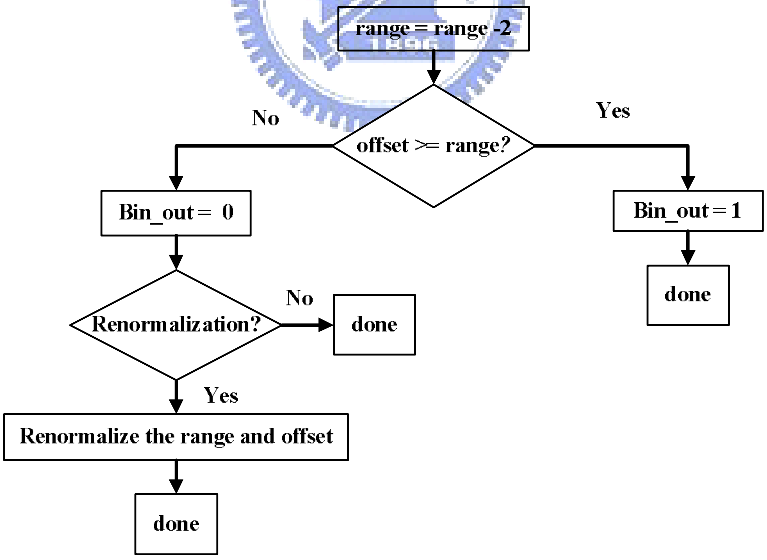


Fig 72. Flow chart of the final decoding process

The flow chart of final decoding process is shown in Fig 72.

4.3.3.3. Architecture of Final Decoding Process

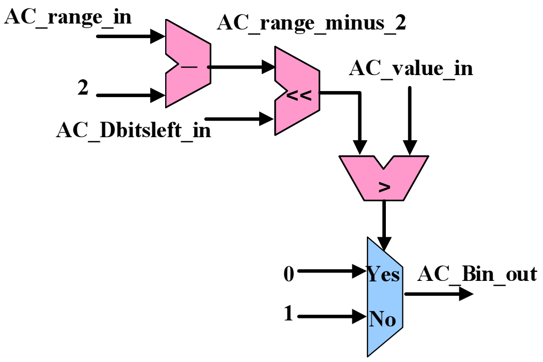


Fig 73. Bin logic in Fig 71

The AC_Bin_out signal is generated by the logic shown in Fig 73.

If AC_range_in_minus_2 is greater than the offset in AC_value_in signal, AC_Bin_out is set to 0. Otherwise, AC_Bin_out is set to 1.

If AC_Bin_out is 1, it indicates finish decoding the current slice. Otherwise, if AC_Bin_out is 0, it indicates finish decoding the current macro block.

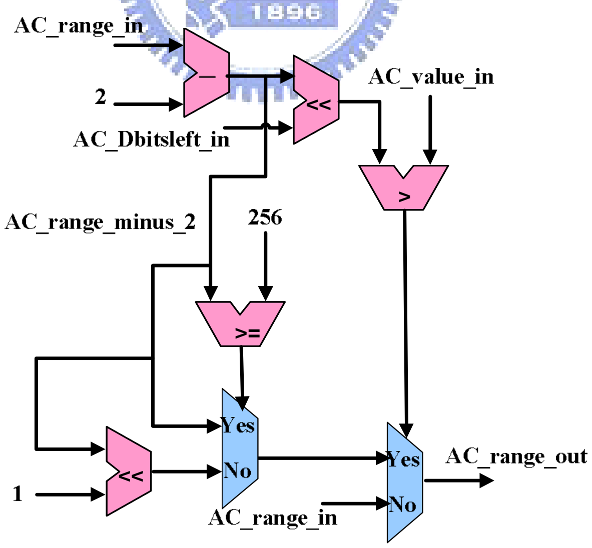


Fig 74. range logic in Fig 71

The AC_range_out signal is generated by the logic shown in Fig 74.

If AC_range_in_minus_2 is small or equal than the offset in AC_value_in signal, AC_range_out is set to AC_range_in. Otherwise, AC_range_out is depending on whether the AC_range_in_minus_2 needed to renormalize. If yes, AC_range_out is set to (AC_range_in_minus_2 << 1). Otherwise, AC_range_out is set to AC_range_in_minus_2.

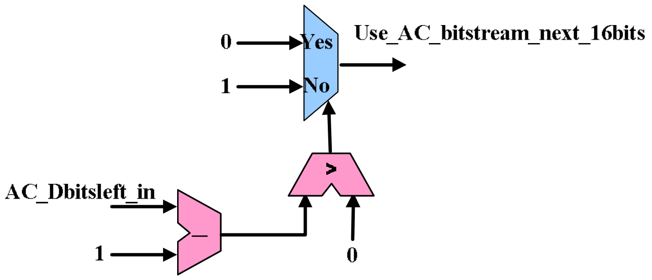


Fig 75. Next logic in Fig 71

The Use_AC_bitstream_next_16bits signal is generated by the logic shown in Fig 75.

Use_AC_bitstream_next_16bits is depending on whether there are enough bits to consume in AC_value_in signal of next decoding process or not. If yes, the value of Use_AC_bitstream_next_16bits is set to 0. Otherwise, the value of Use_AC_bitstream_next_16bits is set to 1.

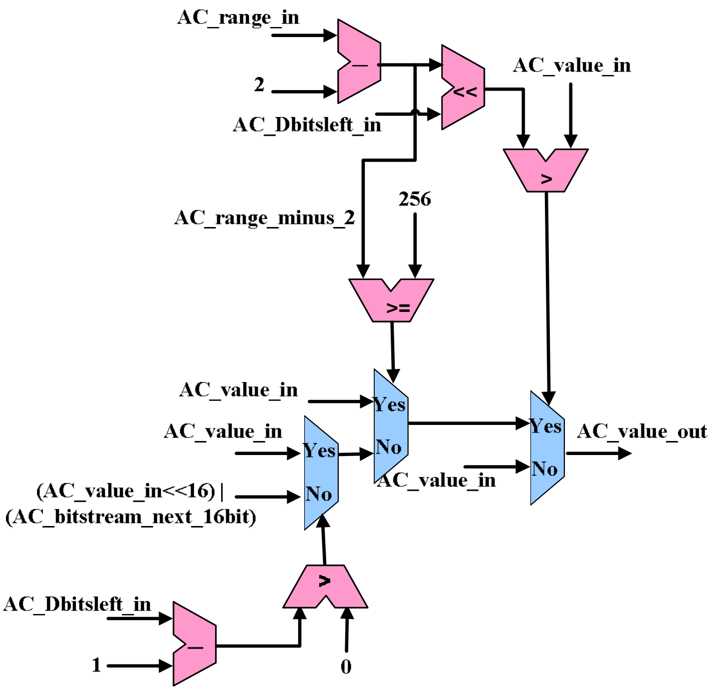


Fig 76. value logic in Fig 71

The AC_value_out signal is generated by the logic shown in Fig 76.

If AC_range_in_minus_2 is greater than the offset in AC_value_in signal, and the most significant bit of AC_range_in_minus_2 is zero, and there are not enough bits to consume in AC_value_in signal of next decoding process, AC_value_out is set to ((AC_value_in << 16) | AC_bitstream_next_16bits). Otherwise, AC_value_out is set to AC_value_in.

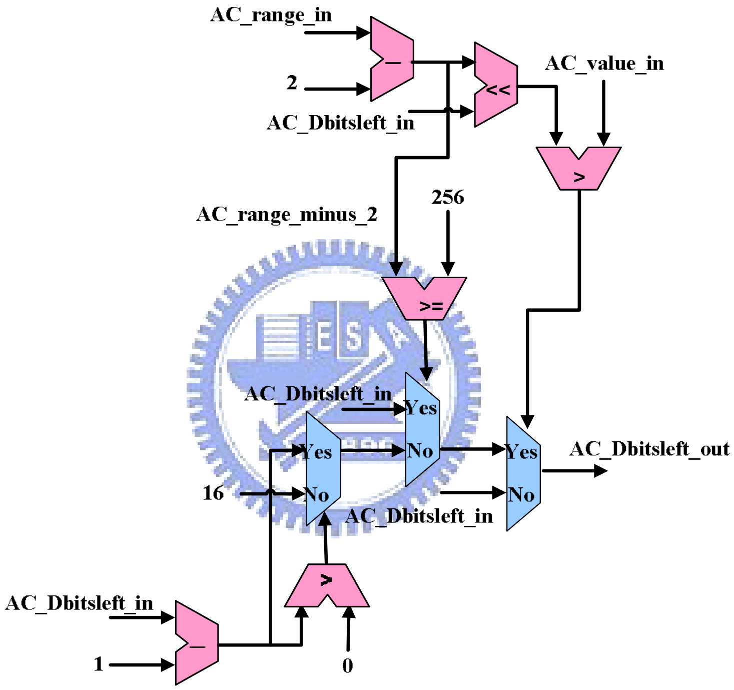


Fig 77. dbitsleft logic in Fig 71

The AC_Dbitsleft_out signal is generated by the logic shown in Fig 77.

If AC_range_in_minus_2 is greater than the offset in AC_value_in signal, and the most significant bit of AC_range_in_minus_2 is zero, AC_Dbitsleft_out is depending on whether there are enough bits to consume in AC_value_in signal of next decoding process or not. If no, AC_Dbitsleft_out is set to 16. Otherwise, AC_Dbitsleft_out is set to (AC_Dbitsleft_in – 1).

Otherwise, the AC_Dbitsleft_out is set to AC_Dbitsleft_in.

4.4. Initialization for context variables

For each context variable, there are two variables Ctx->state and Ctx->MPS which are initialized using m and n pre-stored in ROM. Ctx->state corresponds to estimating probability P_{LPS} and Ctx->MPS corresponds to the value of most probable symbol. And then we stored Ctx->state and Ctx->MPS in Block-RAM for all syntax element decoders as input information for decoding Bins. We supported 258 context variables for I_slice, and 293 context variables for P_slice and B_slice.

4.4.1. Interface of Initialization for context variables

The interface of the context variables initialization module is shown in Fig 78, and the interface signals are explained as follows.

- initial_ram is a 1-bit input signal to enable context variable initialization..
- slice_type is a 2-bit input signal indicates the type of the current slice which is decoded. There are four types of slice: I_slice, SI_slice, P_slice and B_slice.
- img_qp is a 8-bit input signal indicates the quantized parameter of the current slice which will be decoded.
- model_number is a 2-bit input signal corresponding to fixed decision for inter slices. The valur of model_number can be 0, 1 and 2.
- fld is a 1-bit input signal indicates the method of scanning in macro-block is zig-zag scan or interlace scan.
- initial_ram_ok is a 1-bit output signal indicates end of this process.
- ram_din is a 16-bit output signal indicates the context variables which will be stored in block ram.
- ram_addr is a 10-bit output signal indicates the location of the context

variables which will be stored in block ram.

- ram_we is a 1-bit output signal to enable the block ram for context variables.

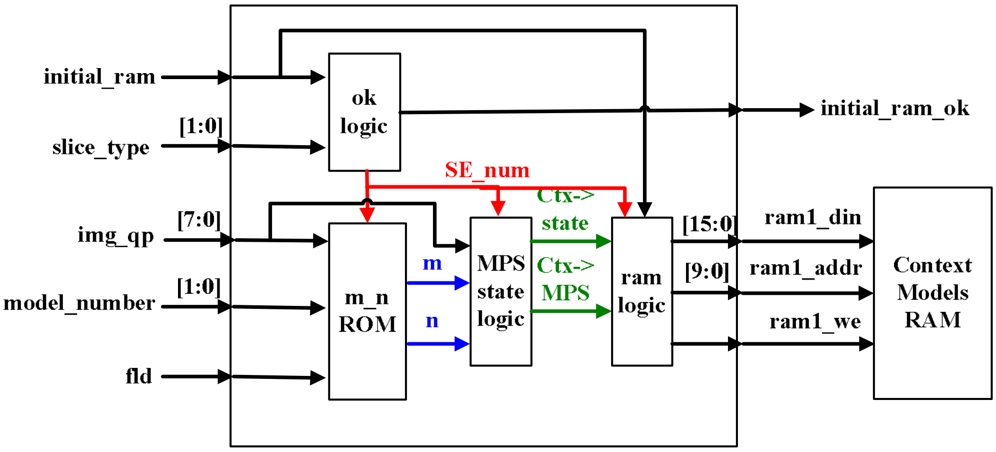


Fig 78. Interface of initialization for context variables

4.4.2. Architecture of Initialization for context variables

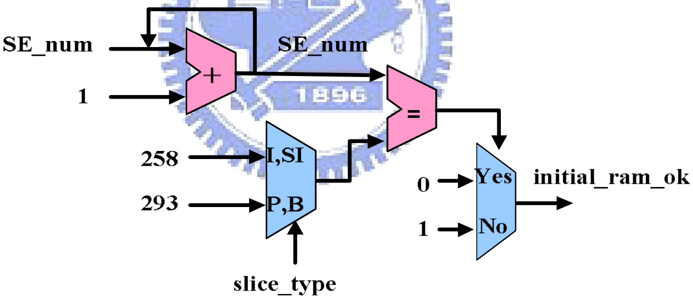


Fig 79. ok logic in Fig 78

The initial_ram_ok signal is generated by the logic shown in Fig 79.

For each context variable, the Ctx->state and Ctx->MPS are initialized, and SE_num will be increased one for each initialization. So if all of the Ctx->state and Ctx->MPS are initialized, initial_ram_ok will be set to 1.

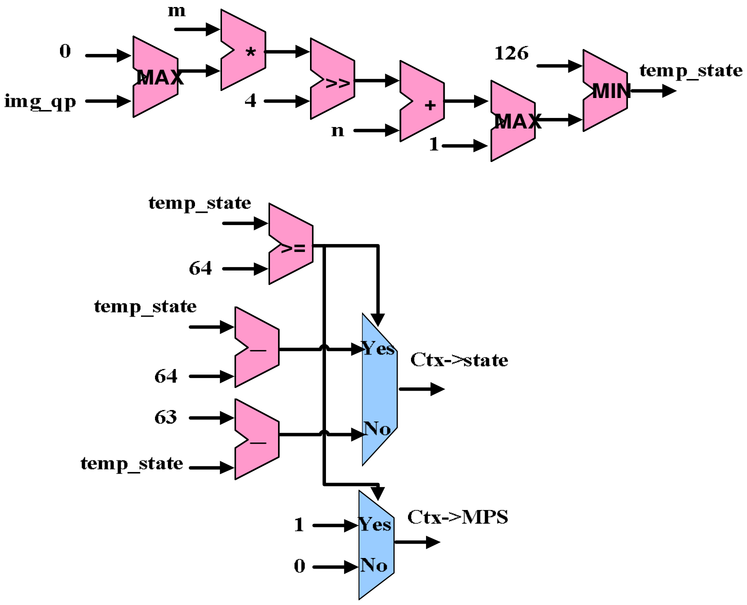


Fig 80. MPS state logic in Fig 78

The Ctx->state and Ctx->MPS signals are generated by the logic shown in Fig 80.

The two values assigned to Ctx->state and Ctx->MPS for the initialization are derived from img_qp, which is derived in equation 4.4.2.1 Given m and n,

*temp_state = min(max((((m * max(0 , img_qp)) >> 4) + n) , 1) , 126); eq.4.4.2.1*

And then the value of Ctx->state and Ctx->MPS are depending on equation 4.4.2.2.

Ctx->state = (temp_state >= 64) ? (temp_state – 64) : (63 – temp_state);

Ctx->MPS = (temp_state >= 64) ? 1 : 0; *eq.4.4.2.2*

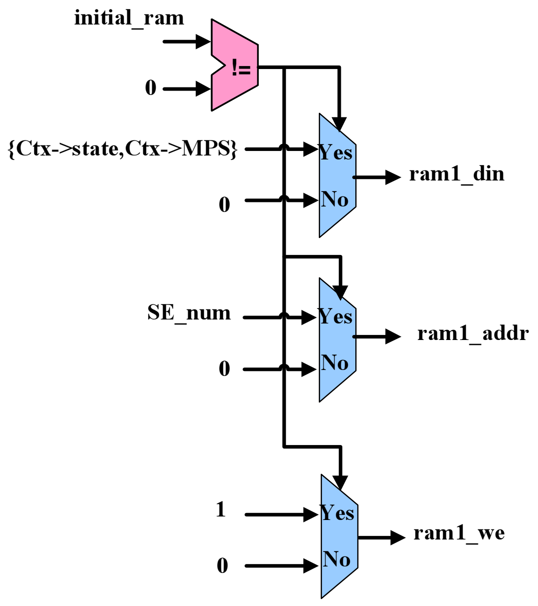


Fig 81. ram logic in Fig 78

The ram1_din, ram1_addr and ram1_we signals are generated by the logic shown in Fig 81.

We put Ctx->state and Ctx->MPS in the Context Models RAM, and the address of each Ctx->state and Ctx->MPS is SE_num.

4.5. State Controller for each Syntax Element Controller

The finite state machine of each syntax element controller is shown in Fig 82.

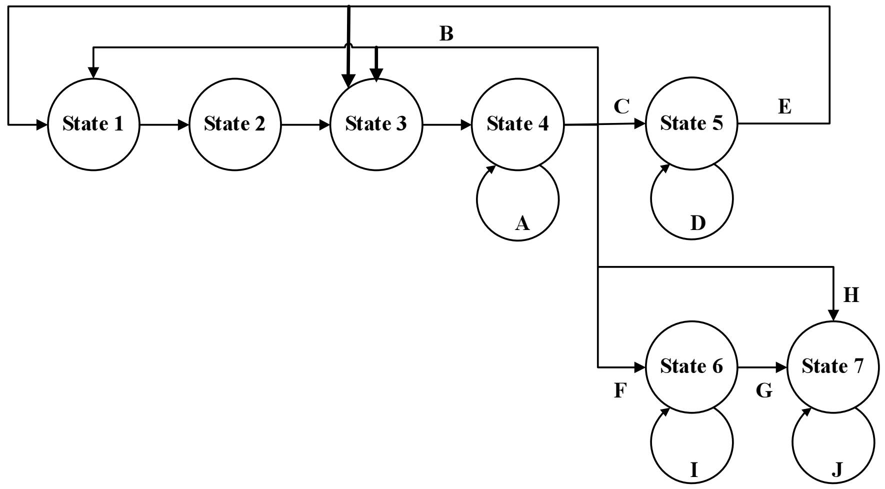


Fig 82. State controllers for each syntax element controller

We used 7 states to decode Bins of each syntax element in the corresponding syntax element controller. If we want to use bypass or final decoding process for binary arithmetic coding, we will start in State 3, because we don't need Ctx->state and Ctx->MPS for these decoding process. Otherwise, if we want to use regular decoding process for binary arithmetic coding, we will start in State 1.

In State 1, we wanted to get Ctx->state and Ctx->MPS from Context Models RAM, and we waited them in State 2, and then we get the values of them and we output some information, ex. AC_value, AC_Dbitsleft, AC_range, Enable_AC, to enable binary arithmetic coding in State 3. To go on, in State 4, when AC is finished decoding this bin, we get the updated value of Ctx->state and Ctx->MPS if we needed and get some information from binary arithmetic coding. If the value of signal Use_AC_bitstream_next_16bits from binary arithmetic coding is 1, we will transfer to State 5 or State 6 depending on whether we will continue to decode this syntax element or not. If yes, we will transfer to State 5, else we will transfer to State 6. Otherwise, If the value of signal Use_AC_bitstream_next_16bits from binary arithmetic coding is 0, we will transfer to State 1, State 3 or State 7 depending on whether we will continue to decode this syntax element or not. If yes, we will transfer to State 1 or State 3 depending on which decoding process of binary arithmetic coding to decode next bin, else we will transfer to State 7 to finish decoding this syntax element. When we in State 5 or State 6, we will wait updated value of AC_bitstream_next_16bits, and then we will transfer to State1, State3 or Stare 7.

'A' through 'J' are state transition conditions, which are described as follows:

- Condition A indicates "OK_AC_X is equal to 0". On the other words, if OK_AC_X is equal to 1, we will transfer to next state. Note: the X of OK_AC_X can be regular, bypass or final.

- Condition B indicates the value of Use_AC_bitstream_next_16bits is equal to 0 and we will continue decoding this syntax element.
- Condition C indicates the value of Use_AC_bitstream_next_16bits is equal to 1 and we will continue decoding this syntax element.
- Condition D and Condition I indicate waiting the updated value of AC_bitstream_next_16bits, if the value of AC_bitstream_next_16bits is updated, we will transfer to next state.
- Condition E and Condition G indicate the value of AC_bitstream_next_16bits is updated and we will transfer to next state.
- Condition F indicates the value of Use_AC_bitstream_next_16bits is equal to 1 and we will finish decoding this syntax element.
- Condition H indicates the value of Use_AC_bitstream_next_16bits is equal to 0 and we will finish decoding this syntax element.
- Condition J indicates "SE_OK is equal to 1". Other the other words, if we finished decoding this syntax element, we will hold in this state until new syntax element comes.

4.6. MB Skip Flag

The value of MB Skip Flag indicates whether the current macro-block in P or B slice is skipped or not.

4.6.1. Interface of MB Skip Flag

The interface of MB Skip Flag is shown in Fig 83, and the interface signals are explained as follows.

- a, b are each 1-bit input signal indicates the MB Skip Flag value of the left and top neighboring macro-block.

- value_in, range_in, and Dbitsleft_in are 25-bit, 9-bit and 5-bit input signals which connect to the input signals of top module or the output signals of binary arithmetic coding depending on the value of from_top_or_AC.
- which_SE is a 5-bit input signal indicates which syntax element is decoding at this time.
- slice_type is a 2-bit input signal indicates the type of the current slice which is decoded. There are four types of slice: I_slice, SI_slice, P_slice and B_slice..
- input_AC_bitstream_next_16bits_from_top_ok is a 1-bit input signal indicates the value of AC_bitstream_next_16bits is updated or not. If yes, the value of input_AC_bitstream_next_16bits_from_top_ok is equal to 1, otherwise, the value of input_AC_bitstream_next_16bits_from_top_ok is equal to 0.
- OK_AC_regular is a 1-bit signal indicates AC_Regular_mode of binary arthimetic coding is finished.
- state_in and MPS_in are 6-bit and 1-bit input signals which connect to the output signals of AC_Regular_mode of binary arthimetic coding.
- Bin_in and use_AC_bitstream_next_16bits_from_AC are 1-bit and 1-bit input signals which connect to the output signals of the binary arithmetic coding.
- ram1_dout is a 16-bit input signal indicates the Ctx->state and Ctx->MPS from the Context Models RAM.
- use_AC_bitstream_next_16bits_to_top is a 1-bit output signal indicates requesting the software to update the value of AC_bitstream_next_16bits.
- SE_OK is a 1-bit output signal indicates whether we have finished decoding this syntax element or not.

- value_out, range_out and Dbitsleft_out are 25-bit, 9-bit and 5-bit output signals which conncet to the output signals of top module and input signals of binary arithmetic coding.
- SE_value1 is 8-bit output signals indicates the value of the syntax element we decoded.
- enable_AC_regular is a 1-bit output signal to enable AC_Regular_mode of binary arthimetic coding.
- state_our and MPS_our are 6-bit and 1-bit output signals which connect to the input signals of AC_Regular_mode of binary arithmetic coding.
- ram1_din, ram1_addr and ram1_we are 16-bit, 10-bit and 1-bit output signals which connect to the input signals of Context Models RAM.
- from_top_or_AC is a 1-bit output signal to decide the value_in, range_in, and Dbitsleft_in input signals connect to the input signals of top module or output signals of binary arithmetic coding.
- last_dquant is a 8-bit output signal indicates the value of dquant.
- We use a 75-bit common_input signal to stand for {clk, rst, value_in, range_in, Dbitsleft_in, which_SE, slice_type, input_AC_bitstream_next_16bits_from_top_ok, OK_AC_regular, state_in, MPS_in, Bin_in, use_AC_bitstream_next_16bits_from_AC, ram1_dout},
- We use a 84-bit common_output signal to stand for {use_AC_bitstream_next_16bits_to_top , SE_OK, value_out, range_out, Dbitsleft_out, SE_value1, enable_AC_regular, state_out, MPS_out, ram1_din, ram1_addr, ram1_we}, and common_input and common_output signal will be used in all syntax element controllers.

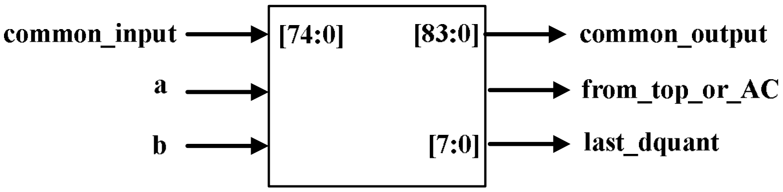


Fig 83. Interface of MB Skip Flag

4.6.2. Flow Chart of MB Skip Flag

The flow chart of MB Skip Flag is shown in Fig 84, and the binarization of MB Skip Flag is 1-bit FL coding.

The detail flow of arithmetic coding engine has described in 4.5 State Controller for each Syntax Element Controller.

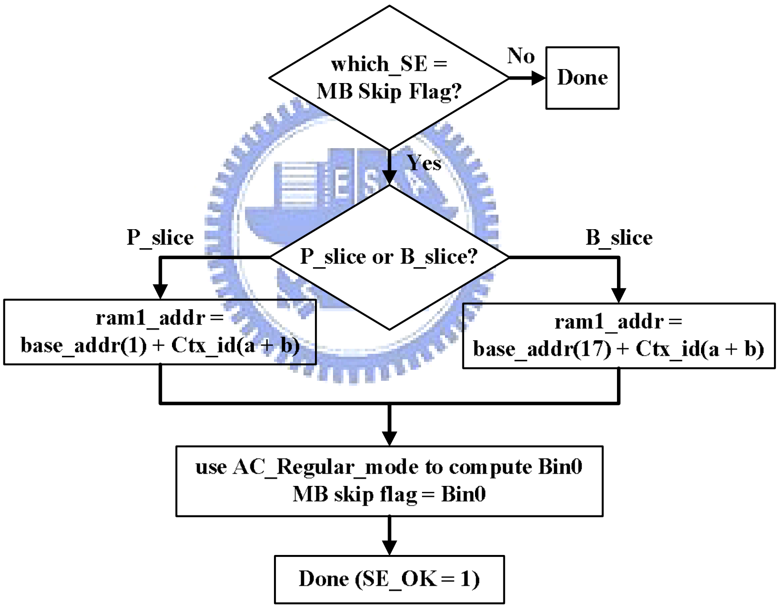


Fig 84. Flow chart of MB Skip Flag

4.7. MB Type

The MB Type will be decoded in each macro-block in I_slice and some macro-blocks in P_slice and B_slice if the macro-blocks are not skipped.

4.7.1. Interface of MB Type

The interface of MB Type is shown in Fig 85, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- a, b are each 1-bit input signal. For I_slice, if the MB Type of the left (or top) neighboring macro-block is not I4MB and I8MB, the value of a (or b) is equal to 1, otherwise, the value of a (or b) is equal to 0. For B_slice, if the MB Type of the left (or top) neighboring macro-block is not B_Direct_16x16, the value of a (or b) is equal to 1, otherwise, the value of a (or b) is equal to 0.
- OK_AC_final is a 1-bit signal indicates AC_Final_mode of binary arthimetic coding is finished.
- enable_AC_final is a 1-bit output signal to enable AC_Final_mode of binary arthimetic coding.
- from_top_or_ACr_or_ACf is a 2-bit output signal to decide the value_in, range_in, and Dbitsleft_in input signals which connect to the input signals of top module or output signals of AC_Regular_mode or AC_Final_mode of binary arithmetic coding.

ES

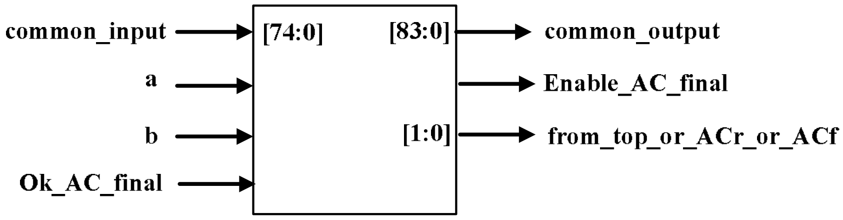
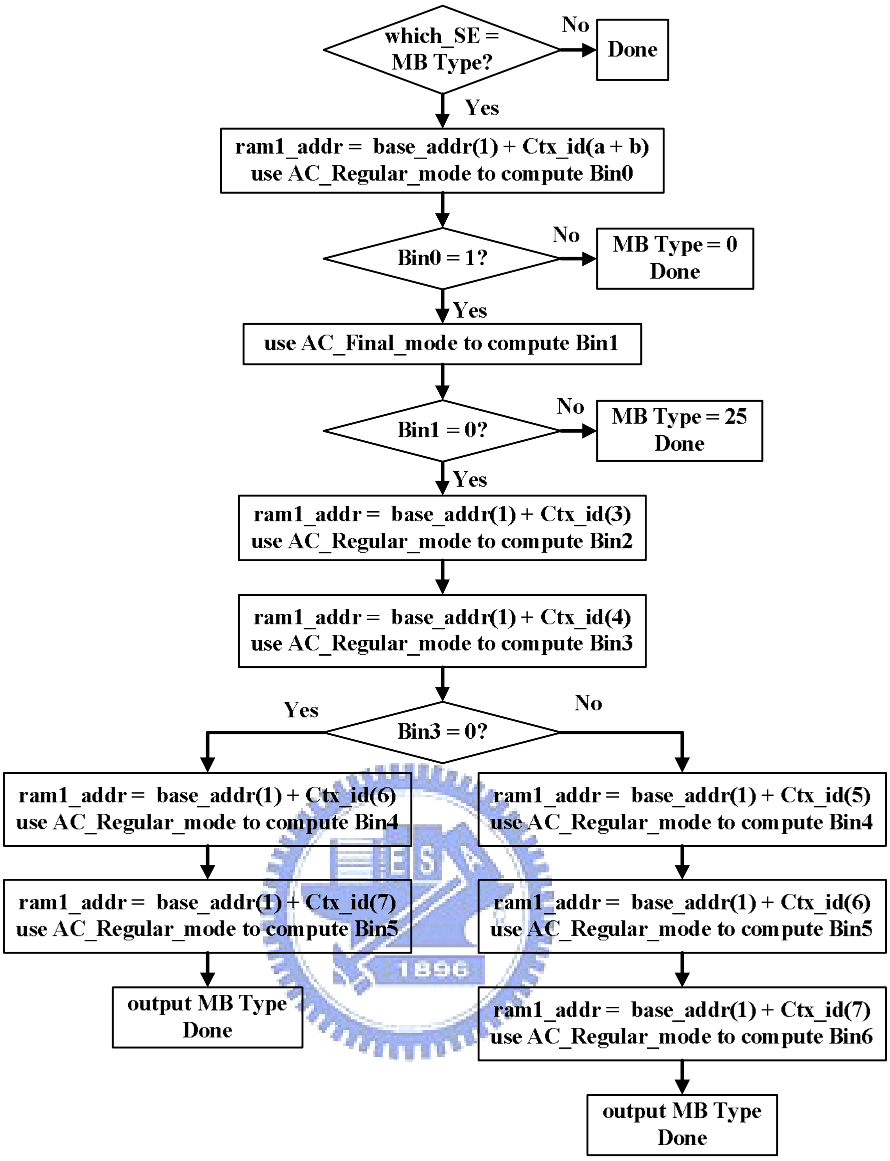


Fig 85. Interface of MB type

4.7.2. Flow Chart of MB Type in I_slice

The flow chart of MB Type in I_slice is shown in Fig 86, and the binarization of MB Type in I_slice is referring to Table 5.



which_SE = MB Type?
No
Done
Yes
ram1_addr = base_addr(1) + Ctx_id(a + b)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
MB Type = 0 Done
Yes
use AC_Final_mode to compute Bin1
Bin1 = 0?
No
MB Type = 25 Done
Yes
ram1_addr = base_addr(1) + Ctx_id(3)
use AC_Regular_mode to compute Bin2
ram1_addr = base_addr(1) + Ctx_id(4)
use AC_Regular_mode to compute Bin3
Bin3 = 0?
Yes
No
ram1_addr = base_addr(1) + Ctx_id(6)
use AC_Regular_mode to compute Bin4
ram1_addr = base_addr(1) + Ctx_id(7)
use AC_Regular_mode to compute Bin5
output MB Type Done
ram1_addr = base_addr(1) + Ctx_id(5)
use AC_Regular_mode to compute Bin4
ram1_addr = base_addr(1) + Ctx_id(6)
use AC_Regular_mode to compute Bin5
ram1_addr = base_addr(1) + Ctx_id(7)
use AC_Regular_mode to compute Bin6
output MB Type Done

Fig 86. Flow chart of MB Type in I_slice

Value (name) of MB Type	**Bin string**						
0 (I_4x4)	0						
1 (I_16x16_0_0_0)	1	0	0	0	0	0	
2 (I_16x16_1_0_0)	1	0	0	0	0	1	
3 (I_16x16_2_0_0)	1	0	0	0	1	0	
4 (I_16x16_3_0_0)	1	0	0	0	1	1	
5 (I_16x16_0_1_0)	1	0	0	1	0	0	0
6 (I_16x16_1_1_0)	1	0	0	1	0	0	1
7 (I_16x16_2_1_0)	1	0	0	1	0	1	0
8 (I_16x16_3_1_0)	1	0	0	1	0	1	1
9 (I_16x16_0_2_0)	1	0	0	1	1	0	0
10 (I_16x16_1_2_0)	1	0	0	1	1	0	1
11 (I_16x16_2_2_0)	1	0	0	1	1	1	0
12 (I_16x16_3_2_0)	1	0	0	1	1	1	1
13 (I_16x16_0_0_1)	1	0	1	0	0	0	
14 (I_16x16_1_0_1)	1	0	1	0	0	1	
15 (I_16x16_2_0_1)	1	0	1	0	1	0	
16 (I_16x16_3_0_1)	1	0	1	0	1	1	
17 (I_16x16_0_1_1)	1	0	1	1	0	0	0
18 (I_16x16_1_1_1)	1	0	1	1	0	0	1
19 (I_16x16_2_1_1)	1	0	1	1	0	1	0
20 (I_16x16_3_1_1)	1	0	1	1	0	1	1
21 (I_16x16_0_2_1)	1	0	1	1	1	0	0
22 (I_16x16_1_2_1)	1	0	1	1	1	0	1
23 (I_16x16_2_2_1)	1	0	1	1	1	1	0
24 (I_16x16_3_2_1)	1	0	1	1	1	1	1
25 (I_PCM)	1	1					
binIdx	0	1	2	3	4	5	6

Table 5. Binarization for MB Type in I_slice

4.7.3. Flow Chart of MB Type in P_slice

The flow chart of of MB Type in P_slice is shown in Fig 87, and the binarization of MB Type in P_slice is referring to Table 6.

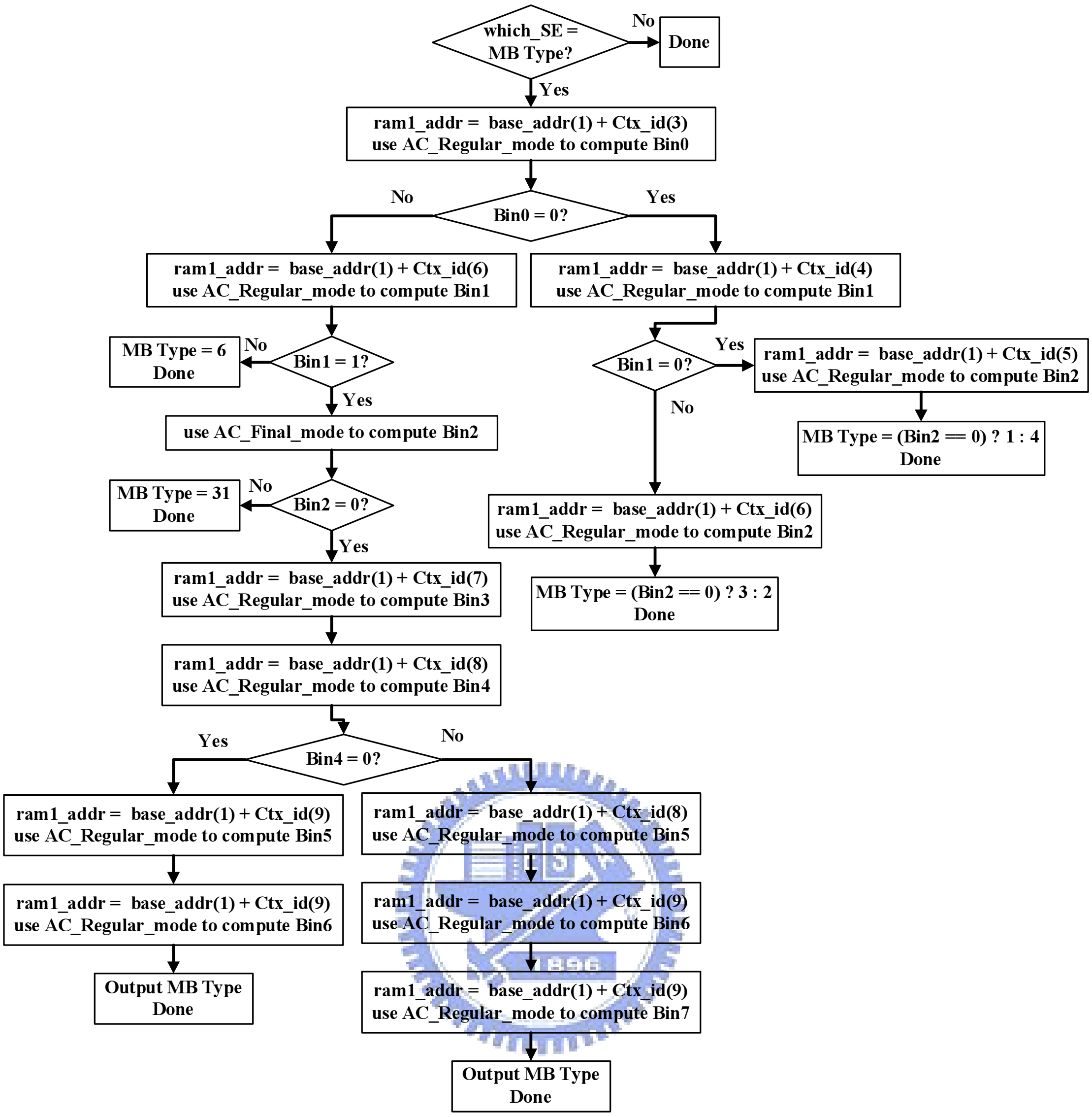

which_SE = MB Type?
No
Done
Yes
ram1_addr = base_addr(1) + Ctx_id(3)
use AC_Regular_mode to compute Bin0
No
Bin0 = 0?
Yes
ram1_addr = base_addr(1) + Ctx_id(6)
use AC_Regular_mode to compute Bin1
ram1_addr = base_addr(1) + Ctx_id(4)
use AC_Regular_mode to compute Bin1
MB Type = 6
Done
No
Bin1 = 1?
Yes
Bin1 = 0?
Yes
No
ram1_addr = base_addr(1) + Ctx_id(5)
use AC_Regular_mode to compute Bin2
MB Type = (Bin2 == 0) ? 1 : 4
Done
use AC_Final_mode to compute Bin2
MB Type = 31
Done
No
Bin2 = 0?
Yes
ram1_addr = base_addr(1) + Ctx_id(6)
use AC_Regular_mode to compute Bin2
MB Type = (Bin2 == 0) ? 3 : 2
Done
ram1_addr = base_addr(1) + Ctx_id(7)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(1) + Ctx_id(8)
use AC_Regular_mode to compute Bin4
Yes
Bin4 = 0?
No
ram1_addr = base_addr(1) + Ctx_id(9)
use AC_Regular_mode to compute Bin5
ram1_addr = base_addr(1) + Ctx_id(8)
use AC_Regular_mode to compute Bin5
ram1_addr = base_addr(1) + Ctx_id(9)
use AC_Regular_mode to compute Bin6
ram1_addr = base_addr(1) + Ctx_id(9)
use AC_Regular_mode to compute Bin6
Output MB Type
Done
ram1_addr = base_addr(1) + Ctx_id(9)
use AC_Regular_mode to compute Bin7
Output MB Type
Done

Fig 87. Flow chart of MB Type in P_slice

Slice type	Value (name) of mb_type	Bin string						
P, SP slice	0 (P_L0_16x16)	0	0	0				
	1 (P_L0_L0_16x8)	0	1	1				
	2 (P_L0_L0_8x16)	0	1	0				
	3 (P_8x8)	0	0	1				
	4 (P_8x8ref0)	na						
	5 to 30 (Intra, **prefix only**)	1						

Table 6. Binarization for MB Type in P_slice

4.7.4. Flow Chart of MB Type in B_slice

The flow chart of MB Type in B_slice is shown in Fig 88 and Fig 89, and the binarization of MB Type in B_slice is referring to Table 7.

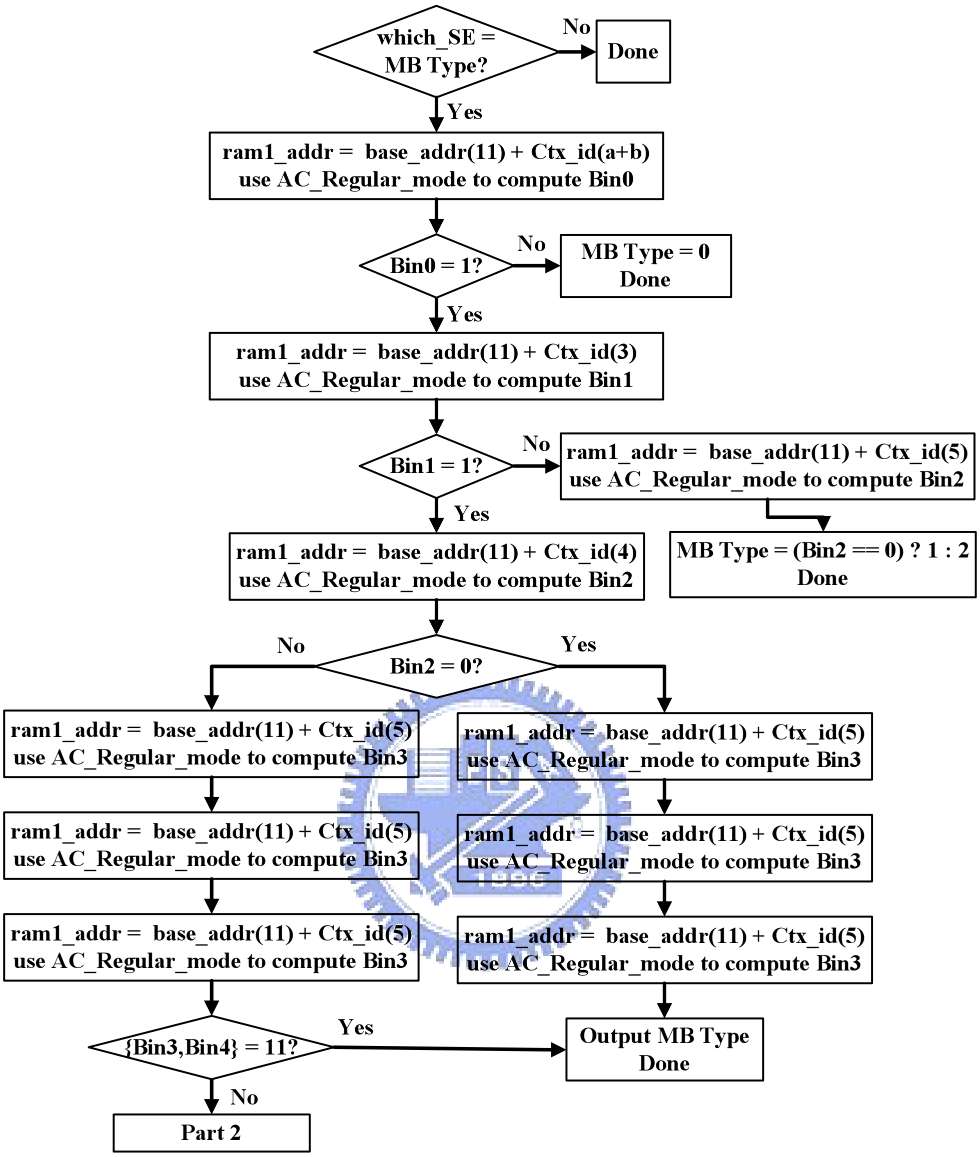
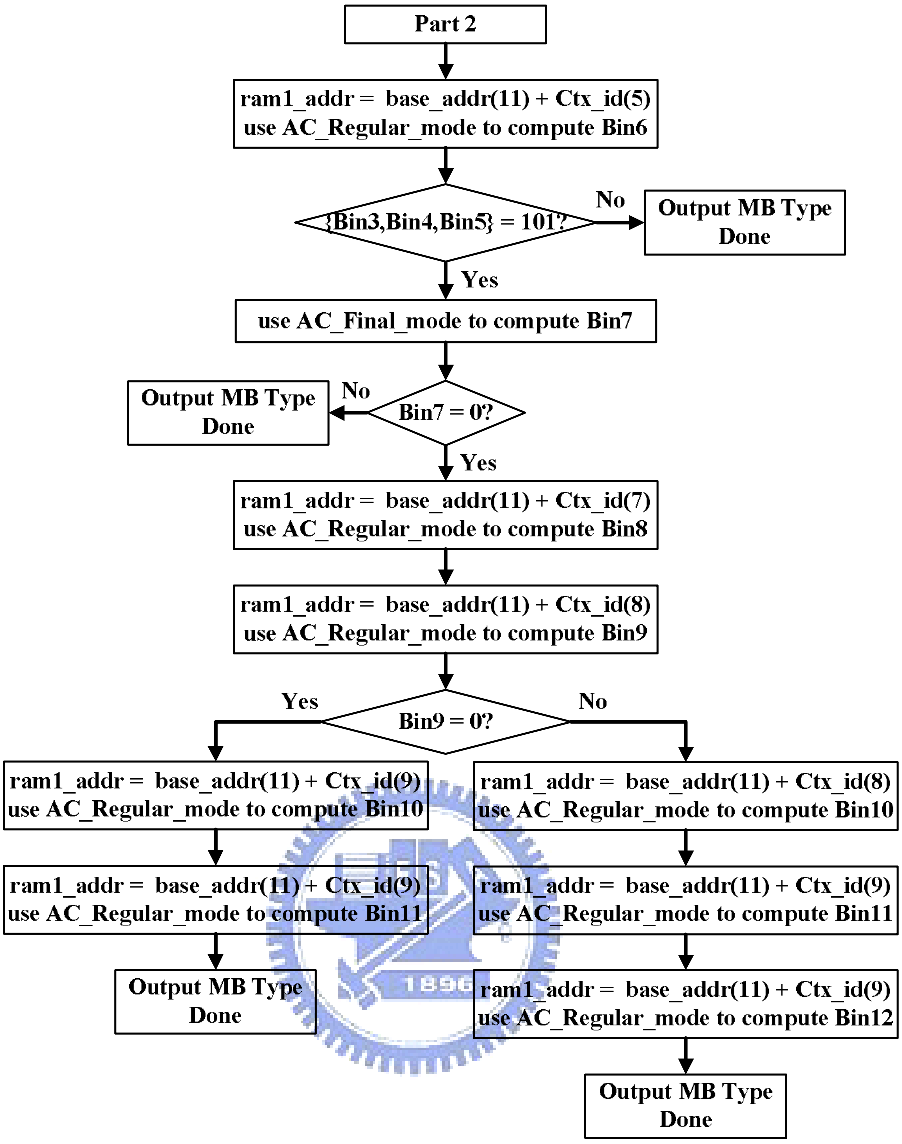

which_SE = MB Type?
No
Done
Yes
ram1_addr = base_addr(11) + Ctx_id(a+b)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
MB Type = 0
Done
Yes
ram1_addr = base_addr(11) + Ctx_id(3)
use AC_Regular_mode to compute Bin1
Bin1 = 1?
No
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin2
MB Type = (Bin2 == 0) ? 1 : 2
Done
Yes
ram1_addr = base_addr(11) + Ctx_id(4)
use AC_Regular_mode to compute Bin2
Bin2 = 0?
No
Yes
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin3
{Bin3,Bin4} = 11?
Yes
Output MB Type
Done
No
Part 2

Fig 88. Flow chart part1 of MB Type in B_slice



Part 2
ram1_addr = base_addr(11) + Ctx_id(5)
use AC_Regular_mode to compute Bin6
{Bin3,Bin4,Bin5} = 101?
No
Output MB Type Done
Yes
use AC_Final_mode to compute Bin7
Bin7 = 0?
No
Output MB Type Done
Yes
ram1_addr = base_addr(11) + Ctx_id(7)
use AC_Regular_mode to compute Bin8
ram1_addr = base_addr(11) + Ctx_id(8)
use AC_Regular_mode to compute Bin9
Bin9 = 0?
Yes
No
ram1_addr = base_addr(11) + Ctx_id(9)
use AC_Regular_mode to compute Bin10
ram1_addr = base_addr(11) + Ctx_id(9)
use AC_Regular_mode to compute Bin11
Output MB Type Done
ram1_addr = base_addr(11) + Ctx_id(8)
use AC_Regular_mode to compute Bin10
ram1_addr = base_addr(11) + Ctx_id(9)
use AC_Regular_mode to compute Bin11
ram1_addr = base_addr(11) + Ctx_id(9)
use AC_Regular_mode to compute Bin12
Output MB Type Done

Fig 89. Flow chart part2 of MB Type in B_slice

Slice type	Value (name) of mb_type	Bin string						
B slice	0 (B_Direct_16x16)	0						
	1 (B_L0_16x16)	1	0	0				
	2 (B_L1_16x16)	1	0	1				
	3 (B_Bi_16x16)	1	1	0	0	0	0	
	4 (B_L0_L0_16x8)	1	1	0	0	0	1	
	5 (B_L0_L0_8x16)	1	1	0	0	1	0	
	6 (B_L1_L1_16x8)	1	1	0	0	1	1	
	7 (B_L1_L1_8x16)	1	1	0	1	0	0	
	8 (B_L0_L1_16x8)	1	1	0	1	0	1	
	9 (B_L0_L1_8x16)	1	1	0	1	1	0	
	10 (B_L1_L0_16x8)	1	1	0	1	1	1	
	11 (B_L1_L0_8x16)	1	1	1	1	1	0	
	12 (B_L0_Bi_16x8)	1	1	1	0	0	0	0
	13 (B_L0_Bi_8x16)	1	1	1	0	0	0	1
	14 (B_L1_Bi_16x8)	1	1	1	0	0	1	0
	15 (B_L1_Bi_8x16)	1	1	1	0	0	1	1
	16 (B_Bi_L0_16x8)	1	1	1	0	1	0	0
	17 (B_Bi_L0_8x16)	1	1	1	0	1	0	1
	18 (B_Bi_L1_16x8)	1	1	1	0	1	1	0
	19 (B_Bi_L1_8x16)	1	1	1	0	1	1	1
	20 (B_Bi_Bi_16x8)	1	1	1	1	0	0	0
	21 (B_Bi_Bi_8x16)	1	1	1	1	0	0	1
	22 (B_8x8)	1	1	1	1	1	1	
	23 to 48 (Intra, prefix only)	1	1	1	1	0	1	
binIdx		0	1	2	3	4	5	6

Table 7. Binarization for MB Type in B_slice

4.8. Sub MB Type

If the macro-block is not skipped, and 8x8 sub-macro-block of this macro-block coded in P_8x8 or B_8x8 mode, an additional syntax element, Sub MB Type is present that indicating the type of the corresponding block.

4.8.1. Interface of Sub MB Type

The interface of Sub MB Type is shown in Fig 90, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.

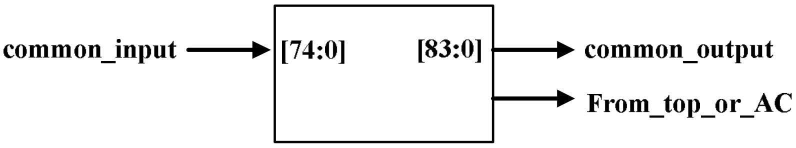


Fig 90. Interface of Sub MB Type

4.8.2. Flow Chart of Sub MB Type in P_slice

The flow chart of Sub MB Type in P_slice is shown in Fig 91, and the binarization of Sub MB Type in P_slice is referring to Table 8.

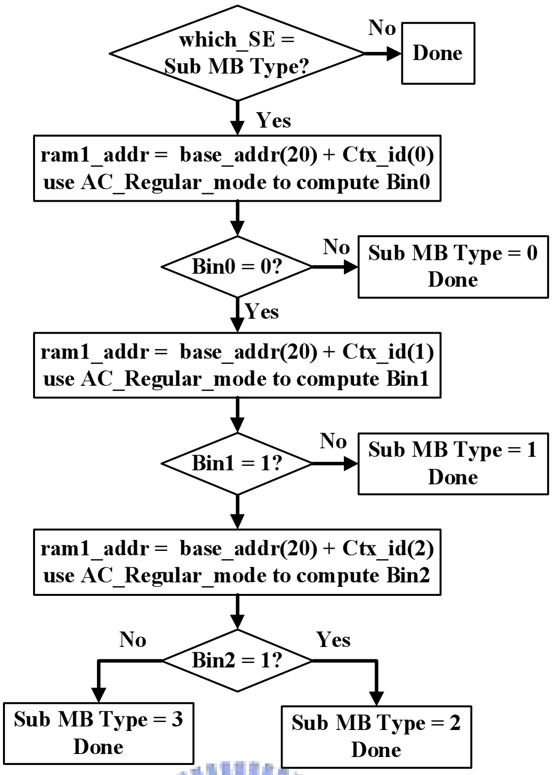

which_SE =
Sub MB Type?
No
Done
Yes
ram1_addr = base_addr(20) + Ctx_id(0)
use AC_Regular_mode to compute Bin0
Bin0 = 0?
No
Sub MB Type = 0
Done
Yes
ram1_addr = base_addr(20) + Ctx_id(1)
use AC_Regular_mode to compute Bin1
Bin1 = 1?
No
Sub MB Type = 1
Done
ram1_addr = base_addr(20) + Ctx_id(2)
use AC_Regular_mode to compute Bin2
No
Bin2 = 1?
Yes
Sub MB Type = 3
Done
Sub MB Type = 2
Done

Fig 91. Flow chart of Sub MB Type in P_slice

Slice type	Value (name) of sub_mb_type	Bin string					
P, SP slice	0 (P_L0_8x8)	1					
	1 (P_L0_8x4)	0	0				
	2 (P_L0_4x8)	0	1	1			
	3 (P_L0_4x4)	0	1	0			

Table 8. Binarization for Sub MB Type in P_slice

4.8.3. Flow Chart of Sub MB Type in B_slice

The flow chart of Sub MB Type in B_slice is shown in Fig 92, and the binarization of Sub MB Type in B_slice is referring to Table 9.

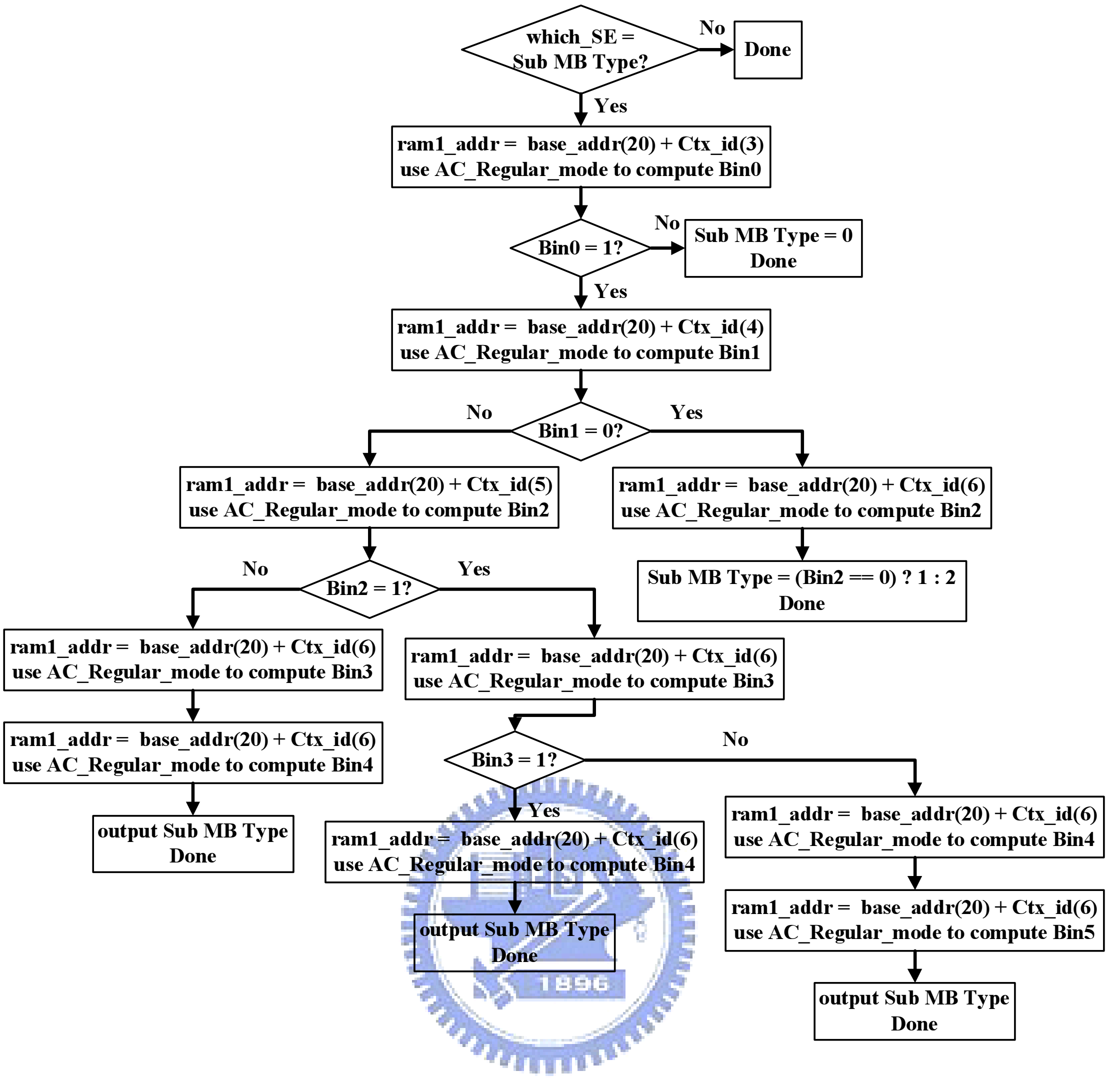

which_SE =
Sub MB Type?
No
Done
Yes
ram1_addr = base_addr(20) + Ctx_id(3)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
Sub MB Type = 0
Done
Yes
ram1_addr = base_addr(20) + Ctx_id(4)
use AC_Regular_mode to compute Bin1
No
Bin1 = 0?
Yes
ram1_addr = base_addr(20) + Ctx_id(5)
use AC_Regular_mode to compute Bin2
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin2
No
Bin2 = 1?
Yes
Sub MB Type = (Bin2 == 0) ? 1 : 2
Done
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin3
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin4
Bin3 = 1?
No
output Sub MB Type
Done
Yes
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin4
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin4
output Sub MB Type
Done
ram1_addr = base_addr(20) + Ctx_id(6)
use AC_Regular_mode to compute Bin5
output Sub MB Type
Done

Fig 92. Flow chart of Sub MB Type in B_slice

Slice type	Value (name) of sub_mb_type	Bin string					
B slice	0 (B_Direct_8x8)	0					
	1 (B_L0_8x8)	1	0	0			
	2 (B_L1_8x8)	1	0	1			
	3 (B_Bi_8x8)	1	1	0	0	0	
	4 (B_L0_8x4)	1	1	0	0	1	
	5 (B_L0_4x8)	1	1	0	1	0	
	6 (B_L1_8x4)	1	1	0	1	1	
	7 (B_L1_4x8)	1	1	1	0	0	0
	8 (B_Bi_8x4)	1	1	1	0	0	1
	9 (B_Bi_4x8)	1	1	1	0	1	0
	10 (B_L0_4x4)	1	1	1	0	1	1
	11 (B_L1_4x4)	1	1	1	1	0	
	12 (B_Bi_4x4)	1	1	1	1	1	
binIdx		0	1	2	3	4	5

Table 9. Binarization for Sub MB Type in B_slice

4.9. Intra Prediction Mode for Luma4x4

The luminance intra prediction modes for 4x4 blocks are itself predicted resulting in the syntax element of the value of prev_intra4x4_pred_mode_flag and the mode indicator rem_intra4x4_pred_mode, where the rem_intra4x4_pred_mode is only presented if prev_intra_pred4x4_mode_flag is equal to 0.

4.9.1. Interface of Intra Prediction Mode for Luma4x4

The interface of Intra Prediction Mode for Luma4x4 is shown in Fig 93, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.

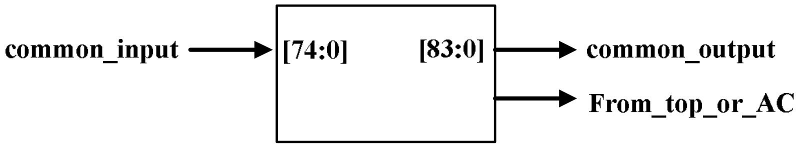


Fig 93. Interface of Intra Prediction Mode for Luma4x4

4.9.2. Flow Chart of Intra Prediction Mode for Luma4x4

The flow chart of Intra Prediction Mode for Luma4x4 is shown in Fig 94, and the binarization of prev_intra4x4_pred_mode_flag is 1-bit FL coding, the binarization of rem_intra4x4_pred_mode is 3-bit FL coding.

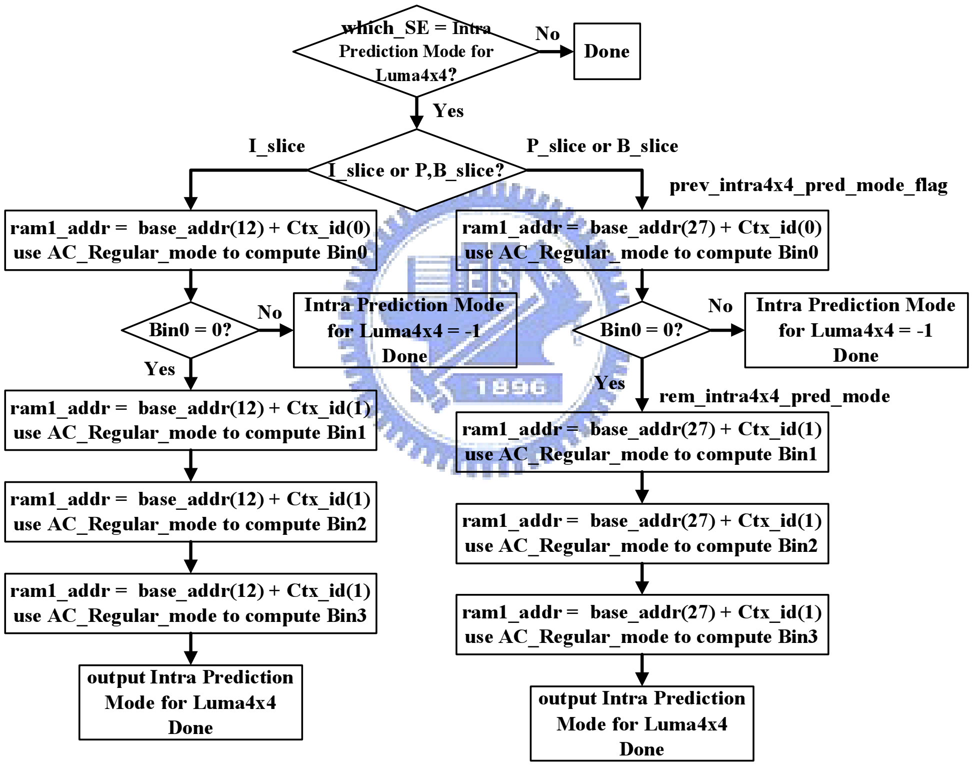

which_SE = Intra Prediction Mode for Luma4x4?
No
Done
Yes
I_slice
P_slice or B_slice
I_slice or P,B_slice?
prev_intra4x4_pred_mode_flag
ram1_addr = base_addr(12) + Ctx_id(0) use AC_Regular_mode to compute Bin0
ram1_addr = base_addr(27) + Ctx_id(0) use AC_Regular_mode to compute Bin0
Bin0 = 0?
No
Intra Prediction Mode for Luma4x4 = -1 Done
Yes
Bin0 = 0?
No
Intra Prediction Mode for Luma4x4 = -1 Done
Yes
rem_intra4x4_pred_mode
ram1_addr = base_addr(12) + Ctx_id(1) use AC_Regular_mode to compute Bin1
ram1_addr = base_addr(12) + Ctx_id(1) use AC_Regular_mode to compute Bin2
ram1_addr = base_addr(12) + Ctx_id(1) use AC_Regular_mode to compute Bin3
output Intra Prediction Mode for Luma4x4 Done
ram1_addr = base_addr(27) + Ctx_id(1) use AC_Regular_mode to compute Bin1
ram1_addr = base_addr(27) + Ctx_id(1) use AC_Regular_mode to compute Bin2
ram1_addr = base_addr(27) + Ctx_id(1) use AC_Regular_mode to compute Bin3
output Intra Prediction Mode for Luma4x4 Done

Fig 94. Flow chart of Intra Prediction Mode for Luma4x4

4.10. Intra Prediction Mode for Chroma

Intra Prediction Mode for Chroma specifies the type of spatial prediction used for chrmoa in macro-blocks using Intra_4x4 or Intra_16x16 prediction.

4.10.1. Interface of Intra Prediction Mode for Chroma

The interface of Intra Prediction Mode for Chroma is shown in Fig 95, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- a, b are each 1-bit input signal. If the mb_type of the left (or top) neighboring macro-block is not equal to IPCM and the Intra_Prediction_Mode_for_Chroma of the left (or top) neighboring macro-block is not equal to 0, the value of a (or b) is equal to 1, otherwise, the value of a (or b) is equal to 0.
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.

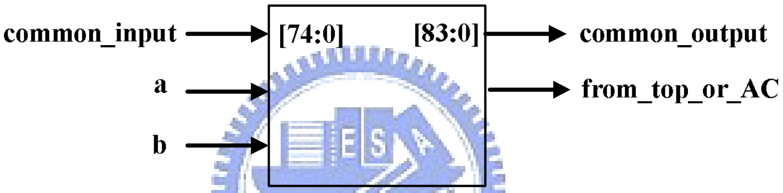


Fig 95. Interface of Intra Prediction Mode for Chroma

4.10.2. Flow Chart of Intra Prediction Mode for Chroma

The flow chart of Intra Prediction Mode for Chroma is shown in Fig 96, and the binarization of Intra Prediction Mode for Chroma is variable-length coding, but the maximum length of Intra Prediction Mode for Chroma is 3.

1896

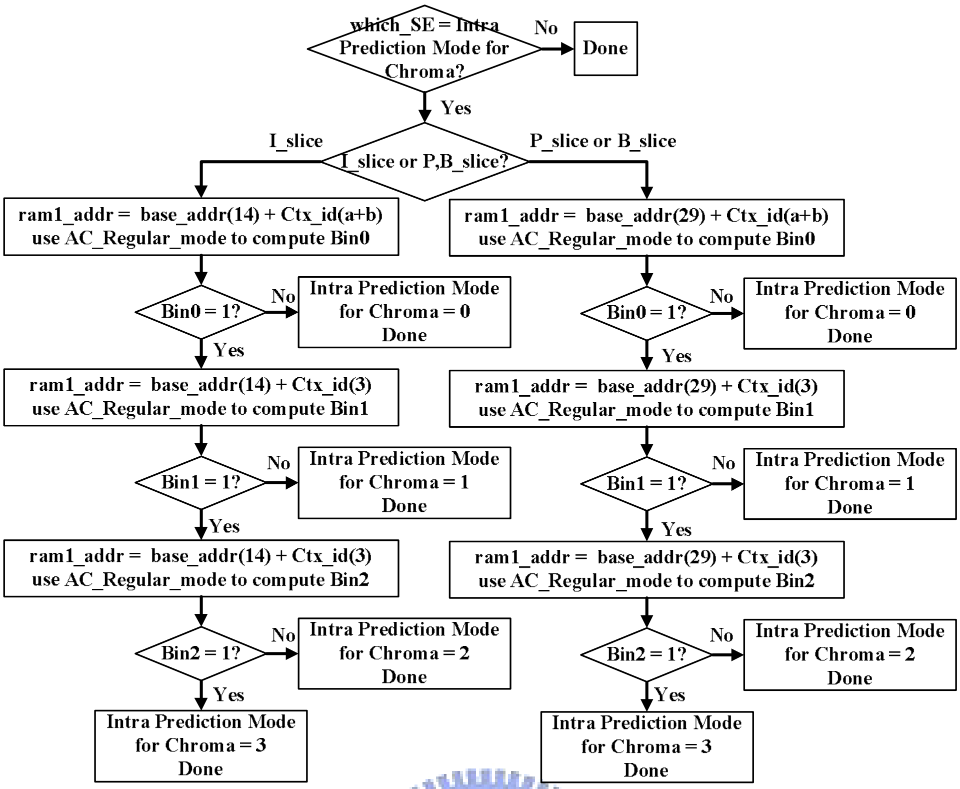

which_SE = Intra Prediction Mode for Chroma?
No
Done
Yes
I_slice
P_slice or B_slice
I_slice or P,B_slice?
ram1_addr = base_addr(14) + Ctx_id(a+b) use AC_Regular_mode to compute Bin0
ram1_addr = base_addr(29) + Ctx_id(a+b) use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
Intra Prediction Mode for Chroma = 0 Done
Yes
Bin0 = 1?
No
Intra Prediction Mode for Chroma = 0 Done
Yes
ram1_addr = base_addr(14) + Ctx_id(3) use AC_Regular_mode to compute Bin1
ram1_addr = base_addr(29) + Ctx_id(3) use AC_Regular_mode to compute Bin1
Bin1 = 1?
No
Intra Prediction Mode for Chroma = 1 Done
Yes
Bin1 = 1?
No
Intra Prediction Mode for Chroma = 1 Done
Yes
ram1_addr = base_addr(14) + Ctx_id(3) use AC_Regular_mode to compute Bin2
ram1_addr = base_addr(29) + Ctx_id(3) use AC_Regular_mode to compute Bin2
Bin2 = 1?
No
Intra Prediction Mode for Chroma = 2 Done
Yes
Bin2 = 1?
No
Intra Prediction Mode for Chroma = 2 Done
Yes
Intra Prediction Mode for Chroma = 3 Done
Intra Prediction Mode for Chroma = 3 Done

Fig 96. Flow chart of Intra Prediction Mode for Chroma

4.11. Reference Frame Index

Reference Frame Index when present, specifies the index in reference picture list0 or reference picture list1 of the reference picture to be used for prediction.

4.11.1. Interface of Reference Frame Index

The interface of Reference Frame Index is shown in Fig 97, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- a, b are each 1-bit input signal. If the RefIdxZeroFlag of the neighboring macro-block or left (or top) sub-macro-block partitions of the current partition is not equal to 0, the value of a (or b) is equal to 0, otherwise, the value of a (or b) is equal to 1. RefIdxZeroFlag us to stand for whether

Reference Frame Index with value 0 is chosen for the corresponding partition.

- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.

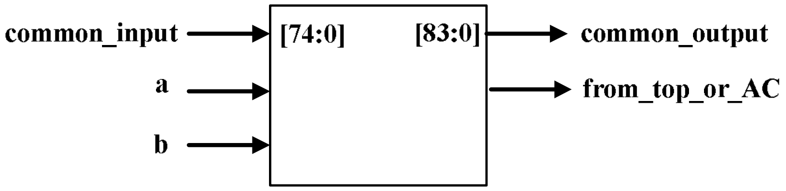


Fig 97. Interface of Reference Frame Index

4.11.2. Flow Chart of Reference Frame Index

The flow chart of Reference Frame Index is shown in Fig 98. and the binarization of Reference Frame Index is unary binarization.

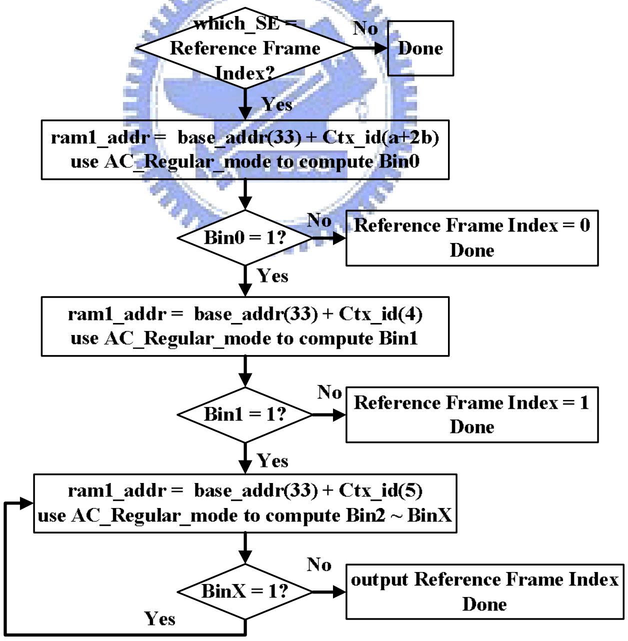

which_SE = Reference Frame Index?
No
Done
Yes
ram1_addr = base_addr(33) + Ctx_id(a+2b)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
Reference Frame Index = 0
Done
Yes
ram1_addr = base_addr(33) + Ctx_id(4)
use AC_Regular_mode to compute Bin1
Bin1 = 1?
No
Reference Frame Index = 1
Done
Yes
ram1_addr = base_addr(33) + Ctx_id(5)
use AC_Regular_mode to compute Bin2 ~ BinX
BinX = 1?
No
output Reference Frame Index
Done
Yes

Fig 98. Flow chart of Reference Frame Index

4.12. Motion Vector Difference

Motion Vector Differences specifies the difference between a vevtor component to be used and its prediction. The horizontal motion vector component difference is decoded first in decoding order, and the vertical motion vector component difference is decoded second in decoding order,

4.12.1. Interface of Motion Vector Difference

The interface of Motion Vector Difference is shown in Fig 99, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- a, b are each 8-bit input signal indicates the value of motion vector difference component for the left and top macro-block or sub-macro-block partition of the current macro-block or sub-macro-block partition.
- OK_AC_bypass is a 1-bit input signal indicates AC_Bypass_mode of binary arthimetic coding is finished.
- MVD_component is a 1-bit input signal indicates the component of motion vector difference is horizontal or vertical.
- Enable_AC_bypass is a 1-bit output signal to enable AC_Bypass_mode of binary arthimetic coding.
- from_top_or_ACr_or_ACb is a 2-bit output signal to decide the value_in, range_in, and Dbitsleft_in input signals which connect to the input signals of top module or output signals of AC_Regular_mode or AC_Bypass_mode of binary arithmetic coding.

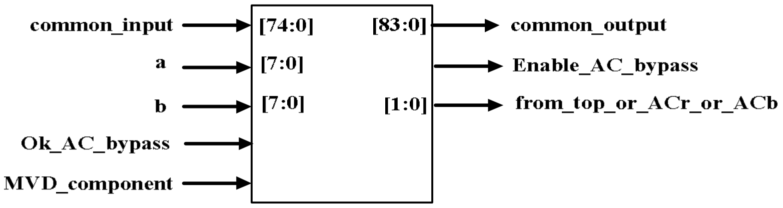
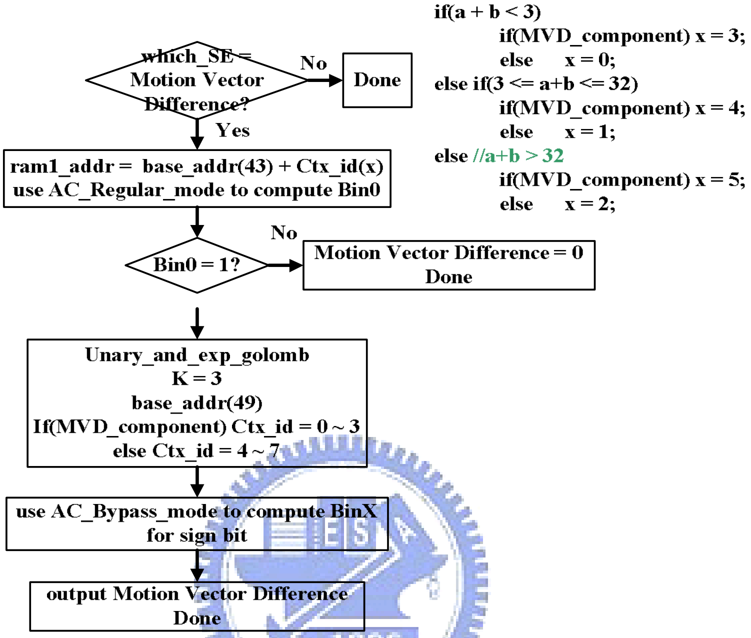


Fig 99. Interface of Motion Vector Difference

4.12.2. Flow chart of Motion Vector Difference

The flow chart of Motion Vector Difference is shown in Fig 100, and the binarization of Motion Vector Difference is UEG3 binarization scheme with a cutoff value of 9.



which_SE = Motion Vector Difference?
No
Done
Yes
ram1_addr = base_addr(43) + Ctx_id(x)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
Motion Vector Difference = 0
Done
Unary_and_exp_golomb
K = 3
base_addr(49)
If(MVD_component) Ctx_id = 0 ~ 3
else Ctx_id = 4 ~ 7
use AC_Bypass_mode to compute BinX
for sign bit
output Motion Vector Difference
Done

```
if(a + b < 3)
        if(MVD_component) x = 3;
        else     x = 0;
else if(3 <= a+b <= 32)
        if(MVD_component) x = 4;
        else     x = 1;
else //a+b > 32
        if(MVD_component) x = 5;
        else     x = 2;
```

Fig 100. Flow chart of Motion Vector Difference

4.13. Coded Block Pattern

For each non-skipped macro-block with prediction mode not equal to intra_16x16, the coded block pattern symbol indicates which of the six 8x8 blocks – four for luminance and two for chrominance – to decide whether the each 8x8 block contains nonzero transform coefficients or not.

4.13.1. Interface of Coded Block Pattern

The interface of Coded Block Pattern is shown in Fig 101, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.

- a, b are each 1-bit input signal indicates whether the bit of the coded block pattern corresponding to the 8x8 blocks to the left and top of the current block is equal to 0 or not. If it is equal to 0, a and b are equal to 1, otherwise, a and b are equal to 0
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.

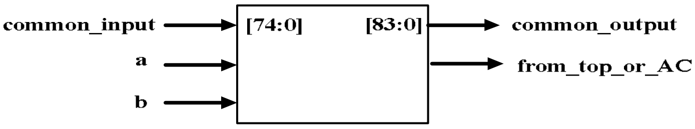


Fig 101. Interface of Coded Block Pattern

4.13.2. Flow Chart of Coded Block Pattern

The flow chart of Coded Block Pattern is shown in Fig 102, and the binarization of Coded Block Pattern is the concatenation of a 4-bit FL coding and a TU binarization wuth cutoff value S = 2.

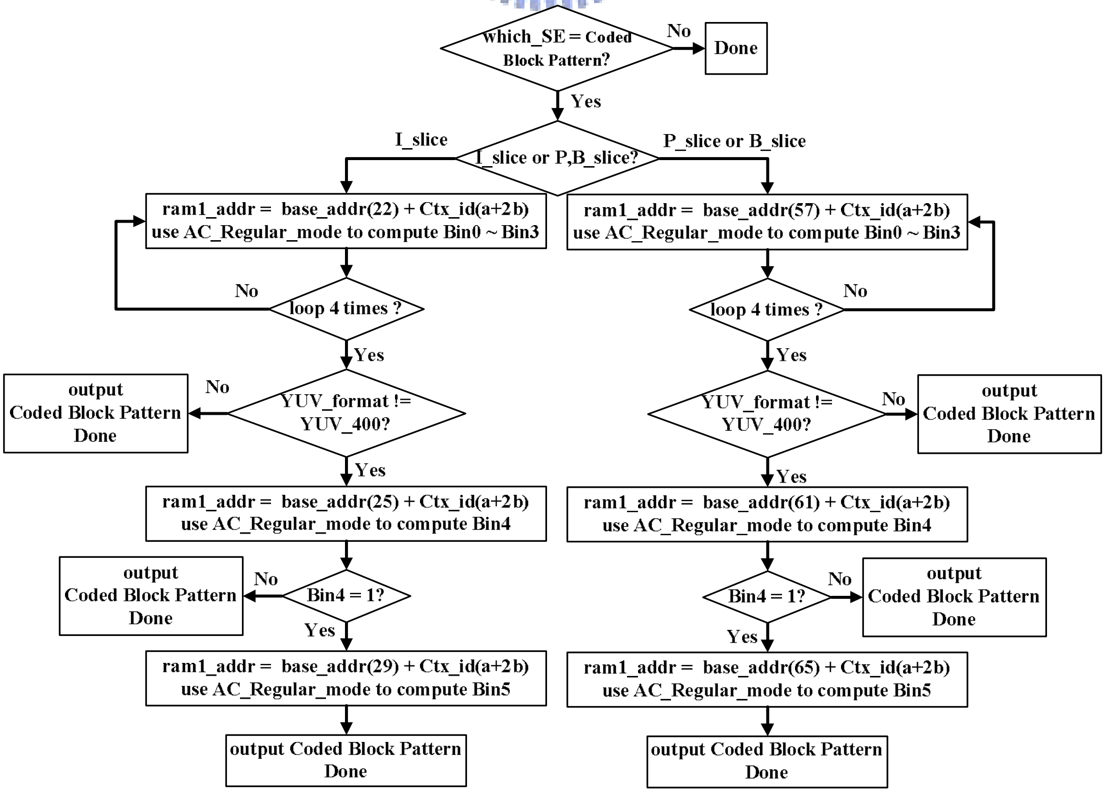

which_SE = Coded Block Pattern?
No
Done
Yes
I_slice
I_slice or P,B_slice?
P_slice or B_slice
ram1_addr = base_addr(22) + Ctx_id(a+2b) use AC_Regular_mode to compute Bin0 ~ Bin3
ram1_addr = base_addr(57) + Ctx_id(a+2b) use AC_Regular_mode to compute Bin0 ~ Bin3
No
loop 4 times ?
loop 4 times ?
No
Yes
Yes
output Coded Block Pattern Done
No
YUV_format != YUV_400?
YUV_format != YUV_400?
No
output Coded Block Pattern Done
Yes
Yes
ram1_addr = base_addr(25) + Ctx_id(a+2b) use AC_Regular_mode to compute Bin4
ram1_addr = base_addr(61) + Ctx_id(a+2b) use AC_Regular_mode to compute Bin4
output Coded Block Pattern Done
No
Bin4 = 1?
Bin4 = 1?
No
output Coded Block Pattern Done
Yes
Yes
ram1_addr = base_addr(29) + Ctx_id(a+2b) use AC_Regular_mode to compute Bin5
ram1_addr = base_addr(65) + Ctx_id(a+2b) use AC_Regular_mode to compute Bin5
output Coded Block Pattern Done
output Coded Block Pattern Done

Fig 102. Flow chart of Coded Block Pattern

4.14. MB Based Quantization Parameter

MB Based Quantization Parameter is present for each non-skipped macro-block with a value of Coded Block Pattern not equal to 0.

4.14.1. Interface of MB Based Quantization Parameter

The interface of MB Based Quantization Parameter is shown in Fig 103, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- dquant_in is a 8-bit input signal which indicates the value of mb_qp_delta for the preceding macro-block of current macro-block in decoding order
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.
- dquant_out is a 8-bit output signal which indicates the value of mb_qp_delta to using in next macro-block in decoding order

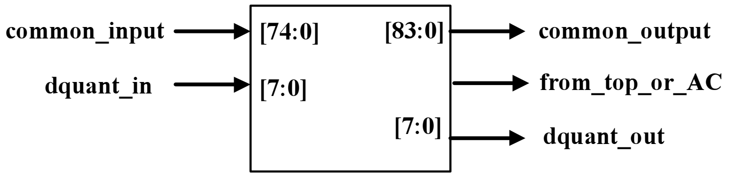


Fig 103. Interface of MB Based Quantization Parameter

4.14.2. Flow Chart of MB Based Quantization Parameter

The flow chart of MB Based Quantization Parameter is shown in Fig 104, and the binarization of MB Based Quantization Parameter is unary binarization.

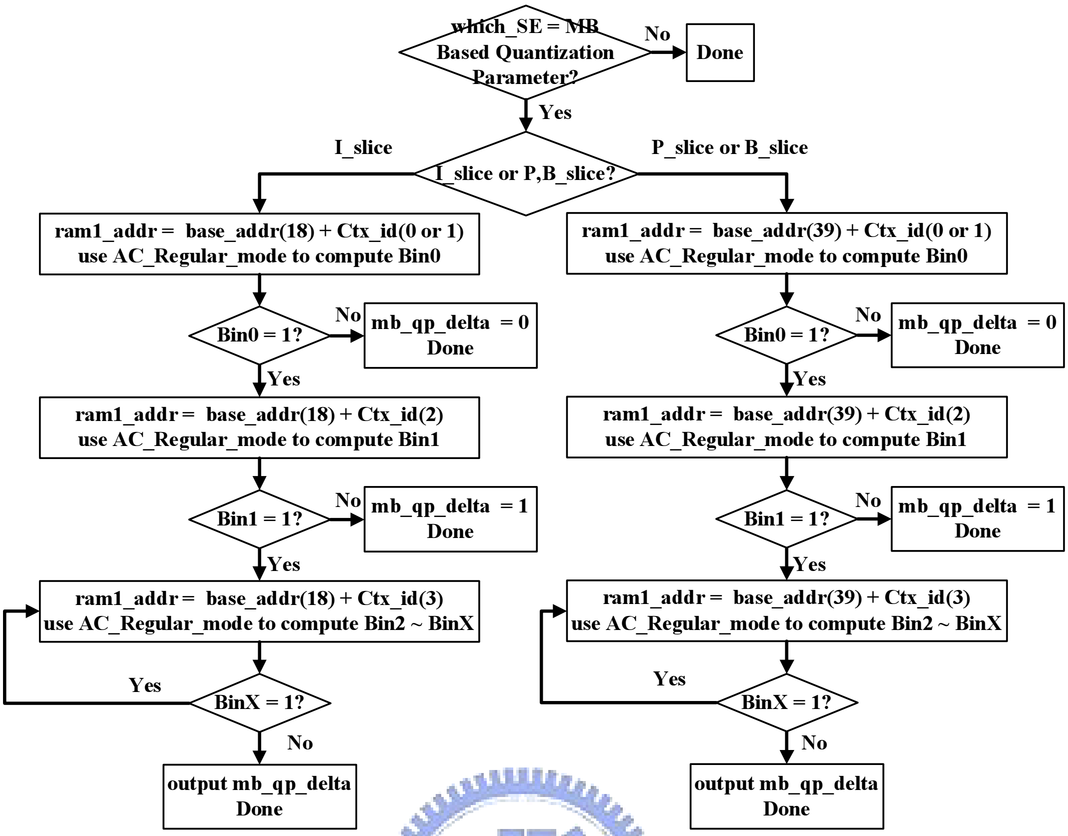

which_SE = MB Based Quantization Parameter?
No
Done
Yes
I_slice
P_slice or B_slice
I_slice or P,B_slice?
ram1_addr = base_addr(18) + Ctx_id(0 or 1) use AC_Regular_mode to compute Bin0
ram1_addr = base_addr(39) + Ctx_id(0 or 1) use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
mb_qp_delta = 0 Done
Yes
ram1_addr = base_addr(18) + Ctx_id(2) use AC_Regular_mode to compute Bin1
ram1_addr = base_addr(39) + Ctx_id(2) use AC_Regular_mode to compute Bin1
Bin1 = 1?
No
mb_qp_delta = 1 Done
Yes
ram1_addr = base_addr(18) + Ctx_id(3) use AC_Regular_mode to compute Bin2 ~ BinX
ram1_addr = base_addr(39) + Ctx_id(3) use AC_Regular_mode to compute Bin2 ~ BinX
Yes
BinX = 1?
No
output mb_qp_delta Done

Fig 104. Flow chart of MB Based Quantization Parameter

4.15. Coded Block Flag

Coded Block Flag specifies whether the block contains non-zero transform coefficient levels or not.

4.15.1. Interface of Coded Block Flag

The interface of Coded Block Flag is shown in Fig 105, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- a and b are each 1-bit input signal indicating the value of Coded Block Flag to the left and top blocks of the current block. Only blocks of the same type are used for context determination. There are five different types of blocks.

If no neighboring block of the same type exists, the value of a or b is equal to 0. If a neighboring block is outside the picture area or positioned in a different slice, the value of a or b is equal to default value. If the current block is coded using an intra prediction mode, the default value is equal to 1, otherwise, the default value is equal to 0.

- typetoctx_bcbp is a 3-bit input signal indicating the different types of blocks corresponding to adaptive context model.
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.

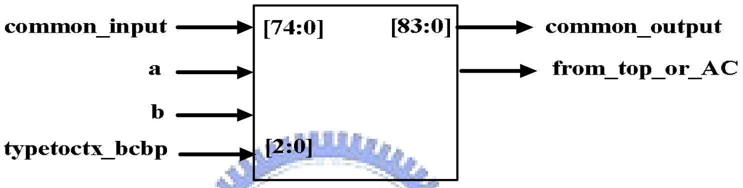
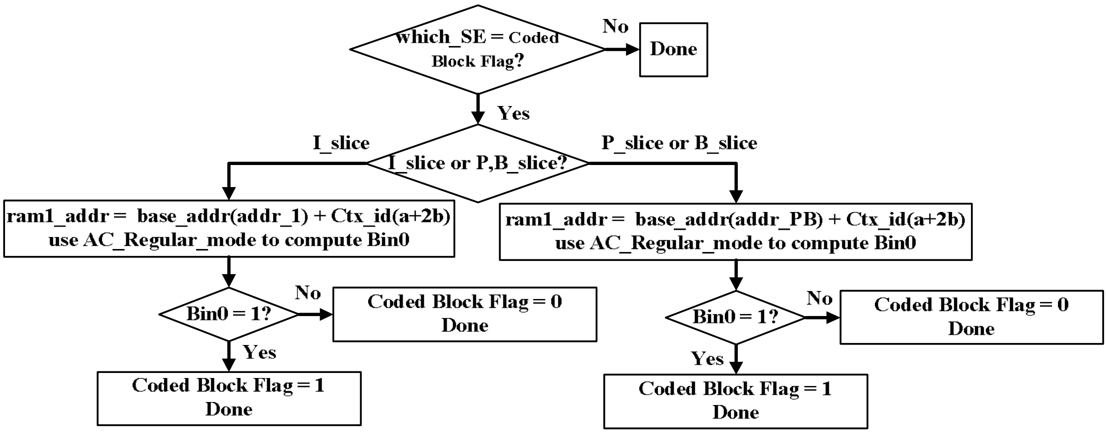


Fig 105. Interface of Coded Block Flag

4.15.2. Flow Chart of Coded Block Flag

The flow chart of Coded Block Flag is shown in Fig 106, and the binarization of Coded Block Flag is 1-bit FL coding.


which_SE = Coded Block Flag?
No
Done
Yes
I_slice
I_slice or P,B_slice?
P_slice or B_slice
ram1_addr = base_addr(addr_I) + Ctx_id(a+2b)
use AC_Regular_mode to compute Bin0
ram1_addr = base_addr(addr_PB) + Ctx_id(a+2b)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
No
Coded Block Flag = 0
Done
Yes
Coded Block Flag = 1
Done
Bin0 = 1?
No
Coded Block Flag = 0
Done
Yes
Coded Block Flag = 1
Done

Case(Block_type)

LUMA16DC:	addr_PB = 69, addr_I = 34
LUMA16AC:	addr_PB = 73, addr_I = 38
LUMA4x4:	addr_PB = 77, addr_I = 42
CHROMADC:	addr_PB = 81, addr_I = 46
CHROMADC2x4:	addr_PB = 81, addr_I = 46
CHROMADC4x4:	addr_PB = 81, addr_I = 46
CHROMAAC:	addr_PB = 85, addr_I = 50

Fig 106. Flow chart of Coded Block Flag

4.16. Significance Map

If the coded block flag indicates that a block has significant coefficients, the decoding Bin string of Significant Map syntax element is enabled. For each coefficient in scanning order, a 1-bit symbol significant_coeff_flag is used. If the significant_coeff_flag is equal to 1, which indicating that a non-zero coefficient exists at the scanning position, a further 1-bit symbol last_significant_coeff_flag is used. This symbol indicates whether the current significant coefficient is the last one inside the block or not. The last_significant_coeff_flag and significant_coeff_flag for the last scanning position of a block are never transmitted.

4.16.1. Interface of Significance Map

The interface of Significant Map is shown in Fig 107, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- type is a 3-bit input signal indicating the types of blocks. There are seven types using for this decoder: LUMA_16DC, LUMA_16AC, LUMA_4x4, CHROMA_DC, CHROMA_AC, CHROMA_DC_2x4 and CHROMA_DC_4x4.
- fld is a 1-bit input signal indicating the scanning order is zig-zag scan or interlace scan.
- from_top_or_AC is a 1-bit output signal which be described in 4.6.1. Interface of MB Skip Flag.
- Coeff_ctr is an 7-bit output signal indicating the number of non-zero coefficients in blocks.

- Coeff_15_to_0 is a 16-bit output signal. Each bit of this signal indicates there is zero or non-zero coefficient in corresponding position in a given scanning order for a block. If there is a non-zero coefficient in corresponding position, the corresponding bit of Coeff_15_to_0 is set to 1, otherwise, the corresponding bit of Coeff_15_to_0 is set to 0.

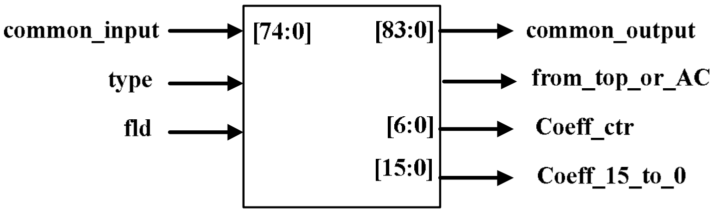
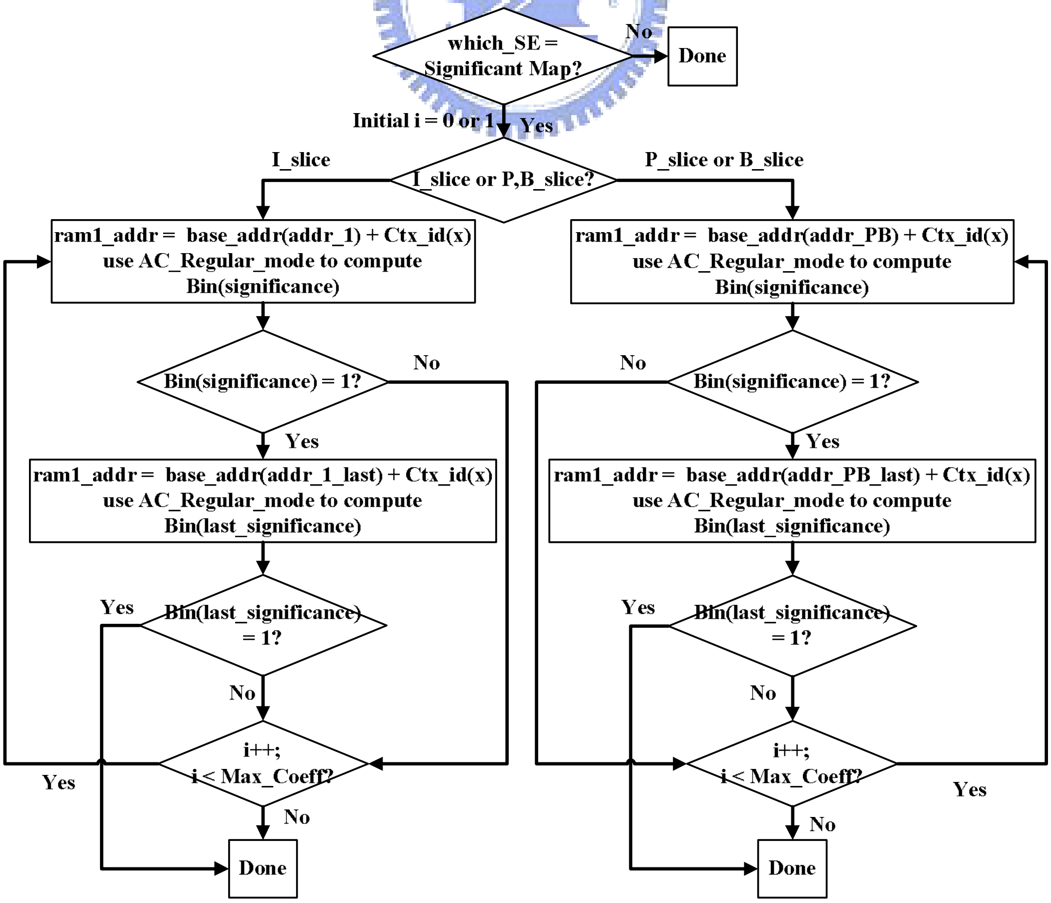


Fig 107. Interface of Significant Map

4.16.2. Flow Chart of Significance Map

The flow chart of Significant Map is shown in Fig 108, and the binarization of each significant_coeff_flag and last_significant_coeff_flag is 1-bit FL coding.


which_SE = Significant Map?
No
Done
Initial i = 0 or 1
Yes
I_slice
I_slice or P,B_slice?
P_slice or B_slice
ram1_addr = base_addr(addr_1) + Ctx_id(x)
use AC_Regular_mode to compute
Bin(significance)
ram1_addr = base_addr(addr_PB) + Ctx_id(x)
use AC_Regular_mode to compute
Bin(significance)
Bin(significance) = 1?
No
Yes
No
Bin(significance) = 1?
Yes
ram1_addr = base_addr(addr_1_last) + Ctx_id(x)
use AC_Regular_mode to compute
Bin(last_significance)
ram1_addr = base_addr(addr_PB_last) + Ctx_id(x)
use AC_Regular_mode to compute
Bin(last_significance)
Yes
Bin(last_significance) = 1?
No
Yes
Bin(last_significance) = 1?
No
i++;
i < Max_Coeff?
Yes
No
Done
i++;
i < Max_Coeff?
Yes
No
Done

```
Case(type)
LUMA16DC:
addr_I = 54; addr_PB = 89; addr_I_last = 130; addr_PB_last = 165; Max_Coeff =  15; Initial i  = 0;
LUMA16AC:
addr_I = 69; addr_PB = 104; addr_I_last = 145; addr_PB_last = 180; Max_Coeff =  15; Initial i  = 1;
LUMA4x4:
addr_I = 98; addr_PB = 133; addr_I_last = 168; addr_PB_last = 203; Max_Coeff =  15; Initial i  = 0;
CHROMADC:
addr_I = 113; addr_PB = 148; addr_I_last = 183; addr_PB_last = 218; Max_Coeff =  3; Initial i  = 0;
CHROMAAC:
addr_I = 116; addr_PB = 151; addr_I_last = 186; addr_PB_last = 221; Max_Coeff =  15; Initial i  = 1;
CHROMADC2x4:
addr_I = 113; addr_PB = 148;  addr_I_last = 183; addr_PB_last = 218; Max_Coeff =  7; Initial i  = 0;
CHROMADC4x4:
addr_I = 113; addr_PB = 148;  addr_I_last = 183; addr_PB_last = 218; Max_Coeff =  15; Initial i  = 0;
```

The value x in Ctx_id(x) is depending on the the scanning position and type

Fig 108. Flow chart of Significant Map

4.17. Level Information

The value of the significant coefficients is decoded using two coding symbols: coeff_abs_level_minus1 (representing the absolute value of the level minus 1), and coeff_sign_flag (representing the sign of levels).

4.17.1. Interface of Level Information

The interface of Level Information is shown in Fig 109, and the interface signals are explained as follows.

- common_input and common_output are 75-bit and 84-bit signals which be described in 4.6.1. Interface of MB Skip Flag.
- type is a 3-bit input signal which be described in 4.16.1. Interface of Significance Map.
- OK_AC_bypass is a 1-bit input signal which be described in 4.12.1 Interface of Motion Vector Difference.
- Coeff_in is a 16-bit input signal which connects to the Coeff_15_to_0 signal of Significant Map module.
- from_top_or_ACr_or_ACb is a 2-bit output signal which be described in 4.12.1 Interface of Motion Vector Difference.

- Enable_AC_bypass is a 1-bit output signal which be described in 4.12.1 Interface of Motion Vector Difference.
- Coeff_0_out ~ Coeff_15_out are each 16-bit output signal indicating the value of the significant coefficients in corresponding position of a block.

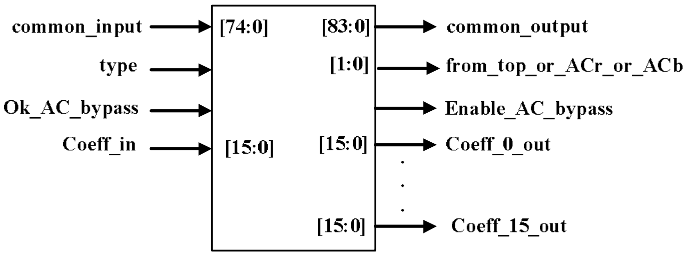
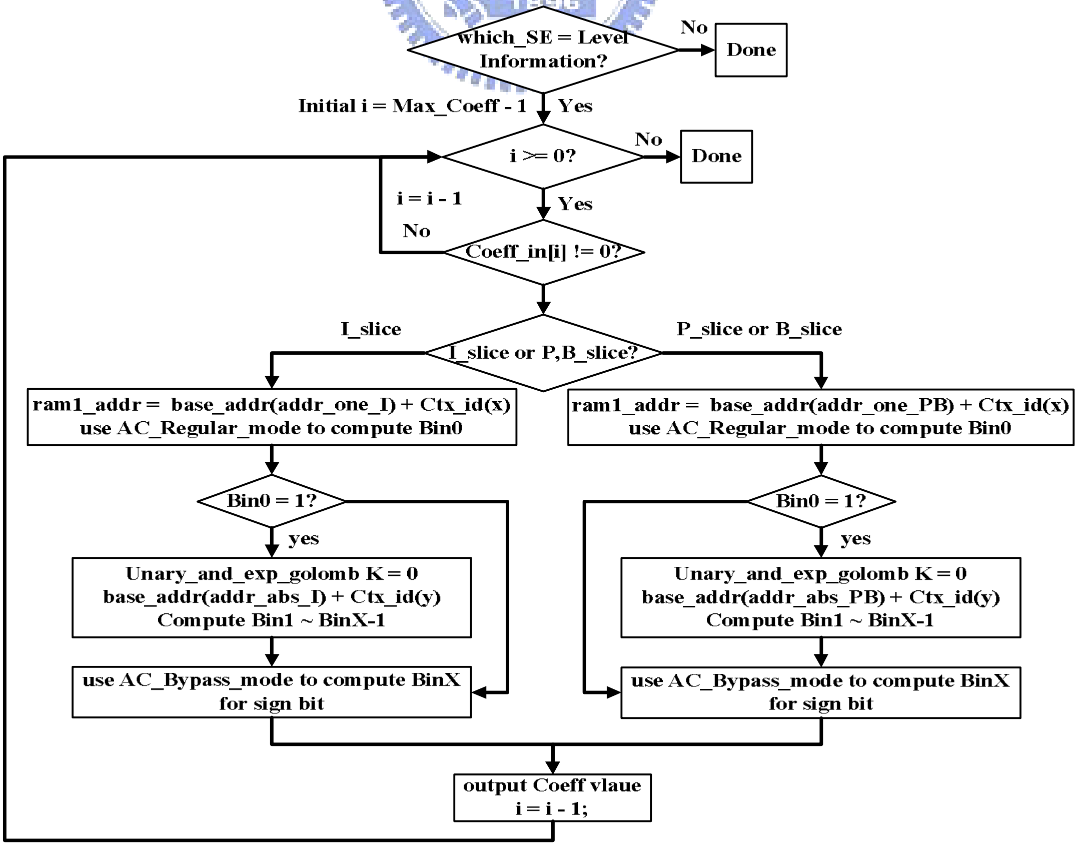


Fig 109. the interface of Level Information

4.17.2. Flow Chart of Level Information

The flow chart of Level Information is shown in 0, and the binarization of coeff_abs_level_minus1 is using UEG0 scheme with the cutoff value S = 13, and the binarization of coeff_sign_flag is using 1-bit FL coding.


which_SE = Level Information?
No
Done
Initial i = Max_Coeff - 1
Yes
i >= 0?
No
Done
i = i - 1
Yes
No
Coeff_in[i] != 0?
I_slice
P_slice or B_slice
I_slice or P,B_slice?
ram1_addr = base_addr(addr_one_I) + Ctx_id(x)
use AC_Regular_mode to compute Bin0
ram1_addr = base_addr(addr_one_PB) + Ctx_id(x)
use AC_Regular_mode to compute Bin0
Bin0 = 1?
yes
Bin0 = 1?
yes
Unary_and_exp_golomb K = 0
base_addr(addr_abs_I) + Ctx_id(y)
Compute Bin1 ~ BinX-1
Unary_and_exp_golomb K = 0
base_addr(addr_abs_PB) + Ctx_id(y)
Compute Bin1 ~ BinX-1
use AC_Bypass_mode to compute BinX for sign bit
use AC_Bypass_mode to compute BinX for sign bit
output Coeff vlaue
i = i - 1;

```
Case(type)
LUMA16DC:
addr_one_I = 200; addr_one_PB = 235; addr_abs_I = 230; addr_abs_PB = 265; Max_Coeff =  15;
LUMA16AC:
addr_one_I = 205; addr_one_PB = 240; addr_abs_I = 235; addr_abs_PB = 270; Max_Coeff =  15;
LUMA4x4:
addr_one_I = 215; addr_one_PB = 250; addr_abs_I = 245; addr_abs_PB = 280; Max_Coeff =  15;
CHROMADC:
addr_one_I = 220; addr_one_PB = 255; addr_abs_I = 250; addr_abs_PB = 285; Max_Coeff =  3;
CHROMAAC:
addr_one_I = 225; addr_one_PB = 260; addr_abs_I = 254; addr_abs_PB = 289; Max_Coeff =  15;
CHROMADC2x4:
addr_one_I = 220; addr_one_PB = 255; addr_abs_I = 250; addr_abs_PB= 285; Max_Coeff =  7;
CHROMADC4x4:
addr_one_I = 220; addr_one_PB = 255; addr_abs_I = 250; addr_abs_PB = 285; Max_Coeff =  15;
```

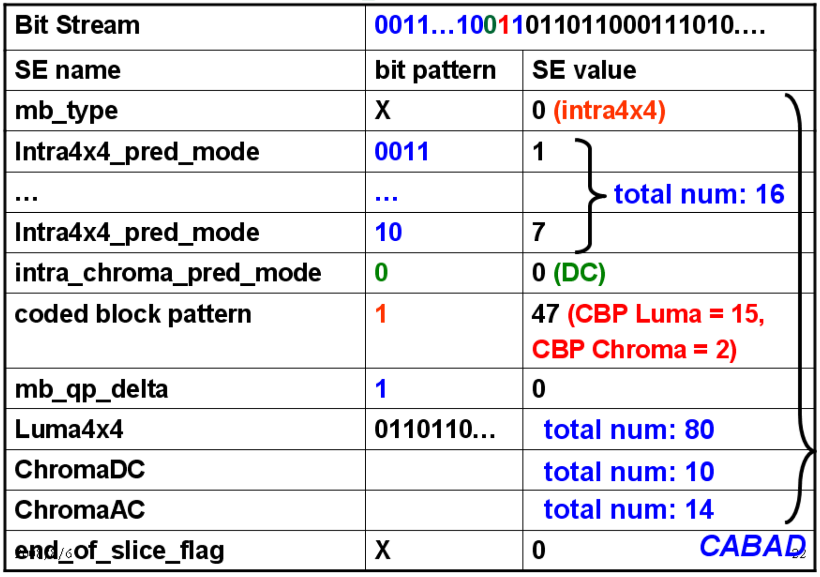
The value x in Ctx_id(x) is depending on the number of T1s

The value y in Ctx_id(y) is depending on type and Coeff_in

Fig 110. Flow chart of Level Information

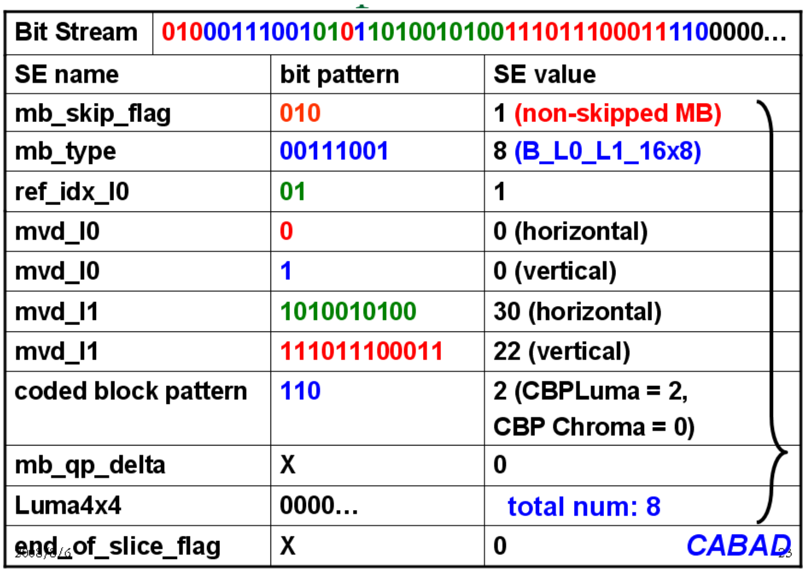
4.18. A CABAD Decoding Example

Bitstream examples for intra frame and inter frame are shown in Table 10 and Table 11, respectively. The entropy encoding method of all SEs is CABAC, so we use our CABAD logic to decode these SEs except end_of_slice_flag.



Bit Stream	0011…10011011011000111010….	
SE name	bit pattern	SE value
mb_type	X	0 (intra4x4)
Intra4x4_pred_mode	0011	1
…	…	total num: 16
Intra4x4_pred_mode	10	7
intra_chroma_pred_mode	0	0 (DC)
coded block pattern	1	47 (CBP Luma = 15, CBP Chroma = 2)
mb_qp_delta	1	0
Luma4x4	0110110…	total num: 80
ChromaDC		total num: 10
ChromaAC		total num: 14
end_of_slice_flag	X	0

Table 10. bitstream example for Intra frame



Bit Stream	010001110010101101001010011101110001111100000…	
SE name	bit pattern	SE value
mb_skip_flag	010	1 (non-skipped MB)
mb_type	00111001	8 (B_L0_L1_16x8)
ref_idx_l0	01	1
mvd_l0	0	0 (horizontal)
mvd_l0	1	0 (vertical)
mvd_l1	1010010100	30 (horizontal)
mvd_l1	111011100011	22 (vertical)
coded block pattern	110	2 (CBPLuma = 2, CBP Chroma = 0)
mb_qp_delta	X	0
Luma4x4	0000…	total num: 8
end_of_slice_flag	X	0 CABAD

Table 11.bitstream example for Inter frame

Fig 111 is an illustration of AC_Regular_mode of a binary arithmetic decoding engine for one bin. The arithmetic decoding engine keeps updating two 9-bit registers, namely "*range*" and "*offset*" during the whole decoding process. The *range* register keeps track of the width of the current interval while the *offset* register keeps track of the input bitstream.

When decoding a bin, *range* is split into 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 111, the left-side plot of Fig 111 shows the case that *MPS occurs*, where the *offset* is less than *rMPS*. The middle plot of Fig 111 shows the case that *LPS occurs*, where the *offset* is greater than or equal to *rMPS*. The renewal of the *range*, *range_new*, and the *offset*, *offset_new*, are shown in right-side of Fig 111. 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 bitstream. In this way, *offset* receives bits from the input bit stream to keep track of the position of the bitstream in the current interval.

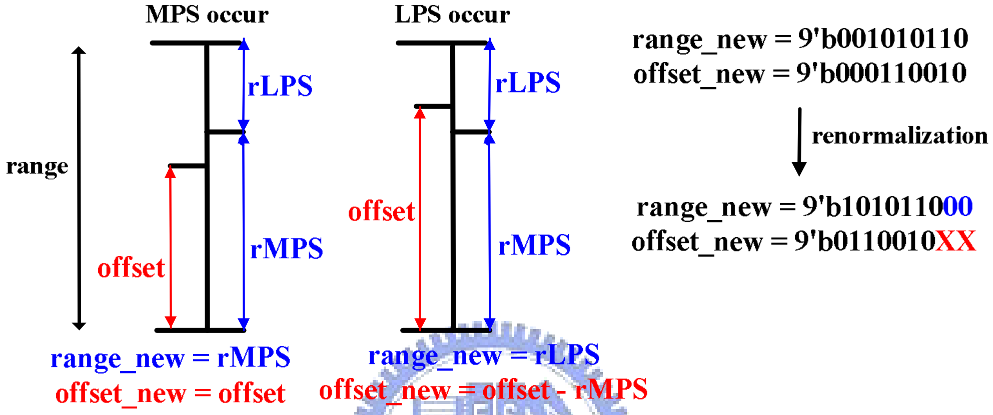


Fig 111. AC_Regular_mode of arithmetic decoding engine for one bin

In the following paragraphs, we will give an example for each mode of binary arithmetic coding.

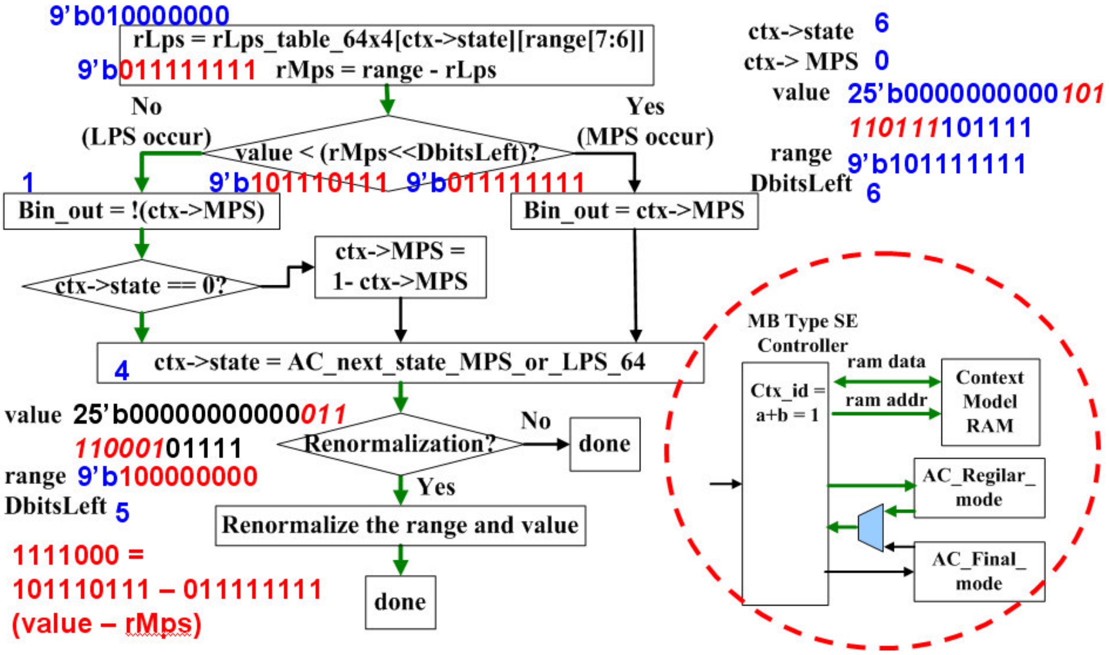


Fig 112. Example for AC_Regular_mode

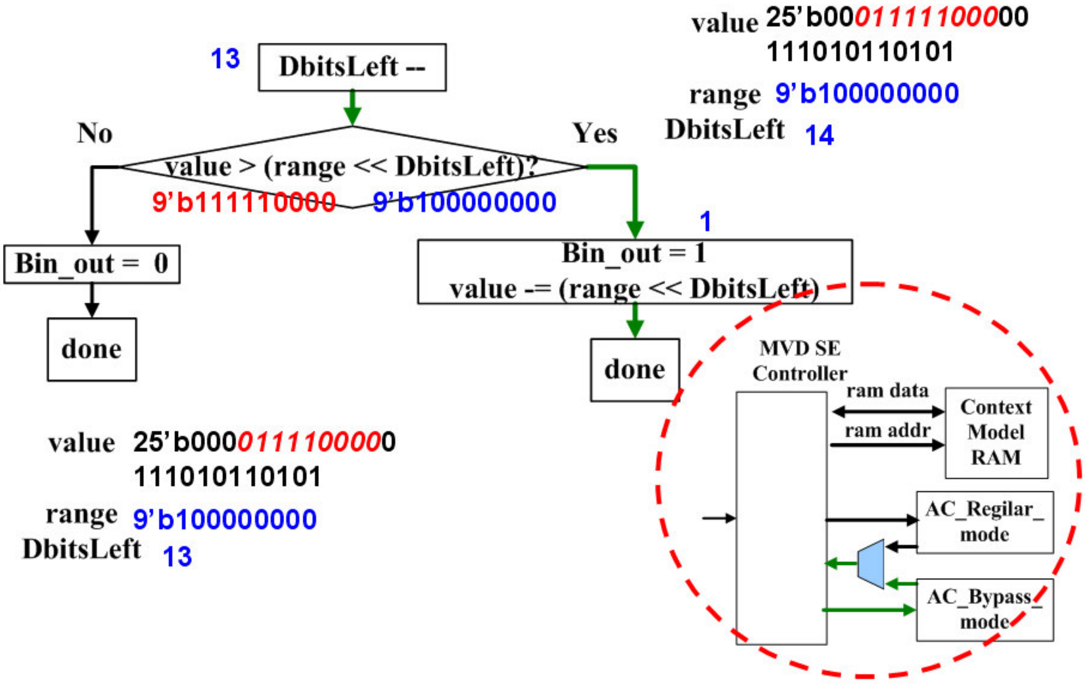


Fig 113. Example for AC_Bypass_mode

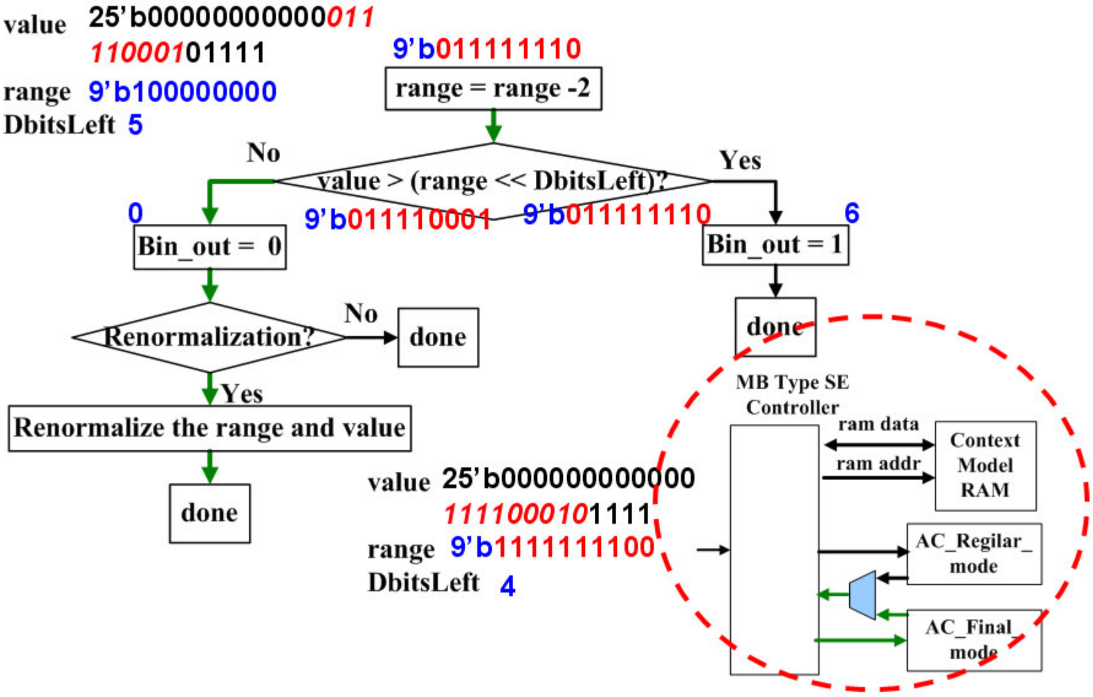


Fig 114. Example for AC_Final_mode

Examples of AC_Regular_mode, AC_Bypass_mode and AC_final_mode are shown in Fig 112 ~ Fig 114.

The “*offset*” in Fig 111 is the same as the 9-bit bit-pattern in value signals which is in italic type in Fig 112 ~ Fig 114, and the right-down side in Fig 112 ~ Fig 114 are our RTL model which implement faithfully by flow chart in Fig 112 ~ Fig 114.

Chapter 5. Experimental Results

This chapter presents some experimental results. As mentioned in Chapter 1, the hardware-software co-implementation platform we used to verify our design is the LEON3/GRLIB platform developed by GR Research. The development board we used for the experiments is Xilinx ML-506. The hardware development toolchain is Xilinx ISE 10.1.01 and the software development toolchain is the gcc-based cross compiler for SPARC processor. The operating system used to support the software partition of the AVC decoder is the eCos operating system. The parameters we used to synthesize the hardware partition of the decoder (i.e. the CAVLD and CABAD logic) is shown in Table 12.

Target Device	xc5vsx50t-1ff1136
Product Version	ISE 10.1.01
Design Goal	Balanced (Area and Speed)
Target Clock Rate	50MHz

Table 12. Synthesis settings

We have modified AVC reference software JM12.2 decoder so that the entropy decoder is replaced by the proposed hardware logic. The target platform is running at 50MHz with a soft core LEON processor and hence the decoding speed is slower than real time. However, the decoded YCbCr frame data is transferred from ML-506 back to a host computer throught Ethernet connection so that we can visually as well as computationally verify the correctness of the decoder.

5.1. Synthesis Results of the Proposed Design

According to the synthesis report, the maximal working frequencies of the proposed CAVLD architecture and CABAD architecture are 113MHz and 53MHz, respectively. Since we must integrate both logics into the target platform, we set the system target clock rate to 50MHz to simplify the clocking system. Table 13 shows the resource statistics of the synthesized CAVLD and CABAD logics.

	CAVLD	CABAD
Adders/Subtractors	21	61
Registers	711	1681
Comparators	35	22
Multiplexers	1	2
Logic shifters	4	8
RAMs	0	1 (128x6-bit single-port block RAM)
ROMs	0	3 (16x20-bit, 256x9-bit, 32x3-bit)
Multipliers	0	1 (9x9-bit)

Table 13. Macro statistics of CAVLD and CABAD logics

Table 14 shows slice utilization of the CAVLD and CABAD logics. The CAVLD and CABAD only occupy 4% and 17%, respectively, of the total number of slice LUTs, which is 32640 for xc5vsx50t-1ff1136.

	CAVLD	CABAD
Number of Slice LUTs	1443	5697

Table 14. Slice logic utilization of CAVLD and CABAD logics

5.2. Performance of the Proposed Design

This section presents the performance of the proposed CAVLD and CABAD architecture implemented on the ML-506 development board.

5.2.1. Performance of the CAVLD Logic

We used five QCIF (176x144) bitstreams, including Foreman, Akiyo, Silent, Mobile, and Stefan, to test the decoder. Each bitstream has 300 frames and is encoded using the parameters shown in Table 15.

	I_SLICE	P_SLICE
Number of Slice	150	150
QP	28	28
YUV format	YUV 420	
Frame/Sec	30	

Table 15. Encoding parameters of each test sequence

Table 16 shows the number of cycles, blocks, coefficients, trailing-ones and total-zeros of the 4x4 blocks in I_SLICE of each sequence.

	number of cycles	number of blocks	number of coefficients	number of trailing-ones	number of total-zeros
Foreman	12957982	248104	623795	321234	491820
Akiyo	9264316	238062	409171	207393	291440
Silent	13585841	290325	661469	378364	516597
Mobile	44027768	338343	2136923	589270	1097691
Stefan	31091208	294780	1499456	436746	771077

Table 16. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of the 4x4 blocks in I_SLICE of each sequence

Table 17 shows the number of cycles, blocks, coefficients, trailing-ones and total-zeros of 4x4 blocks in P_SLICE of each sequence.

	number of cycles	number of blocks	number of coefficients	number of trailing-ones	number of total-zeros
Foreman	890813	31514	39605	29301	40615
Akiyo	47062	1656	2024	1484	3135
Silent	684186	22362	30803	21541	28447
Mobile	4093272	99088	191661	129980	399816
Stefan	5593232	110289	266621	159607	449795

Table 17. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of the 4x4 blocks in P_SLICE of each sequence

Table 18 shows the number of cycles, blocks, coefficients, trailing-ones and total-zeros of the 2x2 blocks in I_SLICE of each sequence.

	number of cycles	number of blocks	number of coefficients	number of trailingones	number of totalzeros
Foreman	638515	21344	30077	20210	9743
Akiyo	782652	21216	38063	19524	9737
Silent	875081	24868	43182	27363	13350
Mobile	1614583	28216	77414	18516	12332
Stefan	1087639	24762	52752	25279	15368

Table 18. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of the 2x2 blocks in I_SLICE of each sequence

Table 19 shows the number of cycles, blocks, coefficients, trailingones and totalzeros of the 2x2 blocks in P_SLICE of each sequence.

	number of cycles	number of blocks	number of coefficients	number of trailingones	number of totalzeros
Foreman	33315	1810	1277	1154	913
Akiyo	3595	220	124	124	160
Silent	25960	1326	1057	951	883
Mobile	83196	6012	2209	2095	2526
Stefan	150080	7820	5870	5019	4776

Table 19. The number of decoding cycles, blocks, coefficients, trailing-ones and total-zeros of 2x2 blocks in P_SLICE of each sequence

According to Table 16 ~ Table 19, we can infer the number of decoding cycles per each block type as shown in Table 20.

Block type	4x4 block in I_SLICE	4x4 block in P_SLICE	2x2 block in I_SLICE	2x2 block in P_SLICE
Foreman	52.22	28.26	29.91	18.40
Akiyo	38.91	28.41	36.89	16.34
Silent	46.79	30.59	35.19	19.57
Mobile	130.12	41.30	57.22	13.83
Stefan	105.47	50.71	43.92	19.19

Table 20. The number of decoding cycles per each block type of each sequence

The totals bits and Mbits/sec of each slice type in each sequence is shown in Table 21, and the average Mbits/sec of these test sequences is 11.66 Mbits/sec.

test sequence	total bits of I_SLICE	total bits of P_SLICE	Mbits/sec
Foreman	3202422	225748	11.8
Akiyo	2379050	12459	11.8
Silent	3452746	174532	12.0
Mobile	10259178	1126972	11.3
Stefan	7157489	1505532	11.4

Table 21. Totals bits and Mbits/sec of each slice type of each sequence

5.2.2. Performance of the CABAD Logic

We used five QCIF (176x144) bitstreams, including Foreman, Akiyo, Silent, Mobile, and Stefan, to test the decoder. Each bitstream has 299 frames and is encoded using the parameters shown in Table 22.

	I_SLICE	P_SLICE	B_SLICE
Number of Slice	75	75	149
QP	28	28	30
YUV format	YUV 420		
Frame/Sec	30		

Table 22. Encoding parameters of each test sequence

The total number of cycles used to decode each slice type in each sequence is shown in Table 23.

test sequence	total cycles of I_SLICE	total cycles of P_SLICE	total cycles of B_SLICE
Foreman	10625867	2259472	945778
Akiyo	7770697	606804	149422
Silent	11570781	1655197	756183
Mobile	31412088	5994625	1049103
Stefan	22450752	8171555	3247914

Table 23. Total number of decoding cycles of each slice type of each sequence

The number of cycles per MB of each slice type in each sequence is shown in Table 24.

test sequence	cycles/MB of I_SLICE	cycles/MB of P_SLICE	cycles/MB of B_SLICE
Foreman	1431	304	64
Akiyo	1047	81	10
Silent	1558	222	51
Mobile	4230	807	71
Stefan	3023	1100	220

Table 24. Cycle/MB of each slice type of each sequence

For each non-skipped MB, the average number of occurrences of each syntax element in I_SLICEs of each sequence is shown in Table 25.

Test sequence	MB Type	Sub MB Type	Intra Prediction Mode for Luma4x4	Intra Prediction Mode for Chroma	Reference Frame Index	
Foreman	1	0	14.53	1	0	
Akiyo	1	0	13.09	1	0	
Silent	1	0	15.49	1	0	
Mobile	1	0	15.98	1	0	
Stefan	1	0	14.90	1	0	
	Motion Vector Difference	Coded Block Pattern	MB Based Quantization Parameter	Coded Block Flag	Significant Map	Level Information
Foreman	0	0.90	0.99	18.04	12.71	44.93
Akiyo	0	0.82	0.99	17.45	9.82	31.12
Silent	0	0.97	1	21.18	15.51	49.26
Mobile	0	0.99	1	24.61	21.83	148.97
Stefan	0	0.93	0.99	21.52	17.59	104.45

Table 25. The average number of occurrences of each syntax element in I_SLICEs

For each non-skipped MB, the average number of occurrences of each syntax element in P_SLICEs of each sequence is shown in Table 26.

Test sequence	MB Type	Sub MB Type	Intra Prediction Mode for Luma4x4	Intra Prediction Mode for Chroma	Reference Frame Index	
Foreman	1	1.27	0.50	0.05	2.19	
Akiyo	1	0.54	0	0	1.55	
Silent	1	0.84	0.37	0.02	1.83	
Mobile	1	1.96	0	0	2.66	
Stefan	1	1.84	1.14	0.07	2.74	
	Motion Vector Difference	Coded Block Pattern	MB Based Quantization Parameter	Coded Block Flag	Significant Map	Level Information
Foreman	5.68	0.98	0.48	4.34	2.85	5.92
Akiyo	3.59	1	0.07	0.42	0.22	0.49
Silent	4.47	0.99	0.29	2.79	1.80	3.95
Mobile	7.86	1	0.84	10.86	6.69	20.73
Stefan	7.73	0.99	0.78	11.9	8.94	32.89

Table 26. The average number of occurrences of each syntax element in P_SLICEs

For each non-skipped MB, the average number of occurrences of each syntax element in B_SLICEs of each sequence is shown in Table 27.

Test sequence	MB Type	Sub MB Type	Intra Prediction Mode for Luma4x4	Intra Prediction Mode for Chroma	Reference Frame Index	
Foreman	1	0.39	.0.08	0.01	0.80	
Akiyo	1	0.53	0	0	0.73	
Silent	1	1.12	0.12	0.01	0.93	
Mobile	1	0.66	0	0	1.06	
Stefan	1	1.30	0.04	0.01	10.9	
	Motion Vector Difference	Coded Block Pattern	MB Based Quantization Parameter	Coded Block Flag	Significant Map	Level Information
Foreman	3.8	0.99	0.22	1.64	0.94	1.91

Akiyo	3.68	1	0.08	0.40	0.22	0.56
Silent	4.72	0.99	0.44	3.67	2.34	4.80
Mobile	4.89	1	0.30	2.74	1.57	6.36
Stefan	2.44	0.99	0.54	5.12	3.22	10.99

Table 27. The average number of occurrences of each syntax element in B_SLICEs

The totals bits and Mbits/sec of each slice type in each sequence is shown in Chapter 1, the average Mbits/sec of these test sequence is 8.68 Mbits/sec.

test sequence	total bits of I_SLICE	total bits of P_SLICE	Total bits of B_SLICE	Mbits/sec
Foreman	1876084	398261	146035	8.8
Akiyo	1412962	76276	6860	8.8
Silent	2028917	278075	120843	8.7
Mobile	5381058	1034163	175381	8.6
Stefan	3767965	1403374	574299	8.5

Table 28. Totals bits and Mbits/sec of each slice type in each sequence

5.2.3. Comparisons with Previous Work

In this section, we compare our design with some previously published design.

5.2.3.1. Comparison of CAVLD Logic with the Design in [8]

First, we compare the proposed CAVLD design with that in [8]. Table 1 shows the average required cycles to decode one MB by our proposed CALVD logics for different sequences and Table 30 shows the maximum working frequency for our proposed design and design in [8], and the target devices are different (FPGA vs. 0.18μm process) so that the maximal working frequencies cannot be compared directly.

Test Sequence			Akiyo	Foreman	Stefan	Mobile
QP 28	average cycles/MB	Proposed design	676	915	2166	3073
		Design in [8]	38	53	124	174

Table 29. Comparisons of average cycle / MB with design in [8]

	Max MHz
Proposed design	219 (FPGA)
Design in [8]	125 (0.18 μm)

Table 30. Comparisons of maximum working frequency with design in [8]

It is obvious that the design in [8] requires much less cycles per MB for CAVLD decoding. However, since CAVLD is usually used only for low bitrate contents (e.g. less than 10 mbps in general), our design goal for CAVLD is not targeted for highest performance but for lowest logic resource usage.

5.2.3.2. Performance of CABAD Logic

Table 1 shows the average required cycles to decode one MB by our proposed CABAD logics for different slice types and Table 32 shows the maximum working frequency for our proposed design and design in [18]. Please note that, the cycles per MB in our design did not take memory access cycle into account. Also, the target devices are different (FPGA vs. TSMC 0.18μm process) so that the maximal working frequencies cannot be compared directly.

Test Sequence		I_SLICE (QP = 36)	P_SLICE (QP = 26)	B_SLICE (QP = 26)
average cycles/MB	Proposed design	1017	901	419
	Design in [18]	462	307	253

Table 31. Comparisons of average cycle / MB with design in [18]

	Max MHz
Proposed design	60 (FPGA)
Design in [18]	120 (TSMC 0.18 μm)

Table 32. Comparisons of maximal working frequency with design in [18]

5.2.4. Performance Analysis on Target Platform

Although the cycle-accurate performances of the proposed logics when working in an ideal situation are analyzed in previous sections, we still have to measure the performance of the proposed logics when they are integrated into a real working system. In this section, we measure the actual average cycles per macroblocks for each of the logic in decoding a QCIF version of the (176x144) Foreman bitstream on the development board. This bitstream has 300 frames and is encoded using the parameters shown in Table 15.

Table 33 shows the measured average cycles/MB performance of the CAVLD logic on the target platform, comparing to the cycle-accurate estimate of the logic when working in an ideal situation (that is, zero wait-state memory and no bus contensions from other logics). It is obvious from Table 33 that external memory accesses adds a large overhead on the performance of the proposed design on the target platform.

CAVLD logic	I_SLICE	P_SLICE
average cycles/MB on the target platform	12041	1436
average cycles/MB using estimation	915	62

Table 33. Performance of the CAVLD logic on the target platform

Table 34 shows the measured average cycles/MB of the CABAD logic on the target platform. The experiment is conducted using a QCIF (176x144) version of Foreman bitstreams with 299 frames and is encoded using the parameters shown in Table 22. Again, external memory access overhead is very high.

CABAD logic	I_SLICE	P_SLICE	B_SLICE
average cycles/MB on experimental	12427	2521	1526

development board			
average cycles/MB using estimation	1434	304	64

Table 34. Comparisons of average cycles/MB on experimental development board and using estimation of CABAD logic

As shown in Table 33 and Table 34, the average numbers of processing cycles per macroblock on the target platform is much more than what we expect based on the cycle count analysis of both logics. This is due to high external memory access overhead of current software-hardware interface on the target paltform. However, we have to point out that the software-hardware interface design in the experimental system is far from being optimized. We only modify JM12.2 (which is very inefficient) slightly to enable verification of the correctness of the proposed logics. Therefore, the external memory access overhad can be significantly reduced if the logic is intended to be used in a practical system.

Chapter 6. Conclusions and Future Work

In this thesis, we proposed the hardware architecture for CAVLD and CABAD decoding of the AVC/H.264 standard. The design has been verified on the Xilinx Vertex 5-based FPGA development board, ML506, using full system verification with the AVC/H.264 reference software JM 12.2. The proposed design achieves reasonable performance (8 ~ 11 mbps at 50MHz) with small logic area. Therefore, it is promising for practical applications.

There are several things that can be improved further. In the proposed design, both the CAVLD and the CABAD logics perform decoding in sequential manner. It is possible to perform concurrent decoding of multiple bits at the cost of more on-chip memory and logic complexity. For high quality full HD applications defined by the AVCHD specification, we have to support up to 24 mbps CABAD decoding in real time. This is beyond the capability of the current design. More advanced architecture and careful pipeline design is required to improve the maximal working frequency and throughput per cycle so that the performance can be tripled to support full AVCHD specification.

References

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

[2] Ming-Ting Sun and Shaw-Min Lei, “A High-speed Entropy Decoder For HDTV,” *Custom Integrated Circuits Conference, 1992., Proceedings of the IEEE 1992,* May, 1992.

[3] Wu Di, Gao Wen, Hu Mingzeng and Ji Zhenzhou, “A VLSI architecture design of CAVLC decoder,” *ASIC, 2003. Proceedings. 5th International Conference,* Oct. 2003.

[4] Hsiu-Cheng Chang, Chien-Chang Lin and Jiun-In Guo, “A novel low-cost high-performance VLSI architecture for MPEG-4 AVC/H.264 CAVLC decoding,” *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium,* May 2005.

[5] Yong Ho Moon, Gyu Yeong Kim and Jae Ho Kim, “An efficient decoding of CAVLC in H.264/AVC video coding standard,” *Consumer Electronics, IEEE Transactions,* Aug. 2005.

[6] Alle, M., Biswas, J. and Nandy, S.K. “High Performance VLSI Architecture Design for H.264 CAVLC Decoder,” *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference*, Sept. 2006.

[7] Heng-Yao Lin, Ying-Hong Lu, Bin-Da Liu and Jar-Ferr Yang, “Low power design of H.264 CAVLC decoder,” *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium*, May 2006.

[8] Guo-Shiuan Yu and Tian-Sheuan Chang, "A zero-skipping multi-symbol CAVLC decoder for MPEG-4 AVC/H.264," *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium*, May 2006.

[9] Shau-Yin Tseng and Tien-Wei Hsieh, "A Pattern-Search Method for H.264/AVC CAVLC Decoding," *Multimedia and Expo, 2006 IEEE International Conference*, July 2006.

[10] Yong-Hwan Kim, Yoon-Jong Yoo, Jeongho Shin, Byeongho Choi and Pa, J., "Memory-efficient H.264/AVC CAVLC for fast decoding," *Consumer Electronics, IEEE Transactions*, Aug. 2006.

[11] Marpe, D., Schwarz, H. and Wiegand, T., "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *Circuits and Systems for Video Technology, IEEE Transactions*, July 2003.

[12] Marpe, D. and Wiegand, T., "A highly efficient multiplication-free binary arithmetic coder and its application in video coding," *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference*, Sept. 2003.

[13] Mrak, M., Marpe, D. and Wiegand, T., "A context modeling algorithm and its application in video compression," *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference*, Sept. 2003.

[14] Mrak, M., Marpe, D. and Grgic, S., "Comparison of context-based adaptive binary arithmetic coders in video compression," *Video/Image Processing and Multimedia Communications, 2003. 4th EURASIP Conference*, July 2003.

[15] Osorio, R.R. and Bruguera, J.D., "Arithmetic coding architecture for H.264/AVC CABAC compression system," *Digital System Design, 2004. DSD 2004. Euromicro Symposium*, Sept. 2004.

[16] Jian-Wen Chen, Cheng-Ru Chang and Youn-Long Lin, “A hardware accelerator for context-based adaptive binary arithmetic decoding in H.264/AVC,” *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium*, May 2005.

[17] Wei Yu and Yun He, “A high performance CABAC decoding architecture,” *Consumer Electronics, IEEE Transactions*, Nov. 2005.

[18] Yao-Chang Yang, Chien-Chang Lin, Hsui-Cheng Chang, Ching-Lung Su and Jiun-In Guo, “A High Throughput VLSI Architecture Design for H.264 Context-Based Adaptive Binary Arithmetic Decoding with Look Ahead Parsing,” *Multimedia and Expo, 2006 IEEE International Conference*, July 2006.

[19] Eeckhaut, H., Christiaens, M., Stroobandt, D., Nollet, V., “Optimizing the critical loop in the H.264/AVC CABAC decoder,” *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference*, Dec. 2006.

[20] Chung-Hyo Kim and In-Cheol Park, “High speed decoding of context-based adaptive binary arithmetic codes using most probable symbol prediction,” *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium*, May 2006.

[21] Yi, Y. and Park, I.-C., “High-Speed H.264/AVC CABAC Decoding,” *Circuits and Systems for Video Technology, IEEE Transactions*, April 2007.

[22] Yan Zheng, Shibao Zheng, Zhonghua Huang and Ziliang Zhao, “A Time and Storage Optimized Hardware Design for Context-Based Adaptive Binary Arithmetic Decoding in H.264/AVC,” *Multimedia and Expo, 2007 IEEE International Conference*, July 2007.

[23] Jian-Wen Chen and Youn-Long Lin, “A High-Performance Hardwired CABAC Decoder,” *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE*

International Conference, April 2007.

[24] Peng Zhang, Wen Gao, Don Xie and Di Wu, "High-Performance CABAC Engine for H.264/AVC High Definition Real-Time Decoding," *Consumer Electronics, 2007. ICCE 2007. Digest of Technical Papers. International Conference*, Jan. 2007.

[25] Mei-hua Xu, Yu-lan Cheng, Feng Ran, Zhang-jin Chen, "Optimizing Design and FPGA Implementation for CABAC Decoder," *High Density packaging and Microsystem Integration, 2007. HDP '07. International Symposium*, June 2007.

[26] Joint Video Team (JVT), AVC reference software Joint Model version 12.2 (JM12.2), Aug. 2007, available from http://bs.hhi.de/~suehring/tml/.