Joint Collaborative Team on Video Coding (JCT-VC)
of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11
2nd Meeting: Geneva, CH, 21-28 July, 2010

Document: JCTVC-B205

Title: **Test Model under Consideration**

Status: Output Document (draft0075)

Source: JCT-VC

Abstract

This document provides the description for a Test Model under Consideration (TMuC). This suggested Test Model is intended to provide a coding efficiency close to the best performing proposals in the subjective test of the CfP submissions and also to provide a complexity point that is close to that of the lowest complexity submissions shown to provide substantial improvement of coding efficiency. The inclusion of a technology in this document does not indicate a final adoption of the technology as an element of an approved test model of the committee. Rather, it indicates a preliminary selection which may require further evaluation and justification to achieve that status.

This TMuC is based on the decisions taken at the 1st meeting of JCT-VC in Dresden from 15-23 April 2010 and the further refinements agreed at the 2nd meeting of JCT-VC in Geneva from 21-28 July 2010. The principal documents agreed at the 2nd meeting of JCT-VC which affect this TMuC are:

- Unified intra prediction (JCTVC-B100)
- Rounding control and transform precision expansion (JCTVC-B074)

TABLE OF CONTENTS

Abstract....1
1 Overview....5
2 Definitions....5
3 Conventions....6
3.1 Arithmetic operators....6
3.2 Logical operators....6
3.3 Relational operators....6
3.4 Bit-wise operators....7
3.5 Assignment operators....7
3.6 Range notation....7
3.7 Mathematical functions....7
3.8 Order of operation precedence....8
3.9 Variables, syntax elements, and tables....9
3.10 Text description of logical operations....10
3.11 Processes....11
4 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships....11
4.1 Spatial subdivision of pictures and slices....11
4.2 Inverse scanning processes....12
4.2.1 Inverse largest coding unit scanning process....12
4.2.2 Recursive coding unit scanning process....12
4.2.3 Inverse coding unit scanning process....13
4.2.4 Inverse prediction unit, partition and transform unit scanning process....13
4.2.4.1 Inverse prediction unit scanning process....14
4.2.4.2 Inverse prediction unit partition scanning process....14
4.2.4.3 Inverse transform unit scanning process....14
4.3 Derivation processes for neighbours....15

INTEL-1018
10,142,659

4.3.1 Derivation process of the availability for largest coding unit addresses 15
4.3.2 Derivation process for neighbouring largest coding unit addresses and their availability 15
4.3.3 Derivation process of the availability for coding units 16
4.3.4 Derivation process for neighbouring units and partitions 16
4.3.4.1 Derivation process for neighbouring largest coding units 17
4.3.4.2 Derivation process for neighbouring coding units 17
4.3.4.3 Derivation process for neighbouring prediction units 18
4.3.4.4 Derivation process for neighbouring partitions 18
4.3.5 Derivation process for neighbouring locations 19
5 Syntax and Semantics 20
5.1 Syntax 20
5.1.1 NAL unit syntax 21
5.1.2 Sequence parameter set RBSP syntax 22
5.1.3 Picture parameter set RBSP syntax 23
5.1.4 Supplemental enhancement information RBSP syntax 23
5.1.5 Supplemental enhancement information message syntax 23
5.1.6 Access unit delimiter RBSP syntax 24
5.1.7 Filler data RBSP syntax 24
5.1.8 Slice layer RBSP syntax 24
5.1.9 RBSP slice trailing bits syntax 25
5.1.10 RBSP trailing bits syntax 25
5.1.11 Slice header syntax 26
5.1.11.1 Reference picture list modification syntax 27
5.1.11.2 Decoded reference picture marking syntax 27
5.1.11.3 Adaptive loop filter parameter syntax 28
5.1.11.4 Switched Filter with Offsets parameter syntax 32
5.1.12 Slice data syntax 34
5.1.13 Coding unit syntax 35
5.1.14 Prediction unit syntax 36
5.1.15 Transform unit syntax 39
5.1.16 Parallel V2V syntax 40
5.1.16.1 Byte-align prefix code 40
5.1.16.2 Parrallel slice header syntax 41
5.2 Semantics 42
5.2.1 NAL unit semantics 42
5.2.2 Sequence parameter set RBSP semantics 44
5.2.3 Picture parameter set RBSP semantics 46
5.2.4 Supplemental enhancement information RBSP semantics 47
5.2.5 Supplemental enhancement information message semantics 47
5.2.6 Access unit delimiter RBSP semantics 47
5.2.7 Filler data RBSP semantics 47
5.2.8 Slice layer RBSP semantics 47
5.2.9 RBSP slice trailing bits semantics 47
5.2.10 RBSP trailing bits semantics 48
5.2.11 Slice header semantics 48
5.2.11.1 Reference picture list modification semantics 50
5.2.11.2 Decoded reference picture marking semantics 51
5.2.11.3 Adaptive loop filter parameter semantics 54
5.2.11.4 Switched Filter with Offsets parameter syntax 60
5.2.12 Slice data semantics 60
5.2.13 Coding unit semantic 60
5.2.14 Prediction unit semantics 61
5.2.15 Transform unit semantics 64
5.2.16 Parallel V2V header semantic 65
6 Decoding Process 65
6.1 Slice decoding process 66
6.1.1 Decoding process for picture order count 66
6.1.2 Decoding process for reference picture lists construction 66
6.1.3 Decoded reference picture marking process 66
6.2 Intra prediction process 66
6.2.1 Intra prediction process for luma samples 66

6.2.1.1 Derivation process for the IntraPredMode 67
6.2.1.2 Intra sample prediction 69
6.2.1.2.1 Specification of intra planar prediction 70
6.2.1.2.2 Specification of directional intra prediction 71
6.2.2 Intra prediction process for chroma samples 76
6.3 Inter prediction process 77
6.3.1 Derivation process for motion vector components and reference indices 79
6.3.1.1 Derivation process for luma motion vectors for skipped and direct prediction units in P slices 80
6.3.1.2 Derivation process for luma motion vectors for skipped and direct prediction units in B slices 80
6.3.1.3 Derivation process for luma motion vector prediction 81
6.3.1.3.1 Derivation process for luma motion vector prediction with explicit mode 81
6.3.1.3.2 Derivation process for motion vector predictor candidates 82
6.3.1.3.3 Derivation process for median luma motion vector prediction 84
6.3.1.3.4 Derivation process for temporal luma motion vector prediction 84
6.3.1.3.5 Removal process for motion vector prediction 86
6.3.1.3.6 Derivation process for luma motion vector prediction with implicit mode 87
6.3.1.4 Derivation process chroma motion vectors 88
6.3.2 Decoding process for inter prediction sample 88
6.3.2.1 Reference picture selection process 89
6.3.2.2 Fractional sample interpolation process 89
6.3.2.2.1 Luma sample interpolation process 90
6.3.2.2.2 Chroma sample interpolation process 95
6.4 Transform coefficient decoding process and picture construction process 95
6.4.1 Specification of transform decoding process for luma transform unit 95
6.4.2 Specification of transform decoding process for chroma transform unit 96
6.4.3 Inverse scanning process for transform coefficients 96
6.4.4 Adaptive Coefficient Scanning 96
6.4.5 Scaling and transformation process for transform unit 97
6.4.5.1 Mode dependent directional transformation process for intra prediction residuals 97
6.4.5.1.1 Mode dependent directional transform for 4x4 block input 99
6.4.5.1.2 Mode dependent directional transform for 8x8 block input 99
6.4.5.2 Rotational transformation process 100
6.4.5.2.1 Rotational transformation process for 4x4 transform coefficients 101
6.4.5.2.2 Rotational transformation process for 8x8 transform coefficients 102
6.4.5.3 Scaling process for residual blocks 104
6.4.5.4 Transformation process for residual blocks 104
6.4.5.4.1 Transformation process for N-point input 104
6.4.5.4.2 Transformation process for 4-point input 105
6.4.5.4.3 Transformation process for 8-point input 105
6.4.5.4.4 Transformation process for 16-point input 105
6.4.5.4.5 Transformation process for 32-point input 106
6.4.5.4.6 Transformation process for 64-point input 107
6.4.6 Picture construction process prior to deblocking filter process 110
6.5 In-loop filter process 110
6.5.1 Deblocking filter process 110
Luma filtering 113
Chroma filtering 114
Planar mode filtering 114
6.5.2 Adaptive loop filter process 115
6.5.2.1 Derivation process for filter coefficients 116
6.5.2.2 Filter process for luma samples 119
6.5.2.3 Filter process for chroma samples 120
7 Parsing Process 120
7.1 VLC Parsing process for side information 120
7.1.1 Derivation of ctxID for prediction unit mode 121
7.1.2 VLC Parsing process for prediction unit mode 121
7.1.2.1 Parsing process for the mode_table_idx 121
7.1.2.2 Derivation of modeIdx 122
7.1.3 VLC Parsing process for coded block pattern 122
7.1.3.1 Parsing process for the cbp_table_idx 122
7.1.3.2 Derivation of modeIdx 123

7.2 VLC Parsing process for transform coefficients 123
7.2.1 Derivation of ctxID for transform coefficients 124
7.2.2 Parsing process for the level and position of last non-zero coefficient 124
7.2.2.1 Parsing process for the position of last non-zero coefficient 125
7.2.2.2 Derivation of lastPosIdx 125
7.2.3 Parsing process for run mode decoding 126
7.2.4 Parsing process for level mode decoding 128
7.2.1 Update process for coeffLevel array 128
7.2.2 Forward scanning process for transform coefficients 129
7.3 VLC Tables 129
7.4 Parsing process for Exp-Golomb codes of order k 129
7.5 Parallel V2V Parsing process 130
8 Unit definition 133
8.1 Coding Tree Block (CTB) 133
8.2 Prediction unit (PU) 134
8.3 Transform unit (TU) 137
9 Motion representation 137
9.1 Motion vector prediction for rectangular partitions 137
9.2 Motion vector prediction for geometric block partitions 140
9.3 Interpolation Methods 141
9.3.1 Single pass switched Interpolation Filters with offsets (single pass SIFO) 141
9.3.2 Choice of filter set and offsets 142
9.3.3 Partition based Illumination Compensation 142
9.4 Adaptive Motion Vector Resolution 143
10 Intra-frame prediction 143
10.1 Adaptive reference sample smoothing 143
10.2 Planar prediction 143
10.3 Angular prediction 144
10.4 Arbitrary Directional Intra (ADI) 145
10.5 Edge based prediction 146
10.6 Combined Intra Prediction (CIP) 146
11 Spatial transforms 147
11.1 Large transform (16x16, 32x32, 64x64) 147
11.2 Rotational transform (ROT) 148
11.3 Mode dependent directional transforms for intra-prediction residuals 149
11.4 Transform Selection 150
11.5 Quantization 150
12 Deblocking filter 151
Luma filtering 151
Chroma filtering 152
Planar mode filtering 152
13 In-loop filtering 153
14 Entropy Coding 153
14.1 Low-Complexity Entropy Coding 153
14.2 High Coding Efficiency Entropy Coding 156
14.2.1 Novel entropy coding concept 156
14.2.2 V2V codes 157
14.2.3 Parallel V2V decoding process 158
14.2.3.1 V2V decoding for a binary decision 158
14.2.3.2 V2V decoding of partial bitstream 159
14.2.3.3 V2V Decode_Sequence 159
14.2.4 Encoding process for load balancing in parallel V2V (Informative) 161
14.2.5 Binarization and context modeling 162
14.2.6 Improved context modeling scheme for coding of transform coefficients 162
Annex A V2V codes 164
Annex B Byte stream format 165
B.1 Byte stream NAL unit syntax and semantics 165
B.1.1 Byte stream NAL unit syntax 165
B.1.2 Byte stream NAL unit semantics 165
B.2 Byte stream NAL unit decoding process 166
B.3 Decoder byte-alignment recovery (informative) 166

Annex C Hypothetical reference decoder 168
C.1 Operation of coded picture buffer (CPB) 168
C.2 Operation of the decoded picture buffer (DPB) 168
C.3 Bitstream conformance 168
C.4 Decoder conformance 168

LIST OF FIGURES

[Ed.: Figure numbering is to be made consistent throughout document, so that all figures are listed]

Figure 1: edge based prediction for intra coding. 76

Fig. 1: Illustration of recursive CTB structure (LCTB size = 128, maximum hierarchical depth = 5) 133

Figure 2: Neighboring blocks used for motion vector prediction 138

Figure 3: Reference frames and their temporal distances from the current frame 138

Figure 4 An example of angular prediction when operating the sixth row of the block with +1 pixel displacement. Projections of the pixels now fall at the 6/8 sub-pixel location (circles) in the reference row of pixels 145

Figure 5: Computation of the edge based prediction for intra coding 146

Figure 7 VLC encoding process 154

Figure 8 VLC decoding process 154

Figure 9 Update of inverse sorting table - example 155

Figure 10: Parallel V2V entropy coding 156

Figure 11: Interleaving codewords by using a first-in-first-out codeword buffer 158

LIST OF TABLES

[Ed.: Tables are to be numbered throughout document, so that all tables are listed]

1 Overview

This suggested Test Model is intended to provide a coding efficiency close to the best performing proposals in the subjective test of the CfP submissions and also to provide a complexity point that is close to that of the lowest complexity submissions shown to provide substantial improvement of coding efficiency. The inclusion of a technology in this document does not indicate a final adoption of the technology as an element of an approved test model of the committee. Rather, it indicates a preliminary selection which may require further evaluation and justification to achieve that status.

Note that the bit depth for signal processing flow elements in this design is increased relative to that for conventional 8 bit video coding design elements, although the picture buffer storage bit depth may be 8 bits when coding 8 bit source video.

2 Definitions

– **coding tree block (CTB)**: A basic unit for specifying the quadtree segmentation of the given square region. CTB can have various sizes of a square shape.

– **largest coding tree block (LCTB)**: Coding tree block of the largest size allowed in the slice. A slice consists of non-overlapped LCTBs.

– **smallest coding tree block (SCTB)**: Coding tree block of the smallest size allowed in the slice. SCTB is not allowed to be split into smaller CTBs.

– **prediction unit (PU)**: A basic unit for specifying the prediction process. The size of PU is the same as that of the CU which is not allowed to be split any more. PU can be split into multiple partitions which may have arbitrary shapes whereas CU is allowed to be split into four square shapes.

– **transform unit (TU)**: A basic unit for specifying transform and quantization process.

– **coding unit (CU)**: Same as coding tree block.

– **largest coding unit (LCU)**: Same as largest coding tree block.

– **smallest coding unit (SCU)**: Same as smallest coding tree block.

[Ed.: every CU, LCU and SCU in the text to be replaced by CTB, LCTB and SCTB]

– **smallest unit size**: Smallest size among the various allowed unit sizes of CTB, PU and TU.

– **prediction unit partition**: A basic unit for specifying the arbitrary shape region inside the PU.

– **maximum hierarchy depth**: The maximum number of allowed splitting from the LCTB.

– **PIPE**: Probability interval partitioning for entropy coding ~~(a method of entropy coding)~~

– **V2V**: Variable Length to Variable Length code

3 Conventions

NOTE – The mathematical operators used in this Specification are similar to those used in the C programming language. However, integer division and arithmetic shift operations are specifically defined. Numbering and counting conventions generally begin from 0.

3.1 Arithmetic operators

The following arithmetic operators are defined as follows:

+	Addition
−	Subtraction (as a two-argument operator) or negation (as a unary prefix operator)
*	Multiplication, including matrix multiplication
x^y	Exponentiation. Specifies x to the power of y. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation.
/	Integer division with truncation of the result toward zero. For example, 7/4 and −7/−4 are truncated to 1 and −7/4 and 7/−4 are truncated to −1.
÷	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\frac{x}{y}$	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\sum_{i=x}^{y} f(i)$	The summation of f(i) with i taking all integer values from x up to and including y.
x % y	Modulus. Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0.

3.2 Logical operators

The following logical operators are defined as follows:

x && y	Boolean logical "and" of x and y.
x || y	Boolean logical "or" of x and y.
!	Boolean logical "not".
x ? y : z	If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

3.3 Relational operators

The following relational operators are defined as follows:

>	Greater than.

>= Greater than or equal to.

< Less than.

<= Less than or equal to.

= = Equal to.

!= Not equal to.

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

3.4 Bit-wise operators

The following bit-wise operators are defined as follows:

& Bit-wise "and". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

| Bit-wise "or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

^ Bit-wise "exclusive or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

x >> y Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for positive integer values of y. Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation.

x << y Arithmetic left shift of a two's complement integer representation of x by y binary digits. This function is defined only for positive integer values of y. Bits shifted into the LSBs as a result of the left shift have a value equal to 0.

3.5 Assignment operators

The following arithmetic operators are defined as follows:

= Assignment operator.

++ Increment, i.e., x++ is equivalent to $x = x + 1$; when used in an array index, evaluates to the value of the variable prior to the increment operation.

−− Decrement, i.e., x−− is equivalent to $x = x - 1$; when used in an array index, evaluates to the value of the variable prior to the decrement operation.

+= Increment by amount specified, i.e., x += 3 is equivalent to x = x + 3, and x += (−3) is equivalent to x = x + (−3).

−= Decrement by amount specified, i.e., x −= 3 is equivalent to x = x − 3, and x −= (−3) is equivalent to x = x − (−3).

3.6 Range notation

The following notation is used to specify a range of values:

x = y..z x takes on integer values starting from y to z, inclusive, with x, y, and z being integer numbers.

3.7 Mathematical functions

The following mathematical functions are defined as follows:

$$\mathrm{Abs}(x) = \begin{cases} x & ; \quad x >= 0 \\ -x & ; \quad x < 0 \end{cases} \tag{1}$$

Ceil(x) the smallest integer greater than or equal to x. (2)

$$\text{Clip1}_Y(x) = \text{Clip3}(0, (1 << \text{BitDepth}_Y) - 1, x) \quad (3)$$

$$\text{Clip1}_C(x) = \text{Clip3}(0, (1 << \text{BitDepth}_C) - 1, x) \quad (4)$$

$$\text{Clip3}(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases} \quad (5)$$

Floor(x) the greatest integer less than or equal to x. (6)

$$\text{InverseRasterScan}(a, b, c, d, e) = \begin{cases} (a\%(d/b))*b & ; \quad e == 0 \\ (a/(d/b))*c & ; \quad e == 1 \end{cases} \quad (7)$$

Log2(x) returns the base-2 logarithm of x. (8)

Log10(x) returns the base-10 logarithm of x. (9)

$$\text{Median}(x, y, z) = x + y + z - \text{Min}(x, \text{Min}(y, z)) - \text{Max}(x, \text{Max}(y, z)) \quad (10)$$

$$\text{Min}(x, y) = \begin{cases} x & ; \quad x <= y \\ y & ; \quad x > y \end{cases} \quad (11)$$

$$\text{Max}(x, y) = \begin{cases} x & ; \quad x >= y \\ y & ; \quad x < y \end{cases} \quad (12)$$

$$\text{Round}(x) = \text{Sign}(x) * \text{Floor}(\text{Abs}(x) + 0.5) \quad (13)$$

$$\text{Sign}(x) = \begin{cases} 1 & ; \quad x >= 0 \\ -1 & ; \quad x < 0 \end{cases} \quad (14)$$

$$\text{Sqrt}(x) = \sqrt{x} \quad (15)$$

3.8 Order of operation precedence

When order of precedence in an expression is not indicated explicitly by use of parentheses, the following rules apply:

– operations of a higher precedence are evaluated before any operation of a lower precedence,

– operations of the same precedence are evaluated sequentially from left to right.

Table 3-1 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

NOTE – For those operators that are also used in the C programming language, the order of precedence used in this Specification is the same as used in the C programming language.

Table 3-1 Operation precedence from highest (at top of table) to lowest (at bottom of table)

operations (with operands x, y, and z)
"x++", "x− −"
"!x", "−x" (as a unary prefix operator)
x^y
"x * y", "x / y", "x ÷ y", "$\frac{x}{y}$", "x % y"
"x + y", "x − y" (as a two-argument operator), "$\sum_{i=x}^{y} f(i)$"
"x << y", "x >> y"
"x < y", "x <= y", "x > y", "x >= y"
"x = = y", "x != y"
"x & y"
"x | y"
"x && y"
"x | | y"
"x ? y : z"
"x = y", "x += y", "x −= y"

3.9 Variables, syntax elements, and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lower case letters with underscore characters), its one or two syntax categories, and one or two descriptors for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e., not bold) type.

In some cases the syntax tables may use the values of other variables derived from syntax elements values. Such variables appear in the syntax tables, or text, named by a mixture of lower case and upper case letter and without any underscore characters. Variables starting with an upper case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an upper case letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lower case letter are only used within the subclause in which they are derived.

In some cases, "mnemonic" names for syntax element values or variable values are used interchangeably with their numerical values. Sometimes "mnemonic" names are used without any associated numerical values. The association of values and names is specified in the text. The names are constructed from one or more groups of letters separated by an underscore character. Each group starts with an upper case letter and may contain more upper case letters.

NOTE – The syntax is described in a manner that closely follows the C-language syntactic constructs.

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in subclause [Ed. insert reference] and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in subclause 3.7) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any

underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).
A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[x, y] or as s_{yx}.
Binary notation is indicated by enclosing the string of bit values by single quote marks. For example, '01000001' represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.
Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation when the number of bits is an integer multiple of 4. For example, 0x41 represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.
Numerical values not enclosed in single quotes and not prefixed by "0x" are decimal values.
A value equal to 0 represents a FALSE condition in a test statement. The value TRUE is represented by any value different from zero.

3.10 Text description of logical operations

In the text, a statement of logical operations as would be described in pseudo-code as

```
if( condition 0 )
   statement 0
else if ( condition 1 )
   statement 1
…
else /* informative remark on remaining condition */
   statement n
```

may be described in the following manner:

... as follows / ... the following applies.

- If condition 0, statement 0
- Otherwise, if condition 1, statement 1
- …
- Otherwise (informative remark on remaining condition), statement n

Each "If ... Otherwise, if ... Otherwise, ..." statement in the text is introduced with "... as follows" or "... the following applies" immediately followed by "If ... ". The last condition of the "If ... Otherwise, if ... Otherwise, ..." is always an "Otherwise, ...". Interleaved "If ... Otherwise, if ... Otherwise, ..." statements can be identified by matching "... as follows" or "... the following applies" with the ending "Otherwise, ...".
In the text, a statement of logical operations as would be described in pseudo-code as

```
if( condition 0a    &&    condition 0b )
   statement 0
else if ( condition 1a    ||    condition 1b )
   statement 1
…
else
   statement n
```

may be described in the following manner:

... as follows / ... the following applies.

- If all of the following conditions are true, statement 0
  - condition 0a
  - condition 0b
- Otherwise, if any of the following conditions are true, statement 1
  - condition 1a

- condition 1b
- …
- Otherwise, statement n

In the text, a statement of logical operations as would be described in pseudo-code as:

```
if( condition 0 )
   statement 0
if ( condition 1 )
   statement 1
```

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

3.11 Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. All syntax elements and upper case variables that pertain to the current syntax structure and depending syntax structures are available in the process specification and invoking. A process specification may also have a lower case variable explicitly specified as the input. Each process specification has explicitly specified an output. The output is a variable that can either be an upper case variable or a lower case variable.
When invoking a process, the assignment of variables is specified as follows.

- If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.
- Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

In the specification of a process, a specific macroblock may be referred to by the variable name having a value equal to the address of the specific macroblock.

4 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships

[Ed.: Consider giving the section a shorter name]

4.1 Spatial subdivision of pictures and slices

This subclause specifies how a picture is partitioned into slices and various kinds of units. Pictures are divided into slices. A slice is a sequence of largest coding units. Location of largest coding unit is specified by the largest coding unit address **lcuAddr**.

Each coding unit including the largest coding unit is divided into four coding units recursively. It results in a quadtree segmentation of the largest coding unit. The location of a coding unit is specified by the coding unit index **cuIdx** relative to the upper-left sample of the largest coding unit.

Once the coding unit is not allowed to be split, it shall be considered as the prediction unit. Similar to the coding unit, the location of a prediction unit is specified by the prediction unit index **puIdx** relative to the upper-left sample of the largest coding unit.

The prediction unit may have multiple partitions. The location of a prediction unit partition is specified by the prediction unit partition index **puPartIdx** relative to the upper-left sample of the prediction unit.

The prediction unit may have multiple transform units. Similar to the coding unit, a transform unit may be split into four small size transform units recursively, which allows quadtree segmentation of the residue signal. Location of the transform unit is specified by the transform unit index **tuIdx** relative to the upper-left sample of the prediction unit.

Fig. 4-1 summarizes the spatial subdivision of one slice with various units and their indices.

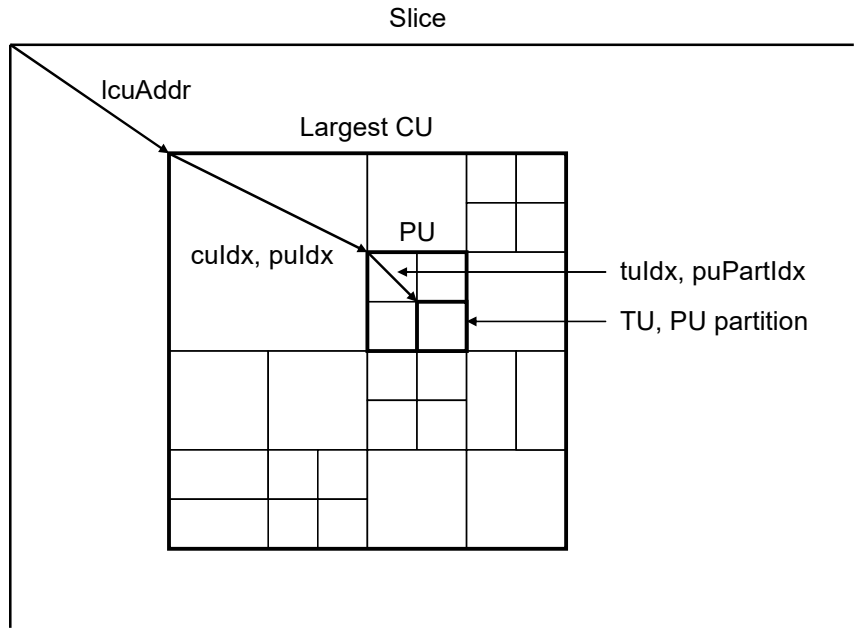


Fig. 4-1 Spatial subdivision of the slice using various units and indices (informative)

4.2 Inverse scanning processes

This subclause specifies inverse scanning processes; i.e, the mapping of indices to locations.

4.2.1 Inverse largest coding unit scanning process

Input to this process is a largest coding unit address lcuAddr.

Output of this process is the location (x, y) of the upper-left luma sample for the largest coding unit with address lcuAddr relative to the upper-left sample of the picture.

The inverse largest coding unit scanning process is specified as follows.

– x = InverseRasterScan(lcuAddr, LcuSize, LcuSize, PicWidthInSamplesN, 0)

  y = InverseRasterScan(lcuAddr, LcuSize, LcuSize, PicWidthInSamplesN, 1)

4.2.2 Recursive coding unit scanning process

Location of coding unit is specified by a raster index cuIdx defined by the smallest unit size, SuSize, as shown in Fig. 4-2.

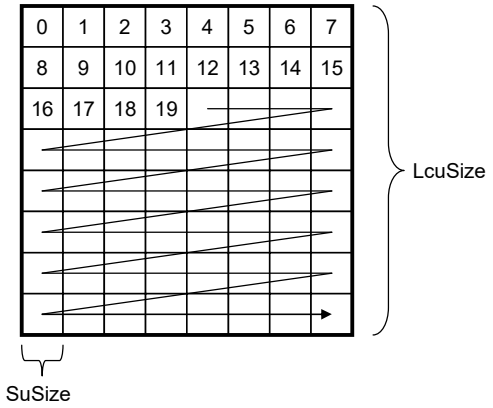


Fig. 4-2 Raster index based on smallest unit size

Coding unit of size CuSize may be split into four smaller coding units, CU_0, CU_1, CU_2 and CU_3 of size CuSize / 2 as shown in Fig. 4-3.

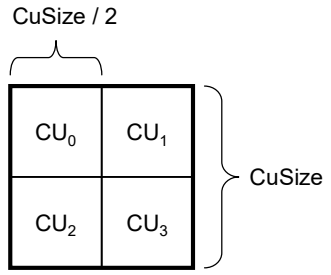


Fig. 4-3 coding unit split

For each split coding unit, the size is equal to CuSize / 2 and the split coding unit index cuIdx is specified as follows.

– cuIdx of CU_0 is assigned to be equal to cuIdx

– cuIdx of CU_1 is assigned to be equal to cuIdx + CuSizeInSu / 2

– cuIdx of CU_2 is assigned to be equal to cuIdx + CuSizeInSu / 2 * LcuSizeInSu

– cuIdx of CU_3 is assigned to be equal to cuIdx + CuSizeInSu / 2 + CuSizeInSu / 2 * LcuSizeInSu

Four each split coding unit, the splitting may be done repeatedly until the size of coding unit is equal to that of the smallest coding unit or the splitting is not allowed any more. It results in the typical z-scan or depth-first quad-tree scanning as shown in Fig. 4-4.

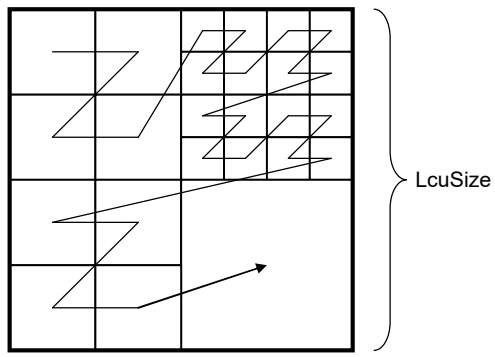


Fig. 4-4 Recursive coding unit scanning example (informative)

4.2.3 Inverse coding unit scanning process

Input to this process is a coding unit index cuIdx.

Output of this process is the location (x, y) of the upper-left luma sample for the coding unit with index cuIdx relative to the upper-left sample of the largest coding unit.

The inverse coding unit scanning process is specified as follows.

– x = InverseRasterScan(cuIdx, SuSize, SuSize, LcuSize, 0)

  y = InverseRasterScan(cuIdx, SuSize, SuSize, LcuSize, 1)

4.2.4 Inverse prediction unit, partition and transform unit scanning process

Once coding unit is decided not to be split any more, it shall be considered to be the prediction unit. Similar to coding unit, location of prediction unit is specified by puIdx defined by the smallest unit size, SuSize. However, prediction unit may be partitioned without any restrictions of a square shape. Fig. 4-5 shows the possible partitions and scans for the prediction unit. [Ed.: geometric partition to be added] The number in each partition specifies the index of the inverse prediction unit partition scan.

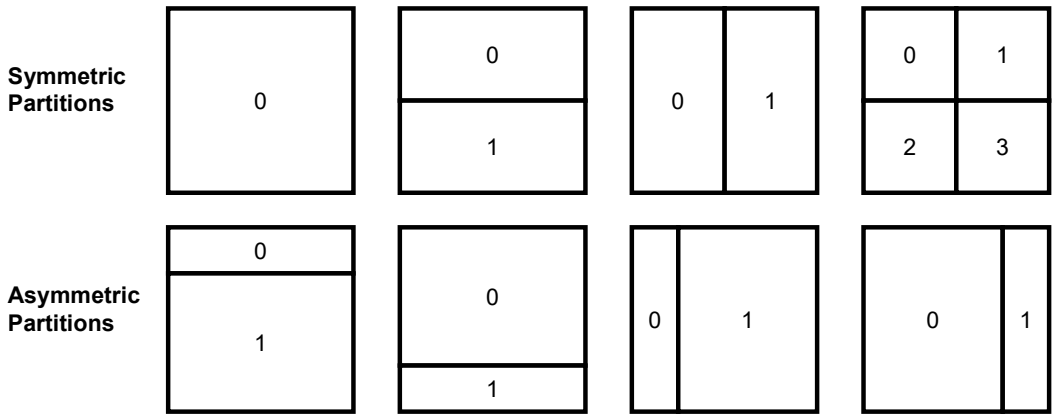


Fig. 4-5 Prediction unit partitions and scans

The functions PuPartWidth( ) and PuPartHeight( ) describing the width and height of prediction unit partitions are specified in Table 5-14 and the function PuPartOffset( ) describing the offset value for the prediction unit partition is specified in Table 5-15. PuPartWidth( ), PuPartHeight( ) and PuPartOffset( ) are set to appropriate values for each prediction unit, depending on the prediction mode and the partitioning type.

4.2.4.1 Inverse prediction unit scanning process

Input to this process is a prediction unit index puIdx.

Output of this process is the location (x, y) of the upper-left luma sample for the prediction unit with index puIdx relative to the upper-left sample of the largest coding unit.

The inverse prediction unit scanning process is specified as follows.

– x = InverseRasterScan(puIdx, SuSize, SuSize, LcuSize, 0)

  y = InverseRasterScan(puIdx, SuSize, SuSize, LcuSize, 1)

4.2.4.2 Inverse prediction unit partition scanning process

Input to this process is a prediction unit partition index puPartIdx.

Output of this process is the location (x, y) of the upper-left luma sample for the prediction unit partition with index puPartIdx relative to the upper-left sample of the prediction unit.

The inverse prediction unit partition scanning process is specified as follows.

– x = InverseRasterScan(PuPartOffset(puPartIdx), SuSize, SuSize, PuSize, 0)

  y = InverseRasterScan(PuPartOffset(puPartIdx), SuSize, SuSize, PuSize, 1)

4.2.4.3 Inverse transform unit scanning process

Input to this process is a transform unit index tuIdx.

Output of this process is the location (x, y) of the upper-left luma sample for the transform unit with index tuIdx relative to the upper-left sample of the prediction unit.

The inverse transform unit scanning process is specified as follows.

– x = InverseRasterScan(tuIdx, SuSize, SuSize, PuSize, 0)

  y = InverseRasterScan(tuIdx, SuSize, SuSize, PuSize, 1)

4.3 *Derivation processes for neighbours*

4.3.1 Derivation process of the availability for largest coding unit addresses

Input to this process is a largest coding unit address lcuAddr.

Output of this process is the availability of the lcuAddr.

The largest coding unit is marked as available, unless any of the following conditions is true in which case the largest coding unit is marked as not available:

– lcuAddr < 0

– lcuAddr > CurrLcuAddr

– the largest coding unit with address lcuAddr belongs to a different slice than the largest coding unit with address CurrLcuAddr

4.3.2 Derivation process for neighbouring largest coding unit addresses and their availability

The outputs of this process are

– lcuAddrA: the address and availability status of the largest coding unit to the left of the current largest coding unit

– lcuAddrB: the address and availability status of the largest coding unit above the current largest coding unit

– lcuAddrC: the address and availability status of the largest coding unit above-right of the current largest coding unit

– lcuAddrD: the address and availability status of the largest coding unit above-left of the current largest coding unit

– lcuAddrE: the address and availability status of the largest coding unit below-left of the current largest coding unit

Fig. 4-6 shows the relative spatial locations of the largest coding units with lcuAddrA, lcuAddrB, lcuAddrC, lcuAdddrD and lcuAddrE relative to the current largest coding unit with CurrLcuAddr.

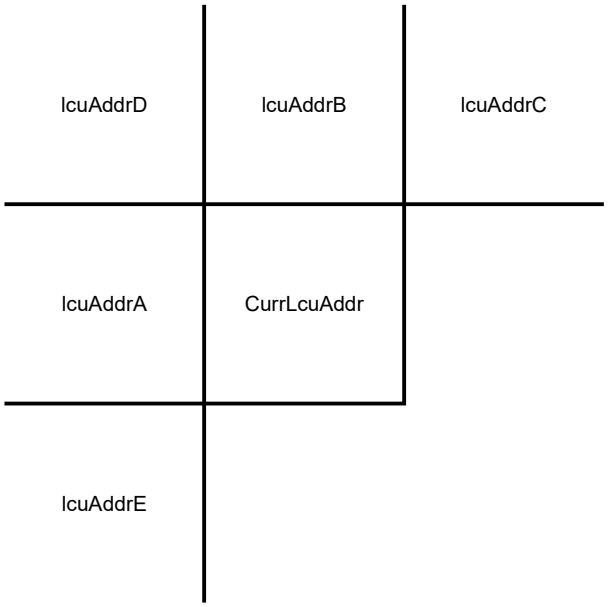


Fig. 4-6 Neighbouring largest coding units for a given largest coding unit

Input to the process in subclause 4.3.1 is lcuAddrA = CurrlcuAddr - 1 and the output is whether the largest coding unit lcuAddrA is available. In addition, lcuAddrA is marked as not available when CurrLcuAddr % PicWidthInLCUs is equal to 0.

Input the process in subclause 4.3.1 is lcuAddrB = CurrLcuAddr - PicWidthInLCUs and the output is whether the largest coding unit lcuAddrB is available.

Input the process in subclause 4.3.1 is lcuAddrC = CurrLcuAddr - PicWidthInLCUs + 1 and the output is whether the largest coding unit lcuAddrC is available. In addition, lcuAddrC is marked as not available when (CurrLcuAddr + 1) % PicWidthInLCUs is equal to 0.

Input the process in subclause 4.3.1 is lcuAddrD = CurrLcuAddr - PicWidthInLCUs − 1 and the output is whether the largest coding unit lcuAddrD is available. In addition, lcuAddrD is marked as not available when CurrLcuAddr % PicWidthInLCUs is equal to 0.

Input the process in subclause 4.3.1 is lcuAddrE = CurrLcuAddr + PicWidthInLCUs − 1 and the output is whether the largest coding unit lcuAddrE is available. In addition, lcuAddrE is marked as not available when CurrLcuAddr % PicWidthInLCUs is equal to 0.

4.3.3 Derivation process of the availability for coding units

Input to this process is a coding unit index cuIdx and the corresponding largest coding unit address lcuAddr.

Output of this process is the availability of the coding unit.

The coding unit is marked as available, unless any of the following conditions is true in which case the coding unit marked as not available:

– lcuAddr < 0

– lcuAddr > CurrLcuAddr

– the LCU address lcuAddr belongs to a different slice than the current slice

– ConvertRasterToZscan(cuIdx) > ConvertRasterToZscan(CurrCuIdx)

4.3.4 Derivation process for neighbouring units and partitions

Subclause 4.3.4.1 specifies the derivation process for neighbouring largest coding units.

Subclause 4.3.4.2 specifies the derivation process for neighbouring coding units.

Subcaluse 4.3.4.3 specifies the derivation process for neighbouring prediction units.

Subcaluse 4.3.4.4 specifies the derivation process for neighbouring partitions.

Table 4-1 specifies the values for the difference of luma location (xD, yD) for the input and the replacement for N in lcuAddrN, cuIdxN, puIdxN and puPartIdxN for the output. These input and output assignments are used in subclauses 4.3.4.1 to 4.3.4.4. The variables PuPartWidth and PuPartHeight are specified when Table 5-14 is referred to.

Table 4-1 Specification of input and output assignments for subclauses 4.3.4 to 4.3.5

N	xD	yD
A	−1	0
B	0	-1
C	PuPartWidth	-1
D	-1	-1
E	-1	PuPartHeight

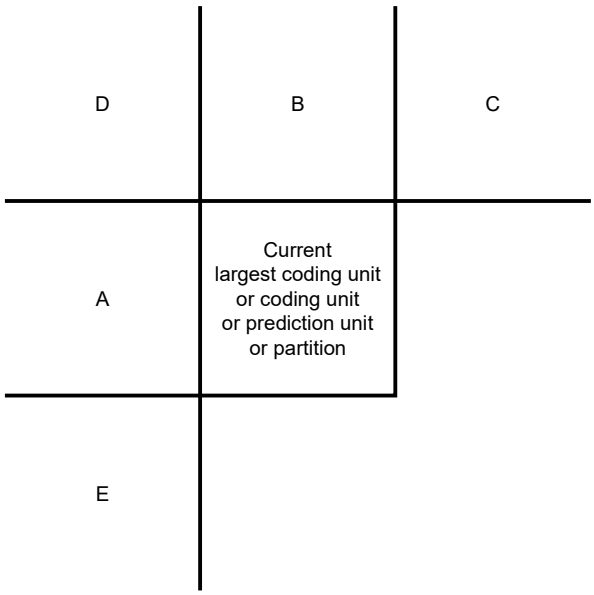


Fig. 4-7 Determination of the neighbouring largest coding units, coding units, prediction units and partitions (informative)

4.3.4.1 Derivation process for neighbouring largest coding units

Input to this process is a largest coding unit index lcuAddr.

The outputs of this process are

– lcuAddrA: the address and its availability status of the largest coding unit to the left of the current largest coding unit

– lcuAddrB: the address and its availability status of the largest coding unit above the current largest coding unit

lcuAddN (with N being A or B) are derived as follows.

– The difference of luma location (xD, yD) is set according to Table 4-1.

– The derivation process for neighbouring locations as specified in subclause 4.3.5 is invoked for luma locations with (xN, yN) equal to (xD, yD), and the output is assigned to lcuAddrN.

4.3.4.2 Derivation process for neighbouring coding units

Input to this process is a largest coding unit index lcuAddr and a coding unit index cuIdx.

The outputs of this process are

– lcuAddrA: the address and its availability status of the largest coding unit to the left of the current largest coding unit

– cuIdxA: the index and its availability status of the coding unit to the left of the current coding unit

– lcuAddrB: the address and its availability status of the largest coding unit above the current largest coding unit

– cuIdxB: the index and its availability status of the coding unit above the current coding unit

lcuAddrN and cuIdxN (with N being A or B) are derived as follows.

– The difference of luma location (xD, yD) is set according to Table 4-1.

– Input to the process in subclause 4.2.3 is cuIdxN and the output is the luma location (x, y).

– The luma location (xN, yN) is specified by

$$xN = x + xD$$

$$yN = y + yD$$

– The derivation process for neighbouring locations as specified in subclause 4.3.5 is invoked for luma locations with (xN, yN) as the input and the output is assigned to lcuAddrN and (xW, yW).

– The variable cuIdxN is derived as follows.

  - If lcuAddrN is not available, cuIdxN is marked as not available.

  - Otherwise (lcuAddrN is available), the coding unit index in the largest coding unit lcuAddrN covering the luma location (xW, yW) shall be assigned to cuIdxN.

4.3.4.3 Derivation process for neighbouring prediction units

Input to this process is a largest coding unit index lcuAddr and a prediction unit index puIdx.

The outputs of this process are

– lcuAddrA: the address and its availability status of the largest coding unit to the left of the current largest coding unit

– puIdxA: the index and its availability status of the prediction unit to the left of the current prediction unit

– lcuAddrB: the address and its availability status of the largest coding unit above the current largest coding unit

– puIdxB: the index and its availability status of the prediction unit above the current prediction unit

– lcuAddrC: the address and its availability status of the largest coding unit above-right of the current largest coding unit

– puIdxC: the index and availability status of the prediction unit above-right of the current prediction unit

– lcuAddrD: the address and its availability status of the largest coding unit above-left of the current largest coding unit

– puIdxD: the index and availability status of the prediction unit above-left of the current prediction unit

– lcuAddrE: the address and its availability status of the largest coding unit below-left of the current largest coding unit

– puIdxE: the index and availability status of the largest coding unit below-left of the current prediction unit

lcuAddrN and puIdxN (with N being A, B, C, D or E) are derived as follows.

– The difference of luma location (xD, yD) is set according to Table 4-1.

– Input to the process in subclause 4.2.4.1 is puIdxN and the output is the luma location (x, y).

– The luma location (xN, yN) is specified by

$$xN = x + xD$$

$$yN = y + yD$$

– The derivation process for neighbouring locations as specified in subclause 4.3.5 is invoked for luma locations with (xN, yN) as the input and the output is assigned to lcuAddrN and (xW, yW).

– The variable puIdxN is derived as follows.

  - If lcuAddrN is not available, puIdxN is marked as not available.

  - Otherwise (lcuAddrN is available), the prediction unit index in the largest coding unit lcuAddN covering the luma location (xW, yW) shall be assigned to puIdxN.

4.3.4.4 Derivation process for neighbouring partitions

Input to this process is a largest coding unit index lcuAddr, a prediction unit index puIdx and a partition index puPartIdx.

The outputs of this process are

– lcuAddrA: the address and its availability status of the largest coding unit to the left of the current largest coding unit

– puIdxA: the index and its availability status of the prediction unit to the left of the current prediction unit

– puPartIdxA: the index and its availability status of the partition index to the left of the current partition index

– lcuAddrB: the address and its availability status of the largest coding unit above the current largest coding unit

– puIdxB: the index and its availability status of the prediction unit above the current prediction unit

– puPartIdxB: the index and its availability status of the partition index above the current partition index

– lcuAddrC: the address and its availability status of the largest coding unit above-right of the current largest coding unit

– puIdxC: the index and availability status of the prediction unit above-right of the current prediction unit

– puPartIdxC: the index and its availability status of the partition index above-right of the current partition index

– lcuAddrD: the address and its availability status of the largest coding unit above-left of the current largest coding unit

– puIdxD: the index and availability status of the prediction unit above-left of the current prediction unit

– puPartIdxD: the index and its availability status of the partition index above-left of the current partition index

– lcuAddrE: the address and its availability status of the largest coding unit below-left of the current largest coding unit

– puIdxE: the index and availability status of the largest coding unit below-left of the current prediction unit

– puPartIdxE: the index and its availability status of the partition index below-left of the current partition index

lcuAddrN, puIdxN and puPartIdxN (with N being A, B, C, D or E) are derived as follows.

– The difference of luma location (xD, yD) is set according to Table 4-1.

– Input to the process in subclause 4.2.4.1 is puIdxN and the output is the luma location (x, y).

– Input to the process in subclause 4.2.4.2 is puPartIdxN and the output is the difference of luma location (xP, yP)

– The luma location (xN, yN) is specified by

  xN = x + xP + xD

  yN = y + yP + yD

– The derivation process for neighbouring locations as specified in subclause 4.3.5 is invoked for luma locations with (xN, yN) as the input and the output is assigned to lcuAddrN and (xW, yW).

– The variable puIdxN and puPartIdxN are derived as follows.

  – If lcuAddrN is not available, puIdxN and puPartIdxN are marked as not available.

  – Otherwise (lcuAddrN is available), the prediction unit index and the partition index in the largest coding unit lcuAddN covering the luma location (xW, yW) shall be assigned to puIdxN and partIdxN.

4.3.5 Derivation process for neighbouring locations

Input to this process is a luma or chroma location (xN, yN) expressed relative to the upper left corner of the current largest coding unit.

Outputs of this process are

– lcuAddrN: either equal to CurrLcuAddr or to the address of neighbouring largest coding unit that containts (xN, yN) and its availability status,

– (xW, yW): the location (xN, yN) expressed relative to the upper-left corner of the largest coding unit lcuAddrN (rather than relative to the upper-left corner of the current largest coding unit).

Let maxW and maxH be variables specifying maximum values of the location components xN, xW and yN, yW, respectively. maxW and maxH are derived as follows.

– if this process is invoked for neighbouring luma locations,

maxW = maxH = LcuSize

– Otherwise (this process is invoked for neighbouring chroma locations),

maxW = maxH = LcuSizeC

The derivation process for neighbouring largest coding unit addresses and their availability in subclause 4.3.2 is invoked with lcuAddrA, lcuAddrB, lcuAddrC, lduAddrD and lcuAddrE as well as their availability status as the output.

Table 4-2 specifies lcuAddrN depending on (xN, yN).

Table 4-2 Specification of lcuAddrN

xN	**yN**	**lcuAddrN**
< 0	< 0	lcuAddrD
< 0	0..maxH − 1	lcuAddrA
0..maxW − 1	< 0	lcuAddrB
0..maxW − 1	0..maxH − 1	CurrLcuAddr
> maxW − 1	< 0	lcuAddrC
> maxW − 1	0..maxH − 1	not available
< 0	> maxH – 1	lcuAddrE
0..maxW – 1	> maxH − 1	not available

The neighbouring location (xW, yW) relative to the upper-left corner of the largest coding unit lcuAddrN is derived as

xW = (xN + maxW) % maxW

yW = (yN + maxH) % maxH

5 Syntax and Semantics

5.1 Syntax

[Ed.: preliminary draft]

5.1.1 NAL unit syntax

nal_unit(NumBytesInNALunit) {	C	Descriptor
forbidden_zero_bit	All	f(1)
nal_ref_idc	All	u(2)
nal_unit_type	All	u(5)
NumBytesInRBSP = 0		
nalUnitHeaderBytes = 1		
if(nal_unit_type = = 1 | | nal_unit_type = = 5) {		
temporal_id	**All**	**u(3)**
output_flag	**All**	**u(1)**
reserved_zero_4bits	**All**	**u(4)**
nalUnitHeaderBytes += 1		
}		
for(i = nalUnitHeaderBytes; i < NumBytesInNALunit; i++) {		
if(i + 2 < NumBytesInNALunit && next_bits(24) = = 0x000003) {		
rbsp_byte[NumBytesInRBSP++ **]**	All	b(8)
rbsp_byte[NumBytesInRBSP++ **]**	All	b(8)
i += 2		
emulation_prevention_three_byte /* equal to 0x03 */	All	f(8)
} else		
rbsp_byte[NumBytesInRBSP++ **]**	All	b(8)
}		
}		

z

5.1.2 Sequence parameter set RBSP syntax

[Ed.: Parameters to be removed and/or added as appropriate.]

seq_parameter_set_rbsp( ) {	C	Descriptor
profile_idc	0	u(8)
reserved_zero_8bits /* equal to 0 */	0	u(8)
level_idc	0	u(8)
seq_parameter_set_id	0	ue(v)
bit_depth_luma_minus8	0	ue(v)
bit_depth_chroma_minus8	0	ue(v)
increased_bit_depth_luma	0	ue(v)
ine_bit_depth_chroma	0	ue(v)
log2_max_frame_num_minus4	0	ue(v)
pic_order_cnt_type	0	ue(v)
if(pic_order_cnt_type == 0)		
log2_max_pic_order_cnt_lsb_minus4	0	ue(v)
else if(pic_order_cnt_type == 1) {		
delta_pic_order_always_zero_flag	0	u(1)
offset_for_non_ref_pic	0	se(v)
num_ref_frames_in_pic_order_cnt_cycle	0	ue(v)
for(i = 0; i < num_ref_frames_in_pic_order_cnt_cycle; i++)		
offset_for_ref_frame[i]	0	se(v)
}		
max_num_ref_frames	0	ue(v)
gaps_in_frame_num_value_allowed_flag	0	u(1)
max_coding_unit_width	0	ue(v)
max_coding_unit_height	0	ue(v)
max_coding_unit_hierarchy_depth_correct	0	u(1)
log2_min_transform_unit_size_minus2	0	ue(v)
if(log2_min_transform_unit_size_minus2 < 4)		
max_transform_unit_hierarchy_depth	0	ue(v)
pic_height_in_luma_samples	0	u(16)
numExtraFilters	0	ue(v)
for(i=0; i< numExtraFilters; i++){		
log2_filterCoeffPrecision	0	ue(v)
halfNumTap	0	ue(v)
for(j=0;j<(3*halfNumTap); j++){		
filterCoef[i][j]	0	i(v)
}		
}		
adaptive_intra_smoothing_flag	0	u(1)
inter_merge_flag	0	u(1)
interleaved_motion_vector_predictor_flag	0	u(1)
pbic_flag	0	u(1)
rbsp_trailing_bits( )	0	
}		

5.1.3 Picture parameter set RBSP syntax

[Ed.: Parameters to be removed and/or added as appropriate.]

pic_parameter_set_rbsp( ) {	C	Descriptor
pic_parameter_set_id	1	ue(v)
seq_parameter_set_id	1	ue(v)
entropy_coding_mode_flag	1	u(1)
num_ref_idx_l0_default_active_minus1	1	ue(v)
num_ref_idx_l1_default_active_minus1	1	ue(v)
pic_init_qp_minus26 /* relative to 26 */	1	se(v)
constrained_intra_pred_flag	1	u(1)
for(i=0;i<15; i++){		
numAllowedFilters[i]	1	ue(v)
for(j=0;j<numAllowedFilters;j++){		
filtIdx[i][j]	1	ue(v)
}		
}		
rbsp_trailing_bits( )	1	
}		

5.1.4 Supplemental enhancement information RBSP syntax

sei_rbsp( ) {	C	Descriptor
do		
sei_message( )	5	
while(more_rbsp_data( ))		
rbsp_trailing_bits( )	5	
}		

5.1.5 Supplemental enhancement information message syntax

sei_message( ) {	C	Descriptor
payloadType = 0		
while(next_bits(8) == 0xFF) {		
ff_byte /* equal to 0xFF */	5	f(8)
payloadType += 255		
}		
last_payload_type_byte	5	u(8)
payloadType += last_payload_type_byte		
payloadSize = 0		
while(next_bits(8) == 0xFF) {		
ff_byte /* equal to 0xFF */	5	f(8)
payloadSize += 255		
}		
last_payload_size_byte	5	u(8)
payloadSize += last_payload_size_byte		
sei_payload(payloadType, payloadSize)	5	
}		

5.1.6 Access unit delimiter RBSP syntax

access_unit_delimiter_rbsp( ) {	C	Descriptor
primary_pic_type	6	u(3)
rbsp_trailing_bits( )	6	
}		

5.1.7 Filler data RBSP syntax

filler_data_rbsp( ) {	C	Descriptor
while(next_bits(8) == 0xFF)		
ff_byte /* equal to 0xFF */	9	f(8)
rbsp_trailing_bits( )	9	
}		

5.1.8 Slice layer RBSP syntax

slice_layer_rbsp( ) {	C	Descriptor
slice_header( )	2	
slice_data( ) /* all categories of slice_data( ) syntax */	2 | 3 | 4	
rbsp_slice_trailing_bits( )	2	
}		

5.1.9 RBSP slice trailing bits syntax

rbsp_slice_trailing_bits( ) {	C	Descriptor
rbsp_trailing_bits( )	All	
if(entropy_coding_mode_flag)		
while(more_rbsp_trailing_data( ))		
cabac_zero_word /* equal to 0x0000 */	All	f(16)
}		

5.1.10 RBSP trailing bits syntax

rbsp_trailing_bits( ) {	C	Descriptor
rbsp_stop_one_bit /* equal to 1 */	All	f(1)
while(!byte_aligned( ))		
rbsp_alignment_zero_bit /* equal to 0 */	All	f(1)
}		

5.1.11 Slice header syntax

[Ed.: Parameters to be removed and/or added as appropriate.]

slice_header( ) {	C	Descriptor
first_lctb_in_slice	2	ue(v)
slice_type	2	ue(v)
pic_parameter_set_id	2	ue(v)
frame_num	2	u(v)
if(IdrPicFlag)		
idr_pic_id	2	ue(v)
pic_order_cnt_lsb	2	u(v)
if(slice_type == P || slice_type == B) {		
num_ref_idx_active_override_flag	2	u(1)
if(num_ref_idx_active_override_flag) {		
num_ref_idx_l0_active_minus1	2	ue(v)
if(slice_type == B)		
num_ref_idx_l1_active_minus1	2	ue(v)
}		
}		
ref_pic_list_modification( )		
if(nal_ref_idc != 0)		
dec_ref_pic_marking( )	2	
if(entropy_coding_mode_flag) {		
pipe_multi_codeword_flag	2	u(1)
if(!pipe_multi_codeword_flag)		
pipe_max_delay_shift_6	2	ue(v)
else		
balanced_cpus	2	u(8)
if(slice_type != I)		
cabac_init_idc	2	ue(v)
}		
slice_qp_delta	2	se(v)
alf_param()		
if(slice_type == P || slice_type == B) {		
mc_interpolation_idc	2	ue(v)
mv_competition_flag	2	u(1)
if (mv_competition_flag) {		
mv_competition_temporal_flag	2	u(1)
}		
}		
if (slice_type == B && mv_competition_flag)		
collocated_from_l0_flag	2	u(1)
sifo_param()		
~~if (entropy_coding_mode_flag == 3)~~		
~~parallel_v2v_header()~~	~~2~~	
edge_based_prediction_flag	2	u(1)

if(edge_prediction_ipd_flag = = 1)		
threshold_edge	2	u(8)
}		

5.1.11.1 Reference picture list modification syntax

ref_pic_list_modification( ) {	C	Descriptor
if(slice_type % 5 != 2) {		
ref_pic_list_modification_flag_l0	2	u(1)
if(ref_pic_list_modification_flag_l0)		
do {		
modification_of_pic_nums_idc	2	u(1)
if(modification_of_pic_nums_idc = = 0 | | modification_of_pic_nums_idc = = 1)		
abs_diff_pic_num_minus1	2	ue(v)
else (modification_of_pic_nums_idc = = 2)		
long_term_pic_num	2	ue(v)
} while(modification_of_pic_nums_idc != 3)		
}		
if(slice_type % 5 = = 1) {		
ref_pic_list_modification_flag_l1	2	u(1)
if(ref_pic_list_modification_flag_l1)		
do {		
modification_of_pic_nums_idc	2	u(1)
if(modification_of_pic_nums_idc = = 0 | | modification_of_pic_nums_idc = = 1)		
abs_diff_pic_num_minus1	2	ue(v)
else if(modification_of_pic_nums_idc = = 2)		
long_term_pic_num	2	ue(v)
} while(modification_of_pic_nums_idc != 3)		
}		
}		

5.1.11.2 Decoded reference picture marking syntax

dec_ref_pic_marking( ) {	C	Descriptor
if(IdrPicFlag) {		
no_output_of_prior_pics_flag	2 | 5	u(1)
long_term_reference_flag	2 | 5	u(1)
} else {		
adaptive_ref_pic_marking_mode_flag	2 | 5	u(1)
if(adaptive_ref_pic_marking_mode_flag)		
do {		
memory_management_control_operation	2 | 5	ue(v)
if(memory_management_control_operation = = 1 | | memory_management_control_operation = = 3)		
difference_of_pic_nums_minus1	2 | 5	ue(v)
if(memory_management_control_operation = = 2)		
long_term_pic_num	2 | 5	ue(v)
if(memory_management_control_operation = = 3 | | memory_management_control_operation = = 6)		
long_term_frame_idx	2 | 5	ue(v)

if(memory_management_control_operation == 4)		
max_long_term_frame_idx_plus1	2 | 5	ue(v)
} while(memory_management_control_operation != 0)		
}		
}		

5.1.11.3 Adaptive loop filter parameter syntax

[3-Input-ALF is still missing]

alf_param() {	C	Descriptor
adaptive_loop_filter_flag	2	u(1)
if (adaptive_loop_filter_flag) {		
alf_length_luma_~~hor_minus_5~~minus5_div2	2	ue(v)
~~**alf_length_luma_ver_minus_5_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_luma_pred_hor_minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_luma_pred_ver_minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_luma_qpe_hor minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_luma_qpe_ver minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_quant_step_size_luma_hor**~~	~~2~~	~~ue(v)~~
~~**alf_quant_step_size_luma_ver**~~	~~2~~	~~ue(v)~~
~~**alf_minKStart_minus1**~~	~~2~~	~~ue(v)~~
~~for (i=0; i < AlfmaxDepth; i++) {~~		
~~**alf_golombIndexBit**~~	~~2~~	~~u(1)~~
~~}~~		
alf_noFilters_minus1	2	ue(v)
if (alf_noFilters_minus1 == 1) {		
alf_startSecondFilter	2	ue(v)
}		
else if (alf_noFilters_minus1 > 1) {		
for (i=0; i< AlfMaxNumFilters; i++) {		
alf_filter_pattern[i]	2	u(1)
}		
}		
if (AlfNumFilters > 1) {		
alf_predMethod	2	u(1)
}		
alf_minKStart_minus1	2	ue(v)
for (i=0; i~~< AlfNumFilters~~ < AlfmaxDepth; i++) {		
~~for (j=0; j< AlfLengthLuma_hor; j++) {~~		
~~scanPos = DepthTable[j]-1;~~		
~~**alf_coeff_luma_hor**[i][j] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~for (j=0; j< AlfLengthLuma_pred_hor; j++) {~~		
~~scanPos = DepthTable[j]-1;~~		
~~**alf_coeff_luma_pred_hor**[i][j] = golombDecodek (kMinTap[scanPos])~~ **alf_golombIndexBit**	2	~~ge(v~~u(1)
}		
for (~~j~~i=0; ~~j< AlfLengthLuma_qpe_hor; j~~i< AlfNumFilters; i++) {		
~~scanPos = DepthTable[j]-1;~~		
~~**alf_coeff_luma_qpe_hor**[i][j] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
for (j=0; j< ~~AlfLengthLuma_ver~~AlfNumCoeffLuma; j++) {		
scanPos = DepthTable[j]-1;		
alf_coeff_luma~~_ver~~ [i][j] = golombDecodek (kMinTap[scanPos])		ge(v)
}		

for (j=0; j< AlfLengthLuma_pred_ver; j++) { }		
scanPos = DepthTable[j] 1;		
alf_coeff_luma_pred_ver[i][j] = golombDecodek (kMinTap[scanPos])		ge(v)
}		
for (j=0; j< AlfLengthLuma_qpe_ver; j++) {		
scanPos = DepthTable[j] 1;		
alf_coeff_luma_qpe_ver[i][j] = golombDecodek (kMinTap[scanPos])		ge(v)
}		
}		
alf_chroma_idc	2	ue(v)
if (alf_chroma_idc) {		
alf_reuse_coefficients_for_chroma_flag	2	u(1)
if (alf_reuse_coefficients_for_chroma_flag)		
{		
alf_length_chroma_hor_minus_5minus5_div2 = alf_length_luma_hor_minus_5_div2	2	ue(v)
alf_length_chroma_ver minus_5_div2 = alf_length_luma_ver minus_5_div2		
alf_length_chroma_pred_hor minus_1_div2 = alf_length_luma_pred_hor minus_1_div2		
alf_length_chroma_pred_ver_minus_1_div2 = alf_length_luma_pred_ver_minus_1_div2		
alf_length_chroma_qpe_hor_minus_1_div2 = alf_length_luma_qpe_hor_minus_1_div2		
alf_length_chroma_qpe_ver minus_1_div2 = alf_length_luma_qpe_ver minus_1_div2		
alf_quant_step_size_chroma_hor = alf_quant_step_size_luma_hor		
alf_quant_step_size_chroma_ver = alf_quant_step_size_luma_ver		
for(i = 0; i< AlfLengthChroma_horAlfNumCoeffChroma; i++) {		
alf_coeff_chroma_hor [i] = alf_coeff_luma_hor [i]	2	se(v)
}		
for(i = 0; i< AlfLengthChroma_pred_hor; i++) { }		
alf_coeff_chroma_pred_hor [i] = alf_coeff_luma_pred_hor [i]		
}		
for(i = 0; i< AlfLengthChroma_qpe_hor; i++) {		
alf_coeff_chroma_qpe_hor [i] = alf_coeff_luma_qpe_hor [i]		
}		
for(i = 0; i< AlfLengthChroma_ver; i++) {		
alf_coeff_chroma_ver [i] = alf_coeff_luma_ver [i]		
}		
for(i = 0; i< AlfLengthChroma_pred_ver; i++) {		
alf_coeff_chroma_pred_ver [i]= alf_coeff_luma_pred_ver [i]		
}		
for(i = 0; i< AlfLengthChroma_qpe_ver; i++) {		
alf_coeff_chroma_qpe_ver [i] = alf_coeff_luma_qpe_ver [i]		
}		

~~} else {~~		
~~**alf_length_chroma_hor_minus_5_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_chroma_ver minus_5_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_chroma_pred_hor minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_chroma_pred_ver_minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_chroma_qpe_hor_minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_length_chroma_qpe_ver minus_1_div2**~~	~~2~~	~~ue(v)~~
~~**alf_quant_step_size_chroma_hor**~~	~~2~~	~~ue(v)~~
~~**alf_quant_step_size_chroma_ver**~~	~~2~~	~~ue(v)~~
~~for(i = 0; i< AlfLengthChroma_hor; i++) {~~		
~~scanPos = DepthTable[j] 1;~~		
~~**alf_coeff_chroma_hor** [i] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~for(i = 0; i< AlfLengthChroma_pred_hor; i++) {~~		
~~scanPos = DepthTable[j] 1;~~		
~~**alf_coeff_chroma_pred_hor** [i] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~for(i = 0; i< AlfLengthChroma_qpe_hor; i++) {~~		
~~scanPos = DepthTable[j]-1;~~		
~~**alf_coeff_chroma_qpe_hor** [i] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~for(i = 0; i< AlfLengthChroma_ver; i++) {~~		
~~scanPos = DepthTable[j] 1;~~		
~~**alf_coeff_chroma_ver** [i] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~for(i = 0; i< AlfLengthChroma_pred_ver; i++) {~~		
~~scanPos = DepthTable[j] 1;~~		
~~**alf_coeff_chroma_pred_ver** [i] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~for(i = 0; i< AlfLengthChroma_qpe_ver; i++) {~~		
~~scanPos = DepthTable[j] 1;~~		
~~**alf_coeff_chroma_qpe_ver** [i] = golombDecodek (kMinTap[scanPos])~~		~~ge(v)~~
~~}~~		
~~}~~		
~~}~~		
alf_cu_control_flag	2	u(1)
if (alf_cu_control_flag) {		
alf_cu_control_max_depth	2	ue(v)
if (alf_cu_control_max_depth) {		
alf_length_cu_control_info	2	u(v)
}		
for(i = 0; i< NumAlfCuFlag; i++)		
alf_cu_flag [i]	2	u(1) | ae(v)
}		

}		
}		

5.1.11.4 Switched Filter with Offsets parameter syntax

sifo_param() {	C	Descriptor
if (slice_type == P || slice_type == B)		
predict_filter_flag	2	u(1)
if (slice_type == P) {		
if (!predict_filter_flag) {		
sifo_type	2	u(2)
if(sifo_type == 0 || sifo_type ==1) {		
for(i=0; i<15; i++)		
sifo_filter[i]	2	u(v)
}		
if(sifo_type==2) {		
sifo_filter[0]	2	u(v)
for(i=1; i<15; i++)		
sifo_filter[i] = sifo_filter[0]		
}	2	u(v)
} else {		
sifo_type	2	u(1)
if(sifo_type == 0) {		
for(i=0; i<15; i++) {		
use_prev_filter_flag	2	u(1)
if(use_prev_filter_flag)		
sifo_filter[i] = prev_filter[i]		
else		
sifo_filter[i]	2	u(v)
}		
} else {		
sifo_filter[0]	2	u(v)
for(i=1; i<15; i++)		
sifo_filter[i] = sifo_filter[0]		
}		
}		
}		
if (slice_type == B) {		
sifo_type	2	u(1)
if(sifo_type==1){		
sifo_filter[0]	2	u(v)
for(i=1; i<15; i++)		
sifo_filter[i] = sifo_filter[0]		
}		
if(sifo_type==0){		
if (!predict_filter_flag){		

for(i=0; i<15; i++)		
sifo_filter[i]	2	u(v)
} else {		
use_prev_filter_flag	2	u(1)
if(use_prev_filter_flag)		
sifo_filter[i] = prev_filter[i]		
else		
sifo_filter[i]	2	u(v)
}		
}		
}		
if (slice_type == P || slice_type == B) {		
zero_offset_flag	2	u(1)
if(! zero_offset_flag) {		
for(i = 0; i <= num_ref_idx_l0_active_minus1; i++) {		
if(i==0){		
for(j=0; j<16; j++) {		
sifo_offset_l0[i]	2	se(v)
sifo_offset[i]_l0 *= (1 << increased_bit_depth_luma)		
}		
} else {		
img_offset_l0[i]	2	se(v)
img_offset_l0[i] *= (1 << increased_bit_depth_luma)		
}		
}		
if (slice_type == B) {		
for(i = 0; i <= num_ref_idx_l1_active_minus1; i++) {		
if(i==0){		
for(j=0; j<16; j++) {		
sifo_offset_l1[i]	2	se(v)
sifo_offset_l1[i] *= (1 << increased_bit_depth_luma)		
}		
} else {		
img_offset_l1[i]	2	se(v)
img_offset_l1[i] *= (1 << increased_bit_depth_luma)		
}		
}		
}		
}		
}		

5.1.12 Slice data syntax

[Ed. preliminary draft]

slice_data( ) {	C	Descriptor
if(entropy_coding_mode_flag && pipe_multi_codeword_flag) {		
pipe_partition_start[11] = currBitStreamPointer [Ed. define]		
for(partIdx = 11; partIdx > 0; partIdx--) {		
pipe_partition_size(partIdx)		
pipe_partition_start[partIdx – 1] = pipe_partition_start[partIdx] + partSize[partIdx]		
}		
for(i=0; i<balanced_cpus-1; i++)		
balanced_offsets[i]		u(8)
}		
CurrMbAddr = first_lctb_in_slice		
moreDataFlag = 1		
do {		
xCU = HorLumaLocation(CurrMbAddr)		
yCU = VerLumaLocation(CurrMbAddr)		
coding_unit(xCU, yCU, MaxCodingUnitSize)	2 | 3 | 4	
if(!entropy_coding_mode_flag)		
moreDataFlag = more_rbsp_data( )		
else {		
end_of_slice_flag	2	ae(v)
moreDataFlag = !end_of_slice_flag		
}		
CurrMbAddr = NextMbAddress(CurrMbAddr)		
} while(moreDataFlag)		
}		

[Ed. preliminary draft]

pipe_partition_size(pIdx) {	C	Descriptor
part_size_byte_one	2	u(8)
partSize[pIdx] = part_size_byte_one		
if((partSize[pIdx] & 1) == 0)		
partSize[pIdx] >>= 1		
else {		
partSize[pIdx] >>= 1		
part_size_byte_two	2	u(8)
partSize[pIdx] += (part_size_byte_two << 7)		
if((partSize[pIdx] & 1) == 0)		
partSize[pIdx] = 128 + (partSize[pIdx] >> 1)		
else {		
partSize[pIdx] >>= 1		
part_size_byte_three	2	u(8)
partSize[pIdx] += (part_size_byte_two << 14)		
if((partSize[pIdx] & 1) == 0)		
partSize[pIdx] = 16 512 + (partSize[pIdx] >> 1)		
else {		
partSize[pIdx] >>= 1		
part_size_byte_four	2	u(8)
partSize[pIdx] += (part_size_byte_two << 21)		
partSize[pIdx] = 2 113 664		
}		
}		
}		
}		

5.1.13 Coding unit syntax

[Ed. preliminary draft]

coding_unit(x0, y0, currCodingUnitSize) {	C	Descriptor
if(x0+currCodingUnitSize < $PicWidthInSamples_L$ && y0+currCodingUnitSize < $PicHeightInSamples_L$ &&		
currCodingUnitSize > MinCodingUnitSize)		
split_coding_unit_flag	2	u(1) | ae(v)
if(split_coding_unit_flag) {		
splitCodingUnitSize = currCodingUnitSize >> 1		
x1 = x0 + splitCodingUnitSize		
y1 = y0 + splitCodingUnitSize		
coding_unit(x0, y0, splitCodingUnitSize)	2 | 3 | 4	
if(x1 < $PicWidthInSamples_L$)		
coding_unit(x1, y0, splitCodingUnitSize)	2 | 3 | 4	
if(y1 < $PicHeightInSamples_L$)		
coding_unit(x0, y1, splitCodingUnitSize)	2 | 3 | 4	
if(x1 < $PicWidthInSamples_L$ && y1 < $PicHeightInSamples_L$)		
coding_unit(x1, y1, splitCodingUnitSize)	2 | 3 | 4	
} else {		

prediction_unit(x0, y0, currCodingUnitSize)	2	
if(PredMode != MODE_SKIP ‖ !(PredMode == MODE_INTRA && **planar_flag** == 1))		
if(entropy_coding_mode_flag) {		
transform_unit_tree(x0, y0, currCodingUnitSize, 0)	3 | 4	
transform_unit_coeff(x0, y0, currCodingUnitSize, 0, 0)	3 | 4	
transform_unit_coeff(x0, y0, currCodingUnitSize, 0, 1)	3 | 4	
transform_unit_coeff(x0, y0, currCodingUnitSize, 0, 2)	3 | 4	
} else		
transform_unit_vlc(x0, y0, currCodingUnitSize)	3 | 4	
}		
}		

5.1.14 Prediction unit syntax

[Ed. preliminary draft]

prediction_unit(x0, y0, currPredUnitSize) {	C	Descriptor
if(slice_type != I)		
skip_flag		
if(skip_flag) {		
if(mv_competition_flag) {		
if(inter_pred_idc != Pred_L1 && NumMVPCand(L0) > 1)		
mvp_idx_l0	2	ue(v) | ae(v)
if(inter_pred_idc != Pred_L0 && NumMVPCand(L1) > 1)		
mvp_idx_l1	2	ue(v) | ae(v)
}		
if(pbic_flag) {		
if(NumICPCand > 1)		
icp_idx	2	ue(v) | ae(v)
}		
}		
else {		
if(!entropy_coding_mode_flag)		
mode_table_idx		vlc(n,v)
else {		
if(slice_type != I)		
pred_mode	2	u(1) | ae(v)
}		
if(PredMode == MODE_INTRA) {		
planar_flag	2	u(1) | ae(v)
if(planar_flag) {		
planar_delta_y = getPlanarDelta()		
planar_delta_uv_present_flag	2	u(1) | ae(v)
if(planar_delta_uv_present_flag) {		
planar_delta_u = getPlanarDelta()		
planar_delta_v = getPlanarDelta()		
}		
} else {		
if(entropy_coding_mode_flag)		
intra_split_flag	2	ae(v)
combined_intra_pred_flag	2	u(1) | ae(v)
for(i = 0; i < (intra_split_flag ? 4 : 1); i++) {		
prev_intra_luma_pred_ flag	2	u(1) | ae(v)
if(!prev_intra_pred_luma_flag)		
rem_intra_luma_pred_mode	2	ue(v) | ae(v)
if(adaptive_intra_smoothing_flag)		
intra_luma_filt_flag	2	u(1) | ae(v)
}		
}		
if(chroma_format_idc != 0)		

intra_chroma_pred_mode	2	ue(v) | ae(v)
}		
}		
else if(PredMode == MODE_INTER) {		
if(entropy_coding_mode_flag)		
inter_partitioning_idc	2	ue(v) | ae(v)
for(i= 0; i < NumPuParts(inter_partitioning_idc); i++) {		
if(NumMergeCandidates > 0) {		
merge_flag[i]	2	ue(1) | ae(v)
if(merge_flag[i] && NumMergeCandidates > 1)		
merge_left_flag[i]	2	ue(1) | ae(v)
}		
if(!merge_flag[i]) {		
if(slice_type == B)		
inter_pred_idc[i]	2	ue(v) | ae(v)
if(inter_pred_idc[i] != Pred_L1) {		
if(entropy_coding_mode_flag) {		
if(num_ref_idx_l0_active_minus1 > 0)		
ref_idx_l0[i **]**	2	ue(v) | ae(v)
if (!pbic_flag)		
mvres_l0 [i **]**	2	ue(1) | ae(v)
}		
else {		
ref_idx_mvres_l0[i **]**	2	ue(v) | ae(v)
}		
if (!pbic_flag) {		
mvd_l0[i **][** 0 **]**	2	se(v) | ae(v)
mvd_l0[i **][** 1 **]**	2	se(v) | ae(v)
if(mv_competition_flag && NumMVPCand(L0, i) > 1)		
mvp_idx_l0[i]	2	ue(v) | ae(v)
}		
}		
if(inter_pred_idc[i] != Pred_L0) {		
if(entropy_coding_mode_flag) {		
if(num_ref_idx_l1_active_minus1 > 0)		
ref_idx_l1[i **]**	2	ue(v) | ae(v)
if (!pbic_flag)		
mvres_l1 [i **]**	2	ue(1) | ae(v)
}		
else {		
ref_idx_mvres_l1[i **]**	2	ue(v) | ae(v)
}		
if (!pbic_flag) {		
mvd_l1[i **][** 0 **]**	2	se(v) | ae(v)
mvd_l1[i **][** 1 **]**	2	se(v) | ae(v)

_if(mv_competition_flag && _NumMVPCand(L1, i) > 1)		
_mvp_idx_l1[i]	2	ue(v) | ae(v)
}		
}		
if(pbic_flag) {		
if(inter_pred_idc[i] != Pred_L1)		
mvres_l0 [i]	2	ue(1) | ae(v)
if(inter_pred_idc[i] != Pred_L0)		
mvres_l1 [i]	2	ue(1) | ae(v)
if (inter_pred_idc[i] != Pred_L1 &&		
inter_pred_idc[i] != Pred_L0)		
ztree_type = 1		
else		
ztree_type = 0		
decZeroTree(ztree_type, ztree_data)	2	
if(ztree_type) {		
if (ztree_data[0] != 0)		
mvd_l0[i][0]	2	se(v) | ae(v)
if (ztree_data[1] != 0)		
mvd_l0[i][1]	2	se(v) | ae(v)
if (ztree_data[2] != 0)		
mvd_l1[i][0]	2	se(v) | ae(v)
if (ztree_data[3] != 0)		
mvd_l1[i][1]	2	se(v) | ae(v)
if (ztree_data[4] != 0)		
icd[i][0]	2	se(v) | ae(v)
if (ztree_data[5] != 0)		
icd[i][1]	2	se(v) | ae(v)
if (ztree_data[6] != 0)		
icd[i][2]	2	se(v) | ae(v)
} else { // if(ztree_type)		
if(inter_pred_idc[i] != Pred_L1) {		
if (ztree_data[0] != 0)		
mvd_l0[i][0]	2	se(v) | ae(v)
if (ztree_data[1] != 0)		
mvd_l0[i][1]	2	se(v) | ae(v)
}		
if(inter_pred_idc[i] != Pred_L0) {		
if (ztree_data[0] != 0)		
mvd_l1[i][0]	2	se(v) | ae(v)
if (ztree_data[1] != 0)		
mvd_l1[i][1]	2	se(v) | ae(v)
}		
if (ztree_data[2] != 0)		
icd[i][0]	2	se(v) | ae(v)
if (ztree_data[3] != 0)		

icd[i][2]	2	se(v) | ae(v)
} // if(ztree_type)		
if(mv_competition_flag) {		
if(inter_pred_idc[i] != Pred_L1 && NumMVPCand(L0, i) > 1)		
mvp_idx_l0[i]	2	ue(v) | ae(v)
if(inter_pred_idc[i] != Pred_L0 && NumMVPCand(L1, I) > 1)		
mvp_idx_l1[i]	2	ue(v) | ae(v)
}		
if(NumICPCand > 1)		
icp_idx[i]	2	ue(v) | ae(v)
}		
}		
}		
}		
else if(PredMode == MODE_DIRECT) {		
if(slice_type == B)		
inter_pred_idc	2	ue(v) | ae(v)
if(mv_competition_flag) {		
if(inter_pred_idc != Pred_L1 && NumMVPCand(L0) > 1)		
mvp_idx_l0	2	ue(v) | ae(v)
if(inter_pred_idc != Pred_L0 && NumMVPCand(L1) > 1)		
mvp_idx_l1	2	ue(v) | ae(v)
}		
if(pbic_flag) {		
if(NumICPCand > 1)		
icp_idx	2	ue(v) | ae(v)
}		
}		
}		

getPlanarDelta() {	C	Descriptor
planar_qdelta_indicator		vlc(5,v) | ae(v)
if (qdelta_indicator > 21)		
Qdelta = ((qdelta_indicator - 14) << 3) + 4		
else if(qdelta_indicator > 9)		
Qdelta = ((qdelta_indicator - 6) << 2) + 2		
else if (qdelta_indicator > 3)		
Qdelta = (qdelta_indicator - 2) << 1		
else		
Qdelta = qdelta_indicator		
if (qDelta > 0) {		
planar_sign		u(1) | ae(v)
PlanarDelta = sign ? – Qdelta: Qdelta		

}		
else		
PlanarDelta = Qdelta		
}		

decZeroTree(type, data) {	C	Descriptor
q = 0		
zero_blk	2	u(1) | ae(v)
val = 1 - zero_blk		
sp = 0		
for(i=0; i<2*ztree_size[type]-1; i++) {		
if(ztree_structure[type][i] = = 0) { // leaf node		
data[ztree_leaf_index[type][q]] = val		
val = stack[--sp]		
q++		
} else {		
if(val) {		
ztree_node_both	2	u(1) | ae(v)
if(ztree_node_both)		
stack[sp++] = true		
else {		
ztree_node_left	2	u(1) | ae(v)
if(ztree_node_left)		
stack[sp++] = false		
else {		
stack[sp++] = true		
val = false		
}		
}// if(ztree_node_both)		
} else		
stack[sp++] = false		
}// if(ztree_structure[type][i] = = 0)		
}// for(i=0; i<2*ztree_size[type]-1; i++)		
}		

5.1.15 Transform unit syntax

[Ed. preliminary draft]

[Note: signalling of chroma coded block flag not complete yet]

transform_unit_tree(x0, y0, currTransformUnitSize, layer) {	C	Descriptor
if(currTransformUnitSize > MinTransformUnitSize && currTransformUnitSize <= MaxTransformUnitSize)		
split_transform_unit_flag[x0][y0][layer]	3 | 4	u(1) | ae(v)
if(currTransformUnitSize <= MaxTransformUnitSize) {		
if(needChromaCbfU()) {		
coded_block_flag_chroma_u	3 | 4	u(1) | ae(v)
codedBlockFlag[x0][y0][layer][1] = coded_block_flag_chroma_u		
}		
if(needChromaCbfV()) {		
coded_block_flag_chroma_v	3 | 4	u(1) | ae(v)
codedBlockFlag[x0][y0][layer][2] = coded_block_flag_chroma_v		
}		
}		
if(split_transform_unit_flag[x0][y0][layer]) {		
splitTransformUnitSize = currTransformUnitSize >> 1		
x1 = x0 + splitTransformUnitSize		
y1 = y0 + splitTransformUnitSize		
transform_unit_tree(x0, y0, splitTransformUnitSize, layer+1)	3 | 4	
for(i=0; i<=2; i++)		
codedBlockFlag[x0][y0][layer][i] |= codedBlockFlag[x0][y0][layer+1][i]		
if(x1 < $PicWidthInSamples_L$) {		
transform_unit_tree(x1, y0, splitTransformUnitSize, layer+1)	3 | 4	
for(i=0; i<=2; i++)		
codedBlockFlag[x0][y0][layer][i] |= codedBlockFlag[x1][y0][layer+1][i]		
}		
if(y1 < $PicHeightInSamples_L$) {		
transform_unit_tree(x0, y1, splitTransformUnitSize, layer+1)	3 | 4	
for(i=0; i<=2; i++)		
codedBlockFlag[x0][y0][layer][i] |= codedBlockFlag[x0][y1][layer+1][i]		
}		
if(x1 < $PicWidthInSamples_L$ && y1 < $PicHeightInSamples_L$) {		
transform_unit_tree(x1, y1, splitTransformUnitSize, layer+1)	3 | 4	
for(i=0; i<=2; i++)		
codedBlockFlag[x0][y0][layer][i] |= codedBlockFlag[x1][y1][layer+1][i]		
}		
} else if(coded_block_flag_chroma_u || coded_block_flag_chroma_v) {		
coded_block_flag_luma	3 | 4	u(1) | ae(v)

codedBlockFlag[x0][y0][layer][0] = coded_block_flag_luma		
}		
}		

[Ed. preliminary draft]

transform_unit_coeff(x0, y0, currTransformUnitSize, layer, plane) {	C	Descriptor
if(codedBlockFlag[x0][y0][layer][plane]) {		
if(split_transform_unit_flag[x0][y0][layer]) {		
splitTransformUnitSize = currTransformUnitSize >> 1		
x1 = x0 + splitTransformUnitSize		
y1 = y0 + splitTransformUnitSize		
transform_unit_coeff(x0, y0, splitTransformUnitSize, layer+1, plane)	3 | 4	
if(x1 < $PicWidthInSamples_L$)		
transform_unit_coeff(x1, y0, splitTransformUnitSize, layer+1, plane)	3 | 4	
if(y1 < $PicHeightInSamples_L$)		
transform_unit_coeff(x0, y1, splitTransformUnitSize, layer+1, plane)	3 | 4	
if(x1 < $PicWidthInSamples_L$ && y1 < $PicHeightInSamples_L$)		
transform_unit_coeff(x1, y1, splitTransformUnitSize, layer+1, plane)	3 | 4	
} else {		
residual_block_pipe( )		
}		
}		
}		

[Ed. preliminary draft]

transform_unit_vlc(x0, y0, currTransformUnitSize) {	C	Descriptor
if(currTransformUnitSize > MinTransformUnitSize && currTransformUnitSize <= MaxTransformUnitSize)		
split_transform_unit_flag	3 | 4	u(1) | ae(v)
if(split_transform_unit_flag) {		
splitTransformUnitSize = currTransformUnitSize >> 1		
x1 = x0 + splitTransformUnitSize		
y1 = y0 + splitTransformUnitSize		
if(!entropy_coding_mode_flag)		
getVLCCBP( )		
transform_unit_vlc(x0, y0, splitTransformUnitSize)	3 | 4	
if(x1 < $PicWidthInSamples_L$)		
transform_unit_vlc(x1, y0, splitTransformUnitSize)	3 | 4	
if(y1 < $PicHeightInSamples_L$)		
transform_unit_vlc(x0, y1, splitTransformUnitSize)	3 | 4	
if(x1 < $PicWidthInSamples_L$ && y1 < $PicHeightInSamples_L$)		
transform_unit_vlc(x1, y1, splitTransformUnitSize)	3 | 4	
} else {		
if(entropy_coding_mode_flag)		

coded_block_flag	3 | 4	u(1) | ae(v)
if(coded_block_flag) {		
rotational_transform_idx	3 | 4	ue(v) | ae(v)
if(!entropy_coding_mode_flag)		
residual_block_vlc( )		
else		
residual_block_v2v( )		
}		
}		
}		

[Ed. preliminary draft]

residual_block_vlc (puIdx, tuSize, ctxID) {	C	Descriptor
last_pos_table_idx	3 | 4	vlc(n,v)
if(levelMagnitudeGreaterThanOneFlag)		
last_pos_level	3 | 4	vlc(0,v)
last_pos_sign	3 | 4	u(1)
runMode = 1		
while(runMode && i < numCodedCoeff) {		
isLevelOne_run	3 | 4	vlc(vlcIdx,v)
i += run		
if(!isLevelOne) {		
level_magnitude_minus2	3 | 4	vlc(0,v)
level_sign		
} else		
level_sign	3 | 4	u(1)
}		
while (i < numCodedCoeff) {		
level_magnitude	3 | 4	vlc(n,v)
if(level_magnitude)		
level_sign	3 | 4	u(1)
i++		
}		
}		

[Ed. preliminary draft]

getVLCCBP( ) {	C	Descriptor
if(modeIdx == 2) {		
cbp_id0	3 | 4	u(1)
}		
else if (modeIdx == 7) {		
cbp_id1	3 | 4	u(1)
if(cbp_id1 < 2)		
cbp_id2	3 | 4	u(1)
}		
else if (modeIdx == 4 || modeIdx == 8 || modeIdx == 10 || modeIdx == 12 || modeIdx == 14 || modeIdx == 15) {		
if(PredMode != MODE_INTRA && PuSize != PU_8x8 && currTransformUnitSize ! = TU_16x16)	3 | 4	u(1)
cbp_table_idx		
}		
}		

[Ed. preliminary draft: needs significant improvement]

residual_block_pipe(tuSize) {	C	Descriptor
log2tuSize = Log2(tuSize)		
maxScanPos = $tuSize^2$		
scanDirection = 1		
scanTopRight = 0		
scanBottomLeft = 0		
lastReceived = 0		
maxSubBlocks = $(tuSize >> 2)^2$		
for(scanPos = 0; scanPos < maxScanPos; scanPos++) {		
blockPos = scanPosLookUp[tuSize][scanDirection][scanPos]		
posY = blockPos >> log2tuSize		
posX = blockPos – (posY << log2tuSize)		
sig_flag	3 | 4	u(1)
if(sig_flag == 1) {		
if(posX > posY) {		
scanTopRight++;		
} else if(posY > posX) {		
scanBottomLeft++;		
}		
last_flag	3 | 4	u(1)
if(last_flag == 1) {		
lastReceived = 1		
break;		
}		
if(scanDirection = 1 && (posX == 0 || posY == tuSize - 1) || scanDirection = 0 && (posY == 0 || posX == tuSize - 1) {		
if(scanTopRight > scanBottomLeft) {		
scanDirection = 1		

} else {		
scanDirection = 0		
}		
}		
if(lastReceived) {		
levels[maxScanPos-1] = 1		
}		
for(subBlocks = 0; subBlocks < maxSubBlocks; subBlocks++) {		
numSigInSubBlock [Ed. todo]		
if(numSigInSubBlock > 0) {		
c1 = 0		
c2 = 0		
if(isFirstSubBlock) {		
ctxOffsest = 5		
} else {		
ctxOffset = numLevelsGreaterOne / 16		
numLevelsGreaterOne = 0		
}		
for(scanPos = 0; scanPos < 16; scanPos++) {		
blockPos = scanLookUpInverseRaster[scanPos] [need offset]		
c1 = min(c1, 4)		
ctxId = c1 + ctxOffset * 5		
abs_greater_one	3 | 4	vlc(n,v)
abs_level = abs_greater_one		
if(abs_level == 1) {		
c2 = min(c2, 4)		
ctxId = c2 + ctxOffset * 5		
coeff_abs_level	3 | 4	vlc(n,v)
abs_level = coeff_abs_level + 2		
c1 = 0		
c2++		
numLevelsGreaterOne++		
} else if(c1 > 0) {		
c1++;		
abs_level++		
} else {		
abs_level++		
}		
sign	3 | 4	u(1)
levels[scanPos] = sign || abs_level		
}		
}		
}		

5.2 Semantics

5.2.1 NAL unit semantics

NumBytesInNALunit specifies the size of the NAL unit in bytes. This value is required for decoding of the NAL unit. Some form of demarcation of NAL unit boundaries is necessary to enable inference of NumBytesInNALunit. One such demarcation method is specified in Annex B of ITU-T Rec. H.264 | ISO/IEC 14496-10 for the byte stream format.

forbidden_zero_bit shall be equal to 0.

nal_ref_idc not equal to 0 specifies that the content of the NAL unit contains a sequence parameter set, a picture parameter set, or a slice of a reference picture. nal_ref_idc equal to 0 for a NAL unit containing a slice indicates that the slice is part of a non-reference picture.

nal_ref_idc shall not be equal to 0 for sequence parameter set or picture parameter set NAL units. When nal_ref_idc is equal to 0 for one slice data NAL unit of a particular picture, it shall be equal to 0 for all slice data NAL units of the picture. nal_ref_idc shall not be equal to 0 for NAL units with nal_unit_type equal to 5.

nal_unit_type specifies the type of RBSP data structure contained in the NAL unit as specified in Table 5-1.

The column marked "C" in Table 5-1 lists the categories of the syntax elements that may be present in the NAL unit. In addition, syntax elements with syntax category "All" may be present, as determined by the syntax and semantics of the RBSP data structure. The presence or absence of any syntax elements of a particular listed category is determined from the syntax and semantics of the associated RBSP data structure. VCL and non-VCL NAL units are specified in Table 5-1 in the column labelled " NAL unit type class".

When the value of nal_unit_type is equal to 5 for a slice data NAL unit of a particular picture, all slice data NAL units of the picture shall have nal_unit_type equal to 5.

Table 5-1 – NAL unit type codes, syntax element categories, and NAL unit type classes

nal_unit_type	Content of NAL unit and RBSP syntax structure	C	NAL unit type class
0	Unspecified		non-VCL
1	Coded slice of a non-IDR picture slice_layer_rbsp( )	2, 3, 4	VCL
2..4	Reserved		non-VCL
5	Coded slice of an IDR picture slice_layer_rbsp( )	2, 3	VCL
6	Supplemental enhancement information (SEI) sei_rbsp( )	5	non-VCL
7	Sequence parameter set seq_parameter_set_rbsp( )	0	non-VCL
8	Picture parameter set pic_parameter_set_rbsp( )	1	non-VCL
9	Access unit delimiter access_unit_delimiter_rbsp( )	6	non-VCL
10..11	Reserved		non-VCL
12	Filler data filler_data_rbsp( )	9	non-VCL
13..23	Reserved		non-VCL
24..31	Unspecified		non-VCL

In the text, coded slice NAL unit collectively refers to a coded slice of a non-IDR picture NAL unit or to a coded slice of an IDR picture NAL unit. The variable IdrPicFlag is specified as

$$\text{IdrPicFlag} = ((\text{nal_unit_type} == 5) \ ? \ 1 \ : \ 0) \tag{16}$$

When the value of nal_unit_type is equal to 5 for a NAL unit containing a slice of a particular picture, the picture shall not contain NAL units with nal_unit_type equal to 1. Such a picture is referred to as an IDR picture.

temporal_id specifies a temporal identifier for the NAL unit. The assignment of values to temporal_id is constrained by the sub-bitstream extraction process as specified in subclause XX.

The value of temporal_id shall be the same for all NAL units of an access unit. When an access unit contains any NAL unit with nal_unit_type equal to 5, temporal_id shall be equal to 0.

output_flag affects the decoded picture output and removal processes as specified in Annex C.

reserved_zero_4bits shall be equal to 0. Decoders shall ignore the value of reserved_zero_4bits.

rbsp_byte[i **]** is the i-th byte of an RBSP. An RBSP is specified as an ordered sequence of bytes as follows.

The RBSP contains an SODB as follows.

– If the SODB is empty (i.e., zero bits in length), the RBSP is also empty.

– Otherwise, the RBSP contains the SODB as follows:

  1) The first byte of the RBSP contains the (most significant, left-most) eight bits of the SODB; the next byte of the RBSP shall contain the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain.

  2) rbsp_trailing_bits( ) are present after the SODB as follows:

     i) The first (most significant, left-most) bits of the final RBSP byte contains the remaining bits of the SODB (if any).

     ii) The next bit consists of a single rbsp_stop_one_bit equal to 1.

     iii) When the rbsp_stop_one_bit is not the last bit of a byte-aligned byte, one or more rbsp_alignment_zero_bit is present to result in byte alignment.

  3) One or more cabac_zero_word 16-bit syntax elements equal to 0x0000 may be present in some RBSPs after the rbsp_trailing_bits( ) at the end of the RBSP.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "_rbsp" suffix. These structures shall be carried within NAL units as the content of the rbsp_byte[i] data bytes. The association of the RBSP syntax structures to the NAL units shall be as specified in Table 5-1.

emulation_prevention_three_byte is a byte equal to 0x03. When an emulation_prevention_three_byte is present in the NAL unit, it shall be discarded by the decoding process.

The last byte of the NAL unit shall not be equal to 0x00.

Within the NAL unit, the following three-byte sequences shall not occur at any byte-aligned position:

– 0x000000

– 0x000001

– 0x000002

Within the NAL unit, any four-byte sequence that starts with 0x000003 other than the following sequences shall not occur at any byte-aligned position:

– 0x00000300

– 0x00000301

– 0x00000302

– 0x00000303

5.2.2 Sequence parameter set RBSP semantics

[Ed.: Parameters to be removed and/or added as appropriate.]

profile_idc and **level_idc** indicate the profile and level to which the coded video sequence conforms.

reserved_zero_8bits shall be equal to 0. Decoders shall ignore the value of reserved_zero_8bits.

seq_parameter_set_id identifies the sequence parameter set that is referred to by the picture parameter set. The value of seq_parameter_set_id shall be in the range of 0 to 31, inclusive.

bit_depth_luma_minus8 specifies the bit depth of the samples of the luma array and the value of the luma quantisation parameter range offset $QpBdOffset_Y$, as specified by

$$BitDepth_Y = 8 + bit_depth_luma_minus8 \quad (17)$$
$$QpBdOffset_Y = 6 * bit_depth_luma_minus8 \quad (18)$$

bit_depth_luma_minus8 shall be in the range of 0 to 6, inclusive.

bit_depth_chroma_minus8 specifies the bit depth of the samples of the chroma arrays and the value of the chroma quantisation parameter range offset $QpBdOffset_C$, as specified by

$$BitDepth_C = 8 + bit_depth_chroma_minus8 \quad (19)$$
$$QpBdOffset_C = 6 * bit_depth_chroma_minus8 \quad (20)$$

bit_depth_chroma_minus8 shall be in the range of 0 to 6, inclusive.

increased_bit_depth_luma specifies the increased bit depth of the samples of the luma array and the value of the luma quantisation parameter range offset $QpIBdOffset_Y$, as specified by

$$QpIBdOffset_Y = 6 * increased_bit_depth_luma \quad (21)$$

increased_bit_depth_chroma specifies the increased bit depth of the samples of the chroma array and the value of the chroma quantisation parameter range offset $QpIBdOffset_C$, as specified by

$$QpIBdOffset_C = 6 * increased_bit_depth_chroma \quad (22)$$

log2_max_frame_num_minus4 specifies the value of the variable MaxFrameNum that is used in frame_num related derivations as follows:

$$MaxFrameNum = 2^{(\, log2_max_frame_num_minus4 + 4\,)} \quad (23)$$

The value of log2_max_frame_num_minus4 shall be in the range of 0 to 12, inclusive.

pic_order_cnt_type specifies the method to decode picture order count (as specified in subclause XX). The value of pic_order_cnt_type shall be in the range of 0 to 2, inclusive.

pic_order_cnt_type shall not be equal to 2 in a coded video sequence that contains an access unit containing a non-reference frame followed immediately by an access unit containing a non-reference picture,

log2_max_pic_order_cnt_lsb_minus4 specifies the value of the variable MaxPicOrderCntLsb that is used in the decoding process for picture order count as follows:

$$MaxPicOrderCntLsb = 2^{(\, log2_max_pic_order_cnt_lsb_minus4 + 4\,)} \quad (24)$$

The value of log2_max_pic_order_cnt_lsb_minus4 shall be in the range of 0 to 12, inclusive.

delta_pic_order_always_zero_flag equal to 1 specifies that delta_pic_order_cnt[0] and delta_pic_order_cnt[1] are not present in the slice headers of the sequence and shall be inferred to be equal to 0.

delta_pic_order_always_zero_flag equal to 0 specifies that delta_pic_order_cnt[0] is present in the slice headers of the sequence and delta_pic_order_cnt[1] may be present in the slice headers of the sequence.

offset_for_non_ref_pic is used to calculate the picture order count of a non-reference picture as specified in subclause XX. The value of offset_for_non_ref_pic shall be in the range of $-2^{31}+1$ to $2^{31}-1$, inclusive.

num_ref_frames_in_pic_order_cnt_cycle is used in the decoding process for picture order count as specified in subclause XX. The value of num_ref_frames_in_pic_order_cnt_cycle shall be in the range of 0 to 255, inclusive.

offset_for_ref_frame[i] is an element of a list of num_ref_frames_in_pic_order_cnt_cycle values used in the decoding process for picture order count as specified in subclause XX. The value of offset_for_ref_frame[i] shall be in the range of $-2^{31}+1$ to $2^{31}-1$, inclusive.

When pic_order_cnt_type is equal to 1, the variable ExpectedDeltaPerPicOrderCntCycle is derived by

```
ExpectedDeltaPerPicOrderCntCycle = 0
for( i = 0;   i < num_ref_frames_in_pic_order_cnt_cycle;   i++ )
    ExpectedDeltaPerPicOrderCntCycle += offset_for_ref_frame[ i ]
```
(25)

max_num_ref_frames specifies the maximum number of short-term and long-term reference frames, complementary reference field pairs, and non-paired reference fields that may be used by the decoding process for inter prediction of any picture in the sequence. max_num_ref_frames also determines the size of the sliding window operation. The value of max_num_ref_frames shall be in the range of 0 to MaxDpbFrames, inclusive.

gaps_in_frame_num_value_allowed_flag specifies the allowed values of frame_num and the decoding process in case of an inferred gap between values of frame_num.

max_coding_unit_width specifies the maximum width of a coding unit

max_coding_unit_height specifies the maximum height of a coding unit

log2_min_transform_unit_size_minus2 specifies the minimum size of a transform unit.

The variable MinTransformUnitSize is derived as

```
MinTransformUnitSize = 1 << ( log2_min_transform_unit_size_minus2 + 2 )
```
(26)

max_transform_unit_hierarchy_depth specifies the maximum hierarchy depth for transform units. If not parsed, this element is set to 0.

The variable MaxTransformUnitSize is derived as

```
MaxTransformUnitSize = 1 << ( log2_min_transform_unit_size_minus2 + 2 +
                                max_transform_unit_hierarchy_depth )
```
(27)

The bitstream shall not contain data that result in a value of MaxTransformUnitSize that is greater than MaxCodingUnitSize.

max_coding_unit_hierarchy_depth_correct equal to 0 specifies that the coding unit hierarchy depth is not incremented. max_coding_unit_hierarchy_depth_correct equal to 1 specifies that the coding unit hierarchy depth is incremented by 1.

The variable MaxCodingUnitDepth is derived as

```
AddCUDepth = 0
While     ( max_coding_unit_width >> max_coding_unit_hierarchy_depth_correct ) >
          ( 1 << ( log2_min_transform_unit_size_minus2 ) + AddCUDepth ) ) ) {
      AddCuDepth
}
MaxCodingUnitHierarchyDepth = max_coding_unit_hierarchy_depth_correct + AddCUDepth
```

pic_width_in_luma_samples specifies the width of each decoded picture in luma samples.

[Ed.: macroblock may be replaced by smallest coding unit and mbs may be replaced by mcu, MinCodingUnitSize]

[Ed: Need to coordinate frame size signaling with cropping, etc.]

pic_height_in_luma_samples specifies the height of each decoded picture in luma samples.

numExtraFilters specifies the number of extra filters to be used along with the fixed set of filters described in Table 5-5, 5-6 and 5-8.

log2_filterCoeffPrecision specifies the number of bits to represent filter coefficient.

halfNumTap specifies half the number of filter tap.

filterCoef[i][j] specifies signed value of jth coefficient of ith extra filter and shall be represented by log2_filterCoeffPrecision bits.

adaptive_intra_smoothing_flag equal to 1 specifies that a flag for applying an additional smoothing filter on intra reference pixels.can be present for intra modes in prediction unti syntax. adaptive_intra_smoothing_flag equal to 0 specifies that no additional flag is present.

inter_merge_flag equal to 1 specifies that motion partition merging is can be applied.

interleaved_motion_vector_predictor_flag equal to 1 specifies that interleaved motion vector prediction can be applied.

pbic_flag equal to 1 specifies that partition based illumination compensation is applied.

5.2.3 Picture parameter set RBSP semantics

[Ed.: Parameters to be removed and/or added as appropriate.]

pic_parameter_set_id identifies the picture parameter set that is referred to in the slice header. The value of pic_parameter_set_id shall be in the range of 0 to 255, inclusive.

seq_parameter_set_id refers to the active sequence parameter set. The value of seq_parameter_set_id shall be in the range of 0 to 31, inclusive.

entropy_coding_mode_flag selects the entropy decoding method to be applied for the syntax elements for which two descriptors appear in the syntax tables as follows.

- If entropy_coding_mode_flag is equal to 0, the method specified by the left descriptor in the syntax table is applied.
- Otherwise (entropy_coding_mode_flag is equal to 1), the method specified by the right descriptor in the syntax table is applied.

num_ref_idx_l0_default_active_minus1 specifies how num_ref_idx_l0_active_minus1 is inferred for P and B slices with num_ref_idx_active_override_flag equal to 0. The value of num_ref_idx_l0_default_active_minus1 shall be in the range of 0 to 31, inclusive.

num_ref_idx_l1_default_active_minus1 specifies how num_ref_idx_l1_active_minus1 is inferred for B slices with num_ref_idx_active_override_flag equal to 0. The value of num_ref_idx_l1_default_active_minus1 shall be in the range of 0 to 31, inclusive.

pic_init_qp_minus26 specifies the initial value minus 26 of $SliceQP_Y$ for each slice. The initial value is modified at the slice layer when a non-zero value of slice_qp_delta is decoded, and is modified further when a non-zero value of mb_qp_delta is decoded at the macroblock layer. The value of pic_init_qp_minus26 shall be in the range of $-(26 + QpBdOffset_Y)$ to +25, inclusive.

constrained_intra_pred_flag equal to 0 specifies that intra prediction allows usage of residual data and decoded samples of neighbouring macroblocks coded using Inter macroblock prediction modes for the prediction of macroblocks coded using Intra macroblock prediction modes. constrained_intra_pred_flag equal to 1 specifies constrained intra prediction, in which case prediction of macroblocks coded using Intra macroblock prediction modes only uses residual data and decoded samples from I macroblock types.

numAllowedFilters[i] specifies the number of filters to be used during filtering process for $(i+1)^{th}$ subpel.

filtIdx[i][j] specifies the index of jth filter for $(i+1)^{th}$ subpel to be used during filtering process.

5.2.4 Supplemental enhancement information RBSP semantics

The Supplemental Enhancement Information (SEI) contains information that is not necessary to decode the samples of coded pictures from VCL NAL units.

5.2.5 Supplemental enhancement information message semantics

An SEI RBSP contains one or more SEI messages. Each SEI message consists of the variables specifying the type payloadType and size payloadSize of the SEI payload. SEI payloads are specified in Annex XX. The derived SEI payload size payloadSize is specified in bytes and shall be equal to the number of RBSP bytes in the SEI payload.

NOTE – The NAL unit byte sequence containing the SEI message might include one or more emulation prevention bytes (represented by emulation_prevention_three_byte syntax elements). Since the payload size of an SEI message is specified in RBSP bytes, the quantity of emulation prevention bytes is not included in the size payloadSize of an SEI payload.

ff_byte is a byte equal to 0xFF identifying a need for a longer representation of the syntax structure that it is used within.

last_payload_type_byte is the last byte of the payload type of an SEI message.

last_payload_size_byte is the last byte of the payload size of an SEI message.

5.2.6 Access unit delimiter RBSP semantics

The access unit delimiter may be used to indicate the type of slices present in a primary coded picture and to simplify the detection of the boundary between access units. There is no normative decoding process associated with the access unit delimiter.

primary_pic_type indicates that the slice_type values for all slices of the primary coded picture are members of the set listed in Table 5-2 for the given value of primary_pic_type.

Table 5-2 – Meaning of primary_pic_type

primary_pic_type	slice_type values that may be present in the primary coded picture
0	2, 7
1	0, 2, 5, 7
2	0, 1, 2, 5, 6, 7
3..7	Reserved

5.2.7 Filler data RBSP semantics

The filler data RBSP contains bytes whose value shall be equal to 0xFF. No normative decoding process is specified for a filler data RBSP.

ff_byte is a byte equal to 0xFF.

5.2.8 Slice layer RBSP semantics

The slice layer RBSP consists of a slice header and slice data.

5.2.9 RBSP slice trailing bits semantics

cabac_zero_word is a byte-aligned sequence of two bytes equal to 0x0000.

5.2.10 RBSP trailing bits semantics

rbsp_stop_one_bit shall be equal to 1.

rbsp_alignment_zero_bit shall be equal to 0.

5.2.11 Slice header semantics

[Ed.: Parameters should be removed and/or added as appropriate.]

When present, the value of the slice header syntax elements pic_parameter_set_id, frame_num, field_pic_flag, bottom_field_flag, idr_pic_id, pic_order_cnt_lsb, delta_pic_order_cnt_bottom, delta_pic_order_cnt[0], delta_pic_order_cnt[1], and slice_group_change_cycle shall be the same in all slice headers of a coded picture.

first_lctb_in_slice specifies the address of the first largest coding tree block in the slice.

The first largest coding tree block address of the slice is derived as follows.

– TBD.

slice_type specifies the coding type of the slice according to Table 5-3.

Table 5-3 – Name association to slice_type

slice_type	Name of slice_type
0	P (P slice)
1	B (B slice)
2	I (I slice)
5	P (P slice)
6	B (B slice)
7	I (I slice)

When slice_type has a value in the range 5..7, it is a requirement of bitstream conformance that all other slices of the current coded picture shall have a value of slice_type equal to the current value of slice_type or equal to the current value of slice_type minus 5.

When nal_unit_type is equal to 5 (IDR picture), slice_type shall be equal to 2 or 7.

When max_num_ref_frames is equal to 0, slice_type shall be equal to 2 or 7.

pic_parameter_set_id specifies the picture parameter set in use. The value of pic_parameter_set_id shall be in the range of 0 to 255, inclusive.

frame_num is used as an identifier for pictures and shall be represented by log2_max_frame_num_minus4 + 4 bits in the bitstream. frame_num is constrained as follows:
The variable PrevRefFrameNum is derived as follows.

– If the current picture is an IDR picture, PrevRefFrameNum is set equal to 0.
– Otherwise (the current picture is not an IDR picture), PrevRefFrameNum is set as follows.
  – If the decoding process for gaps in frame_num was invoked by the decoding process for an access unit that contained a non-reference picture that followed the previous access unit in decoding order that contained a reference picture, PrevRefFrameNum is set equal to the value of frame_num for the last of the "non-existing" reference frames inferred by the decoding process for gaps in frame_num.
  – Otherwise, PrevRefFrameNum is set equal to the value of frame_num for the previous access unit in decoding order that contained a reference picture.

The value of frame_num is constrained as follows.

– If the current picture is an IDR picture, frame_num shall be equal to 0.
– Otherwise (the current picture is not an IDR picture), referring to the primary coded picture in the previous access unit in decoding order that contains a reference picture as the preceding reference picture, the value of frame_num for the current picture shall not be equal to PrevRefFrameNum unless both of the following conditions are true.
  – the current picture and the preceding reference picture belong to consecutive access units in decoding order
  – one or more of the following conditions is true

- the preceding reference picture is an IDR picture
- the preceding reference picture includes a memory_management_control_operation syntax element equal to 5

  NOTE 1 – When the preceding reference picture includes a memory_management_control_operation syntax element equal to 5, PrevRefFrameNum is equal to 0.
- there is a primary coded picture that precedes the preceding reference picture and the primary coded picture that precedes the preceding reference picture does not have frame_num equal to PrevRefFrameNum
- there is a primary coded picture that precedes the preceding reference picture and the primary coded picture that precedes the preceding reference picture is not a reference picture

When the value of frame_num is not equal to PrevRefFrameNum, the following applies.

- There shall not be any previous frame in decoding order that is currently marked as "used for short-term reference" that has a value of frame_num equal to any value taken on by the variable UnusedShortTermFrameNum in the following:

```
UnusedShortTermFrameNum = ( PrevRefFrameNum + 1 ) % MaxFrameNum
while( UnusedShortTermFrameNum   !=   frame_num )
    UnusedShortTermFrameNum = ( UnusedShortTermFrameNum + 1 ) % MaxFrameNum
```

- The value of frame_num is constrained as follows.
  - If gaps_in_frame_num_value_allowed_flag is equal to 0, the value of frame_num for the current picture shall be equal to (PrevRefFrameNum + 1) % MaxFrameNum.
  - Otherwise (gaps_in_frame_num_value_allowed_flag is equal to 1), the following applies.
    - If frame_num is greater than PrevRefFrameNum, there shall not be any non-reference pictures in the bitstream that follow the previous reference picture and precede the current picture in decoding order in which either of the following conditions is true.
      - The value of frame_num for the non-reference picture is less than PrevRefFrameNum.
      - The value of frame_num for the non-reference picture is greater than the value of frame_num for the current picture.
    - Otherwise (frame_num is less than PrevRefFrameNum), there shall not be any non-reference pictures in the bitstream that follow the previous reference picture and precede the current picture in decoding order in which both of the following conditions are true.
      - The value of frame_num for the non-reference picture is less than PrevRefFrameNum.
      - The value of frame_num for the non-reference picture is greater than the value of frame_num for the current picture.

A picture including a memory_management_control_operation equal to 5 shall have frame_num constraints as described above and, after the decoding of the current picture and the processing of the memory management control operations, the picture shall be inferred to have had frame_num equal to 0 for all subsequent use in the decoding process.

idr_pic_id identifies an IDR picture. The values of idr_pic_id in all the slices of an IDR picture shall remain unchanged. When two consecutive access units in decoding order are both IDR access units, the value of idr_pic_id in the slices of the first such IDR access unit shall differ from the idr_pic_id in the second such IDR access unit. The value of idr_pic_id shall be in the range of 0 to 65535, inclusive.

pic_order_cnt_lsb specifies the picture order count modulo MaxPicOrderCntLsb for the top field of a coded frame or for a coded field. The length of the pic_order_cnt_lsb syntax element is log2_max_pic_order_cnt_lsb_minus4 + 4 bits. The value of the pic_order_cnt_lsb shall be in the range of 0 to MaxPicOrderCntLsb − 1, inclusive.

num_ref_idx_active_override_flag equal to 1 specifies that the syntax element num_ref_idx_l0_active_minus1 is present for P and B slices and that the syntax element num_ref_idx_l1_active_minus1 is present for B slices. num_ref_idx_active_override_flag equal to 0 specifies that the syntax elements num_ref_idx_l0_active_minus1 and num_ref_idx_l1_active_minus1 are not present.

When the current slice is a P or B slice and field_pic_flag is equal to 0 and the value of num_ref_idx_l0_default_active_minus1 in the picture parameter set exceeds 15, num_ref_idx_active_override_flag shall be equal to 1.

When the current slice is a B slice and field_pic_flag is equal to 0 and the value of num_ref_idx_l1_default_active_minus1 in the picture parameter set exceeds 15, num_ref_idx_active_override_flag shall be equal to 1.

num_ref_idx_l0_active_minus1 specifies the maximum reference index for reference picture list 0 that shall be used to decode the slice.

When the current slice is a P or B slice and num_ref_idx_l0_active_minus1 is not present, num_ref_idx_l0_active_minus1 shall be inferred to be equal to num_ref_idx_l0_default_active_minus1.

The range of num_ref_idx_l0_active_minus1 is specified as follows.p

– If field_pic_flag is equal to 0, num_ref_idx_l0_active_minus1 shall be in the range of 0 to 15, inclusive. When MbaffFrameFlag is equal to 1, num_ref_idx_l0_active_minus1 is the maximum index value for the decoding of frame macroblocks and 2 * num_ref_idx_l0_active_minus1 + 1 is the maximum index value for the decoding of field macroblocks.
– Otherwise (field_pic_flag is equal to 1), num_ref_idx_l0_active_minus1 shall be in the range of 0 to 31, inclusive.

num_ref_idx_l1_active_minus1 specifies the maximum reference index for reference picture list 1 that shall be used to decode the slice.

When the current slice is a B slice and num_ref_idx_l1_active_minus1 is not present, num_ref_idx_l1_active_minus1 shall be inferred to be equal to num_ref_idx_l1_default_active_minus1.

The range of num_ref_idx_l1_active_minus1 is constrained as specified in the semantics for num_ref_idx_l0_active_minus1 with l0 and list 0 replaced by l1 and list 1, respectively.

pipe_multi_codeword_flag equal to 1 specifies that multi-codeword PIPE is used as entropy coding method and the V2V codewords for the slice data syntax are coded in 12 separate partitions. pipe_multi_codeword_flag equal to 0 specifies that the V2V codeword are interleaved.

pipe_max_delay_in_bits_shift_6 multiplied by 32 specifies the maximum delay in bits for PIPE decoding if pipe_multi_codeword_flag is equal to 0.

Balanced_cpus equal to 1 specifies that mutli-codeword PIPE with V2V codewords is used as a parallel entropy coding method without load balancing. Balanced_cpus>1 specifies the number of load balanced parallel entropy partitions.

cabac_init_idc specifies the index for determining the initialisation table used in the initialisation process for context variables. The value of cabac_init_idc shall be in the range of 0 to 2, inclusive.

slice_qp_delta specifies the initial value of QP_Y to be used for all the macroblocks in the slice until modified by the value of mb_qp_delta in the macroblock layer. The initial QP_Y quantisation parameter for the slice is computed as

$$SliceQP_Y = 26 + pic_init_qp_minus26 + slice_qp_delta \quad (28)$$

The value of slice_qp_delta shall be limited such that $SliceQP_Y$ is in the range of $-QpBdOffset_Y$ to +51, inclusive.

mc_interpolation_idc specifies the method that is used for deriving Inter prediction samples.

mv_competition_flag specifies whether motion vector competition is used. When the motion vector competition is used, the function NumMVPCand(LX, puPartIdx) returns the total number of motion vector prediction candidates with the prediction unit partition puPartIdx and reference list LX as arguments.

mv_competition_temporal_flag specifies whether temporally collocated motion vector is used in the motion competition process.

collocated_from_l0_flag specifies that the reference picture list 0 is used in the derivation process for the temporal motion vector predictor. Otherwise, the reference picture list 1 is used. When collocated_from_l0_flag is not present, it shall be inferred to be equal to 1.

edge_based_prediction_flag: indicates whether the edge based prediction process shall be invoked during intra prediction. A value of 0 indicates that the edge based prediction process shall not be applied, a value of 1 indicates that the edge based prediction process shall be one of the candidate modes for intra prediction.

threshold_edge: specifies the threshold for the decision between the edge based prediction mode and the DC prediction mode during intra prediction. When edge_based_prediction_flag is equal to 0, threshold_edge shall not be present in the bitstream.

5.2.11.1 Reference picture list modification semantics

The syntax elements modification_of_pic_nums_idc, abs_diff_pic_num_minus1, and long_term_pic_num specify the change from the initial reference picture lists to the reference picture lists to be used for decoding the slice.

ref_pic_list_modification_flag_l0 equal to 1 specifies that the syntax element modification_of_pic_nums_idc is present for specifying reference picture list 0. ref_pic_list_modification_flag_l0 equal to 0 specifies that this syntax element is not present.

When ref_pic_list_modification_flag_l0 is equal to 1, the number of times that modification_of_pic_nums_idc is not equal to 3 following ref_pic_list_modification_flag_l0 shall not exceed num_ref_idx_l0_active_minus1 + 1.

When RefPicList0[num_ref_idx_l0_active_minus1] in the initial reference picture list produced as specified in subclause XX is equal to "no reference picture", ref_pic_list_modification_flag_l0 shall be equal to 1 and modification_of_pic_nums_idc shall not be equal to 3 until RefPicList0[num_ref_idx_l0_active_minus1] in the modified list produced as specified in subclause XX is not equal to "no reference picture".

ref_pic_list_modification_flag_l1 equal to 1 specifies that the syntax element modification_of_pic_nums_idc is present for specifying reference picture list 1. ref_pic_list_modification_flag_l1 equal to 0 specifies that this syntax element is not present.

When ref_pic_list_modification_flag_l1 is equal to 1, the number of times that modification_of_pic_nums_idc is not equal to 3 following ref_pic_list_modification_flag_l1 shall not exceed num_ref_idx_l1_active_minus1 + 1.

When decoding a slice with slice_type equal to 1 or 6 and RefPicList1[num_ref_idx_l1_active_minus1] in the initial reference picture list produced as specified in subclause XX is equal to "no reference picture", ref_pic_list_modification_flag_l1 shall be equal to 1 and modification_of_pic_nums_idc shall not be equal to 3 until RefPicList1[num_ref_idx_l1_active_minus1] in the modified list produced as specified in subclause XX is not equal to "no reference picture".

modification_of_pic_nums_idc together with abs_diff_pic_num_minus1 or long_term_pic_num specifies which of the reference pictures are re-mapped. The values of modification_of_pic_nums_idc are specified in Table Table 5-4. The value of the first modification_of_pic_nums_idc that follows immediately after ref_pic_list_modification_flag_l0 or ref_pic_list_modification_flag_l1 shall not be equal to 3.

Table 5-4 – modification_of_pic_nums_idc operations for modification of reference picture lists

modification_of_pic_nums_idc	modification specified
0	abs_diff_pic_num_minus1 is present and corresponds to a difference to subtract from a picture number prediction value
1	abs_diff_pic_num_minus1 is present and corresponds to a difference to add to a picture number prediction value
2	long_term_pic_num is present and specifies the long-term picture number for a reference picture
3	End loop for modification of the initial reference picture list

abs_diff_pic_num_minus1 plus 1 specifies the absolute difference between the picture number of the picture being moved to the current index in the list and the picture number prediction value. abs_diff_pic_num_minus1 shall be in the range of 0 to MaxPicNum − 1. The allowed values of abs_diff_pic_num_minus1 are further restricted as specified in subclause XX.

long_term_pic_num specifies the long-term picture number of the picture being moved to the current index in the list. When decoding a coded frame, long_term_pic_num shall be equal to a LongTermPicNum assigned to one of the reference frames marked as "used for long-term reference".

5.2.11.2 Decoded reference picture marking semantics

The syntax elements no_output_of_prior_pics_flag, long_term_reference_flag, adaptive_ref_pic_marking_mode_flag, memory_management_control_operation, difference_of_pic_nums_minus1, long_term_frame_idx, long_term_pic_num, and max_long_term_frame_idx_plus1 specify marking of the reference pictures.

The marking of a reference picture can be "unused for reference", "used for short-term reference", or "used for long-term reference", but only one among these three. When a reference picture is referred to as being marked as "used for reference", this collectively refers to the picture being marked as "used for short-term reference" or "used for long-term reference" (but not both). A reference picture that is marked as "used for short-term reference" is referred to as a short-term reference picture. A reference picture that is marked as "used for long-term reference" is referred to as a long-term reference picture.

The content of the decoded reference picture marking syntax structure shall be the same in all slice headers of the primary coded picture. When one or more redundant coded pictures are present, the content of the decoded reference picture marking syntax structure shall be the same in all slice headers of a redundant coded picture with a particular value of redundant_pic_cnt.

The syntax category of the decoded reference picture marking syntax structure shall be inferred as follows.

– If the decoded reference picture marking syntax structure is in a slice header, the syntax category of the decoded reference picture marking syntax structure is inferred to be equal to 2.

– Otherwise (the decoded reference picture marking syntax structure is in a decoded reference picture marking repetition SEI message), the syntax category of the decoded reference picture marking syntax structure is inferred to be equal to 5.

no_output_of_prior_pics_flag specifies how the previously-decoded pictures in the decoded picture buffer are treated after decoding of an IDR picture. When the IDR picture is the first IDR picture in the bitstream, the value of no_output_of_prior_pics_flag has no effect on the decoding process. When the IDR picture is not the first IDR picture in the bitstream and the value of PicWidthInMbs, FrameHeightInMbs, or max_dec_frame_buffering derived from the active sequence parameter set is different from the value of PicWidthInMbs, FrameHeightInMbs, or max_dec_frame_buffering derived from the sequence parameter set active for the preceding picture, no_output_of_prior_pics_flag equal to 1 may (but should not) be inferred by the decoder, regardless of the actual value of no_output_of_prior_pics_flag.

long_term_reference_flag equal to 0 specifies that the MaxLongTermFrameIdx variable is set equal to "no long-term frame indices" and that the IDR picture is marked as "used for short-term reference". long_term_reference_flag equal to 1 specifies that the MaxLongTermFrameIdx variable is set equal to 0 and that the current IDR picture is marked "used for long-term reference" and is assigned LongTermFrameIdx equal to 0. When max_num_ref_frames is equal to 0, long_term_reference_flag shall be equal to 0.

adaptive_ref_pic_marking_mode_flag selects the reference picture marking mode of the currently decoded picture as specified in Table 5-5. adaptive_ref_pic_marking_mode_flag shall be equal to 1 when the number of frames, complementary field pairs, and non-paired fields that are currently marked as "used for long-term reference" is equal to Max(max_num_ref_frames, 1).

Table 5-5 – Interpretation of adaptive_ref_pic_marking_mode_flag

adaptive_ref_pic_marking_mode_flag	Reference picture marking mode specified
0	Sliding window reference picture marking mode: A marking mode providing a first-in first-out mechanism for short-term reference pictures.
1	Adaptive reference picture marking mode: A reference picture marking mode providing syntax elements to specify marking of reference pictures as "unused for reference" and to assign long-term frame indices.

memory_management_control_operation specifies a control operation to be applied to affect the reference picture marking. The memory_management_control_operation syntax element is followed by data necessary for the operation specified by the value of memory_management_control_operation. The values and control operations associated with memory_management_control_operation are specified in Table 5-6. The

memory_management_control_operation syntax elements are processed by the decoding process in the order in which they appear in the slice header, and the semantics constraints expressed for each memory_management_control_operation apply at the specific position in that order at which that individual memory_management_control_operation is processed.

For interpretation of memory_management_control_operation, the term reference picture is interpreted as follows.

– If the current picture is a frame, the term reference picture refers either to a reference frame or a complementary reference field pair.

– Otherwise (the current picture is a field), the term reference picture refers either to a reference field or a field of a reference frame.

memory_management_control_operation shall not be equal to 1 in a slice header unless the specified reference picture is marked as "used for short-term reference" when the memory_management_control_operation is processed by the decoding process.

memory_management_control_operation shall not be equal to 2 in a slice header unless the specified long-term picture number refers to a reference picture that is marked as "used for long-term reference" when the memory_management_control_operation is processed by the decoding process.

memory_management_control_operation shall not be equal to 3 in a slice header unless the specified reference picture is marked as "used for short-term reference" when the memory_management_control_operation is processed by the decoding process.

memory_management_control_operation shall not be equal to 3 or 6 if the value of the variable MaxLongTermFrameIdx is equal to "no long-term frame indices" when the memory_management_control_operation is processed by the decoding process.

Not more than one memory_management_control_operation equal to 4 shall be present in a slice header.

Not more than one memory_management_control_operation equal to 5 shall be present in a slice header.

Not more than one memory_management_control_operation equal to 6 shall be present in a slice header.

memory_management_control_operation shall not be equal to 5 in a slice header unless no memory_management_control_operation in the range of 1 to 3 is present in the same decoded reference picture marking syntax structure.

A memory_management_control_operation equal to 5 shall not follow a memory_management_control_operation equal to 6 in the same slice header.

When a memory_management_control_operation equal to 6 is present, any memory_management_control_operation equal to 2, 3, or 4 that follows the memory_management_control_operation equal to 6 within the same slice header shall not specify the current picture to be marked as "unused for reference".

Table 5-6 – Memory management control operation (memory_management_control_operation) values

memory_management_control_operation	Memory Management Control Operation
0	End memory_management_control_operation syntax element loop
1	Mark a short-term reference picture as "unused for reference"
2	Mark a long-term reference picture as "unused for reference"
3	Mark a short-term reference picture as "used for long-term reference" and assign a long-term frame index to it
4	Specify the maximum long-term frame index and mark all long-term reference pictures having long-term frame indices greater than the maximum value as "unused for reference"
5	Mark all reference pictures as "unused for reference" and set the MaxLongTermFrameIdx variable to "no long-term frame indices"
6	Mark the current picture as "used for long-term reference" and assign a long-term frame index to it

When decoding a field and a memory_management_control_operation command equal to 3 is present that assigns a long-term frame index to a field that is part of a short-term reference frame or part of a complementary reference field pair, another memory_management_control_operation command (equal to 3 or 6) to assign the same long-term frame index to the other field of the same frame or complementary reference field pair shall be present in the same decoded reference picture marking syntax structure.

When the first field (in decoding order) of a complementary reference field pair included a long_term_reference_flag equal to 1 or a memory_management_control_operation command equal to 6, the decoded reference picture marking syntax structure for the second field of the complementary reference field pair shall contain a memory_management_control_operation command equal to 6 that assigns the same long-term frame index to the second field.

When the second field (in decoding order) of a complementary reference field pair includes a memory_management_control_operation command equal to 6 that assigns a long-term frame index to this field and the first field of the complementary reference field pair is marked as "used for short-term reference" when the memory_management_control_operation command equal to 6 is processed by the decoding process, the decoded reference picture marking syntax structure of that second field shall contain either a memory_management_control_operation command equal to 1 that marks the first field of the complementary field pair as "unused for reference" or a memory_management_control_operation command equal to 3 that marks the first field of the complementary field pair as "used for long-term reference" and assigns the same long-term frame index to the first field.

difference_of_pic_nums_minus1 is used (with memory_management_control_operation equal to 3 or 1) to assign a long-term frame index to a short-term reference picture or to mark a short-term reference picture as "unused for reference". When the associated memory_management_control_operation is processed by the decoding process, the resulting picture number derived from difference_of_pic_nums_minus1 shall be a picture number assigned to one of the reference pictures marked as "used for reference" and not previously assigned to a long-term frame index.

The resulting picture number is constrained as follows.

– If field_pic_flag is equal to 0, the resulting picture number shall be one of the set of picture numbers assigned to reference frames or complementary reference field pairs.

– Otherwise (field_pic_flag is equal to 1), the resulting picture number shall be one of the set of picture numbers assigned to reference fields.

long_term_pic_num is used (with memory_management_control_operation equal to 2) to mark a long-term reference picture as "unused for reference". When the associated memory_management_control_operation is

processed by the decoding process, long_term_pic_num shall be equal to a long-term picture number assigned to one of the reference pictures that is currently marked as "used for long-term reference".

The resulting long-term picture number is constrained as follows.

– If field_pic_flag is equal to 0, the resulting long-term picture number shall be one of the set of long-term picture numbers assigned to reference frames or complementary reference field pairs.

– Otherwise (field_pic_flag is equal to 1), the resulting long-term picture number shall be one of the set of long-term picture numbers assigned to reference fields.

long_term_frame_idx is used (with memory_management_control_operation equal to 3 or 6) to assign a long-term frame index to a picture. When the associated memory_management_control_operation is processed by the decoding process, the value of long_term_frame_idx shall be in the range of 0 to MaxLongTermFrameIdx, inclusive.

max_long_term_frame_idx_plus1 minus 1 specifies the maximum value of long-term frame index allowed for long-term reference pictures (until receipt of another value of max_long_term_frame_idx_plus1). The value of max_long_term_frame_idx_plus1 shall be in the range of 0 to max_num_ref_frames, inclusive.

5.2.11.3 Adaptive loop filter parameter semantics

adaptive_loop_filter_flag specifies whether adaptive loop filter applies or not for the current slice

alf_length_luma_~~hor~~_minus5_div2 specifies the filter length ~~in horizontal direction~~ for the luma component used in the adaptive loop filter process. ~~This filter shall be applied to the reconstructed signal.~~ The filter length used to define the number of coded ~~luma~~ filter cofficients is specified as follows:

AlfLengthLuma~~_hor = (~~ = (alf_length_luma_~~hor~~_minus5_div2~~<<~~ << 1~~)+~~) + 5

~~**alf_length_luma_ver_minus5_div2** specifies the filter length in vertical direction for the luma component used in the adaptive loop filter process. This filter shall be applied to the reconstructed signal. The filter length used to define the number of coded luma filter cofficients is specified as follows:~~

The number of coded luma filter coefficients is specified as follows:

AlfNumCoeffLuma = ((AlfLengthLuma~~_ver = (alf_length_luma_ver_minus5_div2<<1)+5~~

~~**alf_length_luma_pred_hor_minus1_div2** specifies the filter length in horizontal direction for the luma component used in the adaptive loop filter process. This filter shall be applied to the prediction signal. The filter length used to define the number of coded luma filter cofficients is specified as follows:~~

~~*~~ * AlfLengthLuma~~_pred_hor = (alf_length_luma_pred_hor_minus1_div2<<1)+1~~ + 2) >> 2) + 2

~~**alf_length_luma_pred_ver_minus1_div2** specifies the filter length in vertical direction for the luma component used in the adaptive loop filter process. This filter shall be applied to the prediction signal. The filter length used to define the number of coded luma filter cofficients is specified as follows:~~

~~AlfLengthLuma_pred_ver = (alf_length_luma_pred_ver_minus1_div2 <<1)+1~~

~~**alf_length_luma_qpe_hor_minus1_div2** specifies the filter length in horizontal direction for the luma component used in the adaptive loop filter process. This filter shall be applied to the quantized prediction error signal. The filter length used to define the number of coded luma filter cofficients is specified as follows:~~

~~AlfLengthLuma_qpe_hor = (alf_length_luma_qpe_hor_minus1_div2<<1)+1~~

~~**alf_length_luma_qpe_ver_minus1_div2** specifies the filter length in vertical direction for the luma component used in the adaptive loop filter process. This filter shall be applied to the quantized prediction error signal. The filter length used to define the number of coded luma filter cofficients is specified as follows:~~

~~AlfLengthLuma_qpe_ver = (alf_length_luma_qpe_ver_minus1_div2<<1)+1~~

~~**alf_quant_step_size_luma_hor** specifies the quantization step size of the filter coefficients that are used for the horizontal filtering of the luma component. Table 5-7 specifies the step size Alf_filter_precision_Luma_hor.~~

~~**alf_quant_step_size_luma_ver** specifies the quantization step size of the filter coefficients that are used for the vertical filtering of the luma component.Table 5-8 specifies the step size Alf_filter_precision_ Luma_ver.~~

~~**Table 5-7 Specification of Alf_filter_precision_Luma_hor**~~

~~alf_quant_step_size_luma_hor~~	~~Alf_filter_precision_Luma_hor~~
~~0~~	~~3~~
~~4~~	~~4~~
~~2~~	~~5~~
~~3~~	~~6~~
~~4~~	~~7~~
~~5~~	~~8~~
~~6~~	~~9~~
~~7~~	~~10~~
~~8~~	~~11~~
~~9~~	~~12~~
~~10~~	~~13~~
~~11~~	~~14~~

~~**Table 5-8 Specification of Alf_filter_precision_Luma_ver**~~

~~alf_quant_step_size_luma_ver~~	~~Alf_filter_precision_Luma_ver~~
~~0~~	~~3~~
~~4~~	~~4~~
~~2~~	~~5~~
~~3~~	~~6~~
~~4~~	~~7~~
~~5~~	~~8~~
~~6~~	~~9~~
~~7~~	~~10~~
~~8~~	~~11~~
~~9~~	~~12~~
~~10~~	~~13~~
~~11~~	~~14~~

alf_noFilters_minus1 specifies the number of filter sets for the current slice

AlfNoFilters shall be computed as

AlfNoFilters = alf_noFilters_minus1+1

If AlfNoFilters is less than 3, the number of filter sets shall be defined as

AlfNumFilters = AlfNoFilters

Otherwise, AlfNumFilters shall be defined as

AlfNumFilters = ∑ (alf_filter_pattern[i]) with i=0... AlfMaxNumFilters-1

Note that AlfMaxNumFilters is the number of maximum filter sets used, i.e., 16.

alf_startSecondFilter specifies the index where the second filter is applied, when AlfNoFilters is less than 3.

Index table is alfVarIndTab[i] is defined based on **alf_startSecondFilter.** alfVarIndTab[i] is set equal to 0 for i=0... alf_startSecondFilter-1, and set equal to 1 for i=alf_startSecondFilter...AlfMaxNumFilters-1. The first filter is applied to pixels which have activity measure index is equal to from 0 to alf_startSecondFilter-1. Otherwise the second filter is applied.

alf_filter_pattern[i] specifies the indices where each filter is applied, when AlfNoFilters is more than 2.

when AlfNoFilters is greater than 2, alfVarIndTab is defined as follows:.

alfVarIndTab[0]=0

for (i=1; i< AlfMaxNumFilters; i++) {

if (alf_filter_pattern[i]!=0) alfVarIndTab[i] = alfVarIndTab[i-1] + 1

otherwise alfVarIndTab[i] = alfVarIndTab[i-1]

}

alf_predMethod specifies whether filter prediction method is used for filter coefficients coding.

alf_coeff_luma~~_**hor**~~[[i][j]~~[i]~~] specifies the ~~i~~j-th ~~filter~~ coefficient of ~~j~~i-th ~~horizontal~~ filter for the ~~reconstructed signal of the luma component.~~

~~**alf_coeff_luma_ver**[j][i] specifies the i-th filter coefficient of j-th vertical filter for the reconstructed signal of the luma component.~~

~~**alf_coeff_luma_pred_hor**[j][i] specifies the i-th filter coefficient of j-th horizontal filter for the prediction signal of the luma component.~~

~~**alf_coeff_luma_ pred_ver**[j][i] specifies the i-th filter coefficient of j-th vertical filter for the prediction signal of the luma component.~~

~~**alf_coeff_luma_qpe_hor**[j][i] specifies the i-th filter coefficient of j-th horizontal filter for the quantized prediction error signal of the luma component.~~

~~**alf_coeff_luma_ qpe_ver**[j][i] specifies the i-th filter coefficient of j-th vertical filter for the quantized prediction error signal of the~~ luma component.

alf_minKStart_minus1 specifies the minimum order k of k-th order exponential golomb code for the ALF luma coefficients coding.

alf_golombIndexBit specifies the difference in order k of k-th order exponential golomb code for the different groups of ALF luma coefficients. Note that there are several groups of ALF luma coefficients where each group may have different order k. Above group mapping is defined by DepthTable[i] with i=0...AlfNumCoeffLuma-1. Different AlfLengthLuma leads to different DepthTable[i]. From alf_minKStart_minus1 and alf_golombIndexBit, the order k of each group, kMinTab[i] with i=0...AlfmaxDepth-1, shall be decided as follows. kMin=alf_minKStart_minus1+1

for (i=0; i< AlfmaxDepth; i++) {

if (alf_golombIndexBit) kMinTab[i]=kMin+1

otherwise kMinTab[i]=kMin

kMin = kMinTab[i]

}

where AlfmaxDepth is the maximum number of groups for ALF luma coefficients coding.

Table 5-9 Specification of DepthTable

Coefficient index	AlfLengthLuma=5	AlfLengthLuma=7	AlfLengthLuma=9
0	3	4	5
1	3	4	5
2	4	5	6
3	3	4	5
4	3	4	5
5	4	5	6
6	5	6	7
7	5	5	6
8		4	5
9		4	5
10		5	6
11		6	7
12		7	8
13		7	7
14			6
15			5
16			5
17			6
18			7
19			8
20			9
21			9

alf_chroma_idc specifies which chroma components are to be filtered.

Table 5-10 Specification of alf_chroma_idc

alf_chroma_idc	chroma component to be filtered
0	None
1	Cr
2	Cb
3	Cb and Cr

~~**alf_reuse_coefficients_for_chroma_flag** specifies if the filters for the luma component are also applied to the chroma components.~~

~~If alf_reuse_coefficients_for_chroma_flag is equal to 1, the same filters are used~~

~~If alf_reuse_coefficients_for_chroma_flag is equal to 0, individual filters are used~~

alf_length_chroma_~~hor_minus_5~~minus5_div2 specifies the filter length ~~in horizontal direction for the~~for the chroma components used in the adaptive loop filter process. This filter shall be applied to the reconstructed signal. The filter length used to define the number of coded ~~chroma~~ filter cofficients is specified as follows:

AlfLengthChroma~~_hor = (~~ = (alf_length_chroma_~~hor_minus_5~~minus5_div2 << 1~~)+~~) + 5

~~**alf_length_chroma_ver_minus_5_div2** specifies the filter length in vertical direction for the chroma components used in the adaptive loop filter process. This filter shall be applied to the reconstructed signal. The filter length used to define the number of coded chroma filter cofficients is specified as follows:~~

The number of coded chroma filter coefficients is specified as follows:

AlfNumCoeffChroma = ((AlfLengthChroma~~_ver = (alf_length_chroma_ver_minus_5_div2 <<1)+5~~

alf_length_chroma_pred_hor_minus_1_div2 specifies the filter length in horizontal direction for the chroma components used in the adaptive loop filter process. This filter shall be applied to the prediction signal. The filter length used to define the number of coded chroma filter cofficients is specified as follows:

* AlfLengthChroma_pred_hor = (alf_length_chroma_pred_hor_minus_1_div2 <<1)+ + 1) >> 1) + 1

alf_length_chroma_pred_ver_minus_1_div2 specifies the filter length in vertical direction for the chroma components used in the adaptive loop filter process. This filter shall be applied to the prediction signal. The filter length used to define the number of coded chroma filter cofficients is specified as follows:

AlfLengthChroma_pred_ver = (alf_length_chroma_pred_ver_minus_1_div2 <<1)+1

alf_length_chroma_qpe_hor_minus_1_div2 specifies the filter length in horizontal direction for the chroma components used in the adaptive loop filter process. This filter shall be applied to the quantized prediction error signal. The filter length used to define the number of coded chroma filter cofficients is specified as follows:

AlfLengthChroma_qpe_hor = (alf_length_chroma_qpe_hor_minus_1_div2 <<1)+1

alf_length_chroma_qpe_ver_minus_1_div2 specifies the filter length in vertical direction for the chroma components used in the adaptive loop filter process. This filter shall be applied to the quantized prediction error signal. The filter length used to define the number of coded chroma filter cofficients is specified as follows:

AlfLengthChroma_qpe_ver = (alf_length_chroma_qpe_ver_minus_1_div2 <<1)+1

alf_quant_step_size_chroma_hor specifies the quantization step size of the filter coefficients that are used for the horizontal filtering of the chroma components. Table 5-11 specifies the step size Alf_filter_precision_Chroma_hor.

alf_quant_step_size_chroma_ver specifies the quantization step size of the filter coefficients that are used for the vertical filtering of the chroma components. Table 5-12 specifies the step size Alf_filter_precision_Chroma_ver.

Table 5-11 Specification of Alf_filter_precision_Chroma_hor

alf_quant_step_size_chroma_hor	Alf_filter_precision_Chroma_hor
0	3
4	4
2	5
3	6
4	7
5	8
6	9
7	10
8	11
9	12
10	13
11	14

Table 5-12 Specification of Alf_filter_precision_Chroma_ver

~~alf_quant_step_size_chroma_ver~~	~~Alf_filter_precision_Chroma_ver~~
~~0~~	~~3~~
~~4~~	~~4~~
~~2~~	~~5~~
~~3~~	~~6~~
~~4~~	~~7~~
~~5~~	~~8~~
~~6~~	~~9~~
~~7~~	~~10~~
~~8~~	~~11~~
~~9~~	~~12~~
~~10~~	~~13~~
~~11~~	~~14~~

alf_coeff_chroma~~_hor~~[i] specifies the ith filter coefficient ~~of the horizontal filter for the reconstructed signal of the chroma components.~~

~~**alf_coeff_chroma_ver**[i] specifies the ith filter coefficient of the vertical filter for the reconstructed signal of the chroma components.~~

~~**alf_coeff_chroma_pred_hor**[i] specifies the ith filter coefficient of the horizontal filter for the prediction signal of the chroma components.~~

~~**alf_coeff_chroma_ pred_ver**[i] specifies the ith filter coefficient of the vertical filter for the prediction signal of the chroma components.~~

~~**alf_coeff_chroma_qpe_hor**[i] specifies the ith filter coefficient of the horizontal filter for the quantized prediction error signal of the chroma components.~~

~~**alf_coeff_chroma_ qpe_ver**[i] specifies the ith filter coefficient of the vertical filter for the quantized prediction error signal of the~~for the chroma components.

alf_cu_control_flag specifies whether the adaptive loop filter process for luma component shall be applied adaptively. If alf_cu_control_flag is equal to 1, the filtering process shall be applied only when alf_flag is equal to 1, otherwise, the filtering process shall be applied to all luma samples in the current slice.

alf_cu_control_max_depth specifies the maximum split depth from the largest coding unit for deciding the application of the adaptive loop filter process. The minimum coding unit size used for the decision shall be computed as

$$\text{AlfCtrlMinCodingUnitSize} = \text{MaxCodingUnitSize} >> \text{alf_cu_control_max_depth}$$

alf_length_cu_control_info specifies the information of the number of alf_cu_flag succeeding this information. alf_length_cu_control_info shall present unless alf_cu_control_max_depth is 0. alf_length_cu_control_info is a fixed length code, and the number of bits for alf_length_cu_control_info shall be computed as

$$\text{Ceil}\ (\ \text{Log2}\ (\text{NumLCUsInPicture} * (4^{\text{alf_cu_control_max_depth}} - 1)\ )\ )$$

where NumLCUsInPicture is the number of LCUs in a picture. When alf_length_cu_control_info is not present, it shall be inferred to be 0. NumAlfCuFlag, the number alf_cu_flag, shall be computed as

$$\text{NumAlfCuFlag} = \text{alf_length_cu_control_info} + \text{NumLCUsInPicture}$$

alf_cu_flag[i] specifies whether a coding unit is filtered when the adaptive loop filter is applied. During the CU decoding, alf_cu_flag[i] is used for the adaptive loop filter control flag of the current CU, if one of the following conditions is true:

- currCodingUnitSize is equal to AlfMinCtrlCodingUnitSize and split_coding_unit_flag is 1,
- currCodingUnitSize is greater than AlfCtrlMinCodingUnitSize and split_coding_unit_figag is 0,

where i starts with 0 and i is incremented when alf_cu_flag[i] is used for the current CU.

[3-Input-ALF is still missing]

5.2.11.4 Switched Filter with Offsets parameter syntax

predict_filter_flag specifies whether to use prediction of the filter or not.

If mc_interpolation_idc is equal to 3, the variable numFilters is set to 4, otherwise if mc_interpolation_idc is equal to 4, the variable numFilters is set to 2.

Variable log2_numFilters is derived from numFilters as follows:

log2_numFilters = $\log_2$(numFilters)

sifo_type is used as an identifier for deriving the value of sifo_filters[i].

sifo_filter[i] is used as an identifier for filter index to be used during filtering process for $(i+1)^{th}$ subpel and shall be represented by log2_numFilters bits, where i = $xFrac_L$+($yFracy_L$<<2)

bitsForFilter is derived as follows:

slice_type	predict_filter_flag	sifo_type	i	BitsforFilter
P	0	0	1,2,3,4,8,12	log2_numFilters
			5,6,7,9,10,11,13,14,15	2*log2_numFilters
		1	all subpels	log2_numFilters
		2	-	log2_numFilters
	1	0	1,2,3,4,8,12	log2_numFilters
			5,6,7,9,10,11,13,14,15	2*log2_numFilters
		1	-	log2_numFilters
B	-	-	-	log2_numFilters

The variable bitsForFilter is derived as follows.

- If slice_type is equal to P slice, and if sifo_type is equal to 0, and if both xFracL and yFracL are greather than 1, sifo_filter[i] is represented by 2* log2_numFilters bits.
- Otherwise sifo_filter[i] is represented by log2_numFilters bits.

The variable prev_filter[i] for each $(i+1)^{th}$ subpel is derived as follows.

- If the current picture is an IDR picture, prev_filter[i] is set equal to 0.
- Otherwise (the current picture is not an IDR picture), prev_filter[i] is set equal to the value of sifo_filter[i] for the previous access unit in decoding order that is of the same slice_type as the current slice_type being decoded.

zero_offset_flag eaual to 0 specifies that all the offsets values are zero.

sifo_offset_l0[i **]**, **sifo_offset_l1[** i **]** specify the offset value for i^{th} subpel location for list 0 and list 1 , respectively.

img_offset_l0[i **]**, **img_offset_l1[** i **]** specify the offset value for i^{th} reference picture in list 0 and list 1 , respectively.

5.2.12 Slice data semantics

part_size_byte_one, part_size_byte_two, part_size_byte_three, part_size_byte_four are used to specify the partition size of a V2V partitions as specified by the syntax.

end_of_slice_flag equal to 0 specifies that another macroblock is following in the slice. end_of_slice_flag equal to 1 specifies the end of the slice and that no further macroblock follows.

5.2.13 Coding unit semantic

split_coding_unit_flag specifies whether a coding unit is split into coding units with half horizontal and vertical size. When split_coding_unit_flag is not present, it shall be inferred to be equal to 0 when the current coding unit size is equal to MinCodingUnitSize otherwise it shall be inferred to be equal to 1.

5.2.14 Prediction unit semantics

skip_flag specifies whether current coding unit shall be skipped or not. If skip_flag is equal to 1, no more syntax elements except mvp_idx are parsed after skip_flag. When skip_flag is not present, it shall be inferred to be equal to 0.

mvp_idx_l0[i **] and mvp_idx_l1[** i **]** specify ith the motion vector predictor indices of list 0 and list 1, respectively, where i is the prediction unit index. When mvp_idx_l0[i] or mvp_idx_l1[i] is not present, it shall be inferred to be equal to 0.

icp_idx[i **]** specifies the illumination compensation predictor index for the current partition with partition index i. When icp_idx [i] is not present, it shall be inferred to be equal to 0. NumICPCand is the total number of illumination compensation prediction candidates for the current partition.

pred_mode specifies prediction mode of current prediction unit. The semantics of pred_mode depend on the slice type. Name association to pred_mode, PredMode, shall be done as defined in Table 5-1. When pred_mode is not present, PredMode shall be inferred to be MODE_INTRA when slice_type is equal to I slice and MODE_SKIP when slice_type is equal to P slice or B slice.

Table 5-13 Specification of prediction mode

slice_type	pred_mode	Name association to pred_mode (PredMode)
I	inferred	MODE_INTRA
P and B	0	MODE_DIRECT
	1	MODE_INTER
	2	MODE_INTRA
	inferred	MODE_SKIP

inter_partitioning_idc specifies partitioning type for current coding unit for the purpose of inter prediction. Total number of prediction units within the current prediction unit shall be obtained by NumPuParts syntax function with inter_partitioning_idc as an argument.

Table 5-14 Specification of partitioning type

inter_ partioning_ idc	Name association to inter_partioning_idc (PartMode)	Number of Partitions (NumPuParts)	PuPartWidth( ) 0) first partition 1) second partition	PuPartHeight( ) 0) first partition 1) second partition
0	PART_2Nx2N	1	PuSize	PuSize
1	PART_2NxN	2	PuSize	PuSize / 2
2	PART_Nx2N	2	PuSize / 2	PuSize
3	PART_NxN	4	PuSize / 2	PuSize / 2
4	PART_2NxnU	2	PuSize	0) PuSize / 4 1) PuSize * 3 / 4
5	PART_2NxnD	2	PuSize	0) PuSize * 3 / 4 1) PuSize / 4
6	PART_nLx2N	2	0) PuSize / 4 1) PuSize * 3 / 4	PuSize
7	PART_nRx2N	2	0) PuSize * 3 / 4 1) PuSize / 4	PuSize

Table 5-15 Specification of offset for each partitioning type

inter_ partioning_ idc	Name association to inter_partioning_idc (PartMode)	puPartIdx	PuPartOffset( ) (PuSizeInSu = PuSize / SuSize)
0	PART_2Nx2N	0	0
1	PART_2NxN	0 1	0 PuSizeInSu / 2 * PuSizeInSu
2	PART_Nx2N	0 1	0 PuSizeInSu / 2
3	PART_NxN	0 1 2 3	0 PuSizeInSu / 2 PuSizeInSu / 2 * PuSizeInSu PuSizeInSu / 2 + PuSizeInSu / 2 * PuSizeInSu
4	PART_2NxnU	0 1	0 PuSizeInSu / 4 * PuSizeInSu
5	PART_2NxnD	0 1	0 PuSizeInSu * 3 / 4 * PuSizeInSu
6	PART_nLx2N	0 1	0 PuSizeInSu / 4
7	PART_nRx2N	0 1	0 PuSizeInSu * 3 /4

merge_flag[i **]** specifies whether the inter prediction parameters for the current partition with partition index i are inferred from a neighbouring inter-predicted partition.

merge_left_flag[i **]** specifies whether the inter prediction parameters for the current partition with partition index i are inferred from the left or the top neighbouring inter-predicted partition.

inter_pred_idc[i **]** specifies whether list 0, list 1, or bi-prediction is used for the current partition with partition index i. When inter_pred_idc[i] is not present, it shall be inferred to be equal to Pred_L0 when slice_type is equal to P slice and BiPred when slice_type is equal to B slice.

ref_idx_l0[i **]**, **ref_idx_l1[** i **]** specify the list 0 and list 1 reference picture indices, respectively, for the current partition with partition index i.

mvres_l0[i **]**, **mvres_l1[** i **]** specify the list 0 and list 1 motion resolution, respectively, for the current partition with partition index i.

ref_idx_mvres_l0[i **]**, **ref_idx_mvres_l1[** i **]** specify the list 0 and list 1 reference picture indices and motion resolution, respectively, for the current partition with partition index i when entropy_coding_mode_flag is not equal to 0. The reference index and motion resolution can be calculated as

```
if(ref_idx_mvres_lX[ i ]==0)
{
    ref_idx_lX[ i ] = 0
    mvres_lX[ i ] = 1
}
else if (ref_idx_mvres_lX[ i ]==1)
{
    ref_idx_lX[ i ] = 0
    mvres_lX[ i ] = 0
}
```

```
else
{
    ref_idx_lX[ i ] =   ref_idx_mvres_lX[ i ] -1
    mvres_lX[ i ] = 1
}
```

mvd_l0[i **][** 0 **]**, **mvd_l0[** i **][** 1 **]** specify the horizontal and vertical component, respectively, of the list 1 motion vector difference for the current partition with partition index i. If mvres_l0[i] is equal to 1, mvd_l0[i][0] and mvd_l0[i][1] are in one-fourth scale. Otherwise, mvd_l0[i][0], mvd_l0[i][1] are in one-eighth scale. When any of the two components is not present, the inferred value is 0.

mvd_l1[i **][** 0 **]**, **mvd_l1[** i **][** 1 **]** specify the horizontal and vertical component, respectively, of the list 1 motion vector difference for the current partition with partition index i. If mvres_l1[i] is equal to 1, mvd_l1[i][0] and mvd_l0[i][1] are in one-fourth scale. Otherwise, mvd_l1[i][0], mvd_l1[i][1] are in one-eighth scale. When any of the two components is not present, the inferred value is 0.

icd[i **][** 0 **], icd[** i **][** 1 **], icd[** i **][** 2 **]** specify the difference values of transformed scale t0, transformed scale t1 and offset o respectively, for the current partition with partition index i. When any of the three components is not present, the inferred value is 0.

intra_split_flag specifies whether a prediction unit is split into four prediction units with half horizontal and vertical size. When intra_split_flag is not present, it shall be inferred to be equal to 0.
mode_table_idx indicates the rank of the modeIdx variable in the modeCtxList context list array. modeIdx variable is used to compute the values of the following elements: pred_mode, intra_split_flag, inter_partitioning_idc, ref_idx_l0, inter_pred_idc, split_transform_unit_flag, PU_dQP_idx.

planar_flag specifies if planar prediction mode is enabled for the prediction unit.

planar_delta_uv_present_flag specifies if the planar coding utilizes non-zero delta values for chrominance components

planar_sign specifies the sign of the planar delta variable.

planar_qdelta_indicator specifies the magnitude of the planar delta variable. From this and planar_sign value, the planar delta variable is calculated as

```
if (planar_qdelta_indicator > 21)
    Qdelta = ((planar_qdelta_indicator - 14) << 3) + 4
else if(planar_qdelta_indicator > 9)
    Qdelta = ((planar_qdelta_indicator - 6) << 2) + 2
else if (planar_qdelta_indicator > 3)
    Qdelta = (planar_qdelta_indicator - 2) << 1
else
    Qdelta = planar_qdelta_indicator
if(planar_sign == 1)
    Qdelta = -1*Qdelta
```

combined_intra_pred_flag specifies whether the combined intra prediction process is invoked for the intra-predicted samples.

prev_intra_luma_pred_flag[i] and rem_intra_luma_pred_mode[i] specifies the intra prediction mode of the ith prediction unit for luma samples with index i = 0 when intra_split_flag is equal to 0 or i =0..3 when intra_split_flag is equal to 1. When prev_intra_luma_pred_flag[i] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted prediction unit. The number of bins/bits indicating the rem_intra_luma_pred_mode[i] depends on the size of the prediction unit and limits the number of available prediction modes as follows:

PuSize	Bins/bins	Number of modes
PU_2x2	1	3
PU_4x4	4	17
PU_8x8	5	34
PU_16x16	5	34
PU_32x32	5	34
PU_64x64	2	5
PU_128x128	2	5

In the case of PU_8x8, PU_16x16 and PU_32x32 when all the 5 bins/bins are set to 1, an additional bin/bit is read to distinguish between modes 32 (the additional bin/bit set to 0) and 33 (the additional bin/bit set to 1).

intra_luma_filt_flag equal to 1 specifies that the reference samples used in intra prediction for the current prediction unit are filtered. intra_luma_filt_flag equal to 0 specifies that the reference samples are not filtered

intra_chroma_pred_mode specifies the intra prediction direction for chroma samples of the current prediction unit. The value shall be in the range of 0 to 4. 0: DC prediction, 1: vertical prediction, 2: horizontal prediction, 3: diagonal prediction and 4: same mode with IntraLumaPredMode. The diagonal prediction is specified as IntraPredType = = Intra_Vertical and IntraPredAngleID = = -8.

Table 5-1. Specification of intra prediction mode for chroma samples

intra_pred_mode_chroma	Meaning of intra_pred_mode_chroma
0	DC prediction
1	vertical prediction
2	horizontal prediction
3	diagonal prediction
4	same mode with luma samples

ztree_size[i] specifies the number of leaf nodes in zero-tree with index i.
ztree_structure[i][j] specifies whether the node j in a zero-tree with index i is a leaf node. A value 0 indicates a leaf node.
ztree_leaf_index[i][j] specifies the index in the data array that holds the zero/non-zero status for leaf node j in a zero-tree with index i.
Table 5-18 specifies the zero-trees used.

Table 5-18 Specification of Zero-trees

Index	ztree_size	ztree_structure	ztree_leaf_index	Mapping to syntax elements
0	4	{ 1, 1, 0, 0, 1, 0, 0 }	{ 0, 1, 2, 3 }	{ mvd_lx[i][0], mvd_lx[i][1], icd[i][0], icd[i][2] } x is either 0 or 1 depending on list used
1	7	{ 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0 }	{ 0, 2, 1, 3, 4, 5, 6 }	{ mvd_l0[i][0], mvd_l1[i][0], mvd_l0[i][1], mvd_l1[i][1], icd[i][0], icd[i][1], icd[i][2] }

zero_blk equal to 1 specifies that all leaf nodes of a zero-tree are zero.

ztree_node_both equal to 1 specifies that both subtrees of a zero-tree node contain nonzero leaf nodes.

ztree_node_left equal to 1 specifies that the left subtree of a zero-tree node contains nonzero leaf nodes.

5.2.15 Transform unit semantics

split_transform_unit_flag[x0][y0][layer] specifies whether a coding unit is split into coding units with half horizontal and vertical size. The array indices x0, y0 are the top-left coordinate of the block for which this syntax specifies whether it shall be split, layer gives the subdivision hierarchy level, starting with 0 at CTB level. When split_transform_unit_flag[x0][y0][layer] is not present, it shall be inferred to be equal to (currTransformUnitSize > MinTransformUnitSize ? 1 : 0).

coded_block_flag_luma specifies whether the transform unit contains one or more non-zero transform coefficient levels for the luma plane.

coded_block_flag_chroma_u specifies whether the transform unit contains one or more non-zero transform coefficient levels for the chroma (U) plane.

coded_block_flag_chroma_v specifies whether the transform unit contains one or more non-zero transform coefficient levels for the chroma (V) plane.

split_transform_unit_flag specifies whether a coding unit is split into coding units with half horizontal and vertical size. When split_transform_unit_flag is not present, it shall be inferred to be equal to (currTransformUnitSize > MinTransformUnitSize ? 1 : 0).

coded_block_flag specifies whether the transform unit contains one or more non-zero transform coefficient levels.

rotational_transform_idx specifies the rotational transform index used in the rotational transform process for the transform coefficients.
last_pos_table_idx specifies the rank of the lastPosIdx variable in the lastPosCtxList context list array. lastPosIdx variable is used to decode the position of the last non-zero coefficient in zigzag scan order given by lastPos variable, and if the magnitude of the last non-zero coefficient in zigzag scan order is greater than one given by levelMagnitudeGreaterThanOneFlag variable.
last_pos_level specifies the magnitude of the last non-zero coefficient in zigzag scan order.
last_pos_sign specifies the sign of the last non-zero coefficient in zigzag scan order.
is_level_one_run specifies jointly the values for isLevelOne and run variables. isLevelOne variable specifies if the magnitude of the next non-zero coefficient is greater than one (1) or not (0). run specifies the number of zero-coefficients before the next non-zero coefficient in reverse zigzag scan order,
level_magnitude_sign specifies the magnitude and sign of the next non-zero coefficient in reverse zigzag scan order.
level_sign specifies the sign of the next non-zero coefficient in reverse zigzag scan order.
cbp_table_idx indicates the rank of cbpIdx variable in the cbpCtxList context list array. cbpIdx variable is used to compute the coded block pattern for luminance and chrominance for certain values of mode_idx syntax element.
cbp_id0, cbp_id1, cbp_id2 are the syntax elements used to code the coded block pattern for luminance and chrominance blocks for certain values of mode_idx syntax element.

[Ed. transform coefficients...]

sig_flag equal to 1 specifies that the current transform coefficient level is not equal to 0. sig_flag equal to 0 specifies that the current transform coefficient level is equal to 0.

last_flag equal to 1 specifies that the current transform coefficent level is the last nonzero transform coefficient level in the current transform block. last_flag equal to 0 specified that the current transform coefficient level is not the last nonzero transform coefficient level in the current transform block.

abs_greater_one_flag equal to 1 specified that the absolute value of the current transform coefficient level is greater than 1. abs_greater_one_flag equal to 0 specifies that the absolute value of the current transform coefficient level is equal to 1.

coeff_abs_level plus 2 specifies the absolute value of the current transform coefficient level.

sign equal to 1 specifies that the current transform coefficient level is less than zero. sign equal to 1 specifies that the current transform coefficient level is greater than 0.

6 Decoding Process

Outputs of this process are decoded samples of the current picture (sometimes referred to by the variable CurrPic).

This clause describes the decoding process, given syntax elements and upper-case variables from clause 5.

An overview of the decoding process is given as follows:

- The processes in subclauses 6.1, 6.3, 6.4 and 6.5 specify decoding processes using syntax elements in the coding unit layer and above.
  - The intra prediction process for MODE_INTRA prediction unit as specified in subclause 6.1 has intra prediction samples as its output. The outputs are the constructed samples prior to the deblocking filter process.
  - The inter prediction process for MODE_SKIP, MODE_DIRECT and MODE_INTER prediction units is specified in subclause 6.3 with inter prediction samples being the output.
  - The transform coefficient decoding process and picture construction process prior to deblocking filter process are specified in subclause 6.4. The outputs are constructed samples prior to the deblocking filter process.
  - The constructed samples prior to the deblocking filter process that are next to the edges of transform units, prediction units and coding units are processed by a deblocking filter as specified in subclause 6.5.1. The outputs are the constructed samples prior to the adaptive loop filter process.
  - The constructed samples in the current slice prior to the adaptive in-loop filter process are processed by an adaptive loop filter as specified in subclause 6.5.2 with the output being the decoded samples.

6.1 Slice decoding process

6.1.1 Decoding process for picture order count

[Ed: Text to be taken from MPEG-4 AVC/H.264 (section 8.2.1 but without support of field coding)]

6.1.2 Decoding process for reference picture lists construction

[Ed: Text to be taken from MPEG-4 AVC/H.264 (section 8.2.4 but without support of field coding)]

6.1.3 Decoded reference picture marking process

[Ed: Text to be taken from MPEG-4 AVC/H.264 (section 8.2.5 but without support of field coding)]

6.2 Intra prediction process

This process is invoked for MODE_INTRA prediction units.

Inputs to this process are constructed samples prior to the deblocking filter process and the values of IntraPredMode from neighbouring prediction units.

The outputs are constructed luma samples of the prediction unit $pred_L$ and (when ChromaArrayType is not equal to 0) chroma prediction samples of the prediction unit $pred_C$, whether C is equal to Cb and Cr.

The decoding process for intra prediction modes are described for the luma component in subclause 6.2.1.

The decoding process for intra prediction modes are described for the chroma component in subclause 6.2.2.

Samples used in the intra prediction process shall be sample values prior to alteration by any in-loop filter operation.

6.2.1 Intra prediction process for luma samples

This process is invoked for prediction units when PredMode is equal to MODE_INTRA and planar_flag == 0.

Inputs of this process are the values of IntraPredMode (if available) from neighbouring prediction units.

Depending on the value of intra_split_flag, the luma component of prediction unit consists of one or four partitions of luma samples. The variable PartMode is inferred to PART_2Nx2N when intra_split_flag is equal to 0 and PART_NxN when intra_split_flag is equal to 1. These partitions are inverse scanned using the inverse partition scanning process specified in subclause 4.2.4.

For all prediction unit partitions, the derivation process for the IntraPredMode as specified in subclause 6.2.1.1 is invoked with puPartIdx and constructed samples prior (in decoding order) to the deblocking filter process from adjacent luma prediction units as the input and the variable IntraPredMode as the output.

For each prediction unit, the following applies.

– The intra sample prediction process in subclause 6.2.1.2 is invoked with puPartIdx and constructed samples prior (in decoding order) to the deblocking filter process from adjacent luma prediction units as the input and the output are the intra luma prediction samples $predPart_L[x, y]$ with x = 0..PuPartWidth(puPartIdx) and y = 0..PuPartHeight(puPartIdx).

– The position of the upper-left sample of a prediction unit partition with index puPartIdx inside the current prediction unit is derived by invoking the inverse prediction unit partition scanning process in subclause 4.2.4.2 with puPartIdx as the input and the output being assigned to (xO, yO) and x = 0..PuPartWidth(puPartIdx) and y = 0..PuPartHeight(puPartIdx).

$$pred_L[xO + x, yO + y] = predPart_L[x, y]$$

– The transform coefficient decoding process and picture construction process prior to deblocking filter process in subclause 6.4 is invoked with $pred_L$ and puPartIdx as the input and the constructed samples for the current prediction unit partition S'_L as the output.

6.2.1.1 Derivation process for the IntraPredMode

Inputs to this process are the index puPartIdx and variable arrays IntraPredMode (if available) that are previously (in decoding order) derived for adjacent prediction units.

Output of this process is the variable IntraPredMode[puPartIdx].

Table 6-1 specifies the values for IntraPredMode[puPartIdx] and the associated names.

Table 6-1 Specification of IntraPredMode[puPartIdx] and associated names

IntraPredMode [puPartIdx]	IntraPredType [puPartIdx]	IntraPredAngleID [puPartIdx]
0	Intra_Vertical	0
1	Intra_Horizontal	0
2	Intra_DC	-
3	Intra_Vertical	-8
4	Intra_Vertical	-4
5	Intra_Vertical	+4
6	Intra_Vertical	+8
7	Intra_Horizontal	-4
8	Intra_Horizontal	+4
9	Intra_Horizontal	+8
10	Intra_Vertical	-6
11	Intra_Vertical	-2
12	Intra_Vertical	+2
13	Intra_Vertical	+6
14	Intra_Horizontal	-6
15	Intra_Horizontal	-2
16	Intra_Horizontal	+2
17	Intra_Horizontal	+6
18	Intra_Vertical	-7
19	Intra_Vertical	-5
20	Intra_Vertical	-3

21	Intra_Vertical	-1
22	Intra_Vertical	+1
23	Intra_Vertical	+3
24	Intra_Vertical	+5
25	Intra_Vertical	+7
26	Intra_Horizontal	-7
27	Intra_Horizontal	-5
28	Intra_Horizontal	-3
29	Intra_Horizontal	-1
30	Intra_Horizontal	+1
31	Intra_Horizontal	+3
32	Intra_Horizontal	+5
33	Intra_Horizontal	+7

Table 6-2 specifies the values for MappedMostProbable according to IntraPredMode and IntraPredModeNum.

Table 6-2 Specification of MappedMostProbable according to IntraPredMode and IntraPredModeNum

IntraPredModeNum [puPartIdx] / IntraPredMode [puPartIdx]	5	17	34
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	0	5	5
6	2	6	6
7	1	7	7
8	1	8	8
9	2	9	9
10	2	10	10
11	0	11	11
12	0	12	12
13	2	13	13
14	2	14	14
15	1	15	15
16	1	16	16
17	2	2	17
18	2	3	18
19	4	10	19
20	4	11	20
21	0	0	21
22	0	0	22
23	0	12	23
24	2	5	24
25	2	13	25
26	2	14	26
27	2	7	27
28	1	15	28
29	1	1	29
30	1	1	30
31	1	16	31

32	2	8	32
33	2	2	33

IntraPredMode[puPartIdx] is derived as follows.

– The derivation process for neighbouring partitions specified in subclause 4.3.4.4 is invoked with puPartIdx given as the input and the output is assigned to lcuAddrA, puIdxA, partIdxA, lcuAddrB, puIdxB and partIdxB.

– IntraPredMode[puPartIdx] is derived by applying the following applies.

  – for N being either replaced by A or B, the variables intraPredModeN are derived as follows.

    – if the largest coding unit with address lcuAddN is not available, intraPredModeN is set equal to -1

    – else if the prediction unit with index puIdxN is not coded in intra prediction mode, intraPredModeN is set equal to Intra_DC

    – Otherwise, intraPredModeN is set equal to IntraPredMode[puPartIdxN], where IntraPredMode is the variable array assigned to the prediction unit puIdxN.

  – The following procedure applies.

```
MostProbable = Min( intraPredModeA, intraPredModeB )
if ( MostProbable >= IntraPredModeNum[ puPartIdx ] )
    MostProbable = MappedMostProbable ( IntraPredModeNum[ puPartIdx ], MostProbable )
if ( MostProbable == -1 )
    MostProbable = 2
if ( prev_intra_luma_pred_flag[ puPartIdx ] )
    IntraPredMode[ puPartIdx ] = MostProbable
else
    if ( rem_intra_luma_pred_mode[ puPartIdx ] < MostProbable )
        IntraPredMode[ puPartIdx ] = rem_intra_luma_pred_mode[ puPartIdx ]
    else
        IntraPredMode[ puPartIdx ] = rem_intra_luma_pred_mode[ puPartIdx ] + 1
```

6.2.1.2 Intra sample prediction

This process is invoked for each prediction unit partition followed by the transform decoding process and picture construction process prior to deblocking filter.

Input to this process are

– the index of the prediction unit partition puPartIdx,

– the intra prediction mode IntraPredMode[puPartIdx] derived as described in 6.2.1.1, and

– the flag intra_luma_filt_flag indicating reference sample filtering .

Outputs of this process are the prediction samples $predPart_L$[x, y], with x = 0..PuPartWidth[puPartIdx] and y = 0..PuPartHeight[puPartIdx] for the prediction unit partition with index puPartIdx.

The position of the upper-left sample of the prediction unit partition with index puPartIdx inside the current prediction unit is derived by invoking the inverse prediction unit partition scanning process in subclause 4.2.4.2 with puPartIdx as the input and the output being assigned to (xO, yO).

Let PuPartSize = max(PuPartWidth, PuPartHeight).

The PuPartSize*4 + 1 neighbouring samples p[x, y] and pf[x, y] are constructed luma samples prior to the deblocking filter process, with x = -1, y = -1..PuPartSize*2 - 1 and x = 0..PuPartSize*2 - 1, y = -1, are derived as follows.

– The luma location (xN, yN) is specified by

  xN = xO + x

  yN = yO + y

– The derivation process for neighbouring locations in 4.3.5 is invoked for luma locations with (xN, yN) as input and lcuAddrN and (xW, yW) as output.

– If lcuAddN is available, the prediction unit partition index in the largest coding unit lcuAddN covering the luma location (xW, yW) is assigned to partIdxN, otherwise, partIdxN is marked as not available.

– Each sample p[x, y] with x = -1, y = -1..PuPartSize*2 – 1 and x = 0..PuPartSize*2 – 1, y = -1 is derived as follows.

  – If any of the following conditions is true, the sample p[x, y] is marked as "not available for intra prediction"

    – lcuAddrN is not available,

    – partIdxN is not available

  – Otherwise, the sample p[x, y] is marked as "available for intra prediction" and the luma sample at luma location (xW, yW) inside the largest coding unit lcuAddrN is assigned to p[x, y].

When samples p[x, -1], with x = PuPartSize..PuPartSize*2 – 1 are marked as "not available for intra prediction," and the sample p[PuPartSize – 1, -1] is marked as "available for intra prediction," the sample value of p[PuPartSize – 1, -1] is substituted for samples values p[x, -1], with x = PuPartSize..PuPartSize*2 – 1 and samples p[x, -1], with x = PuPartSize..PuPartSize*2 – 1 are marked as "available for intra prediction."

When samples p[-1, y], with y = PuPartSize..PuPartSize*2 – 1 are marked as "not available for intra prediction," and the sample p[-1, PuPartSize – 1] is marked as "available for intra prediction," the sample value of p[-1, PuPartSize – 1] is substituted for samples values p[-1, y], with y = PuPartSize..PuPartSize*2 – 1 and samples p[-1, y], with y = PuPartSize..PuPartSize*2 – 1 are marked as "available for intra prediction."

For all remaining samples p[x, y] marked as "not available for intra prediction," the following applies.

  p[x, y] = 1 << (($BitDepth_Y$ + increased_bit_depth_luma) – 1)

If intra_luma_filt_flag == 1, the filtered neighbouring samples pf[x, y] with x = -1, y = -1..PuPartSize*2 - 1 and x = 0..PuPartSize*2 - 1, y = -1 are constructed as follows.

pf[-1, y]	= p[-1, y]	for y = 0 or y=PuPartSize*2 – 1
pf[-1, y]	= (p[-1, y + 1] + 2*p[-1, y] + p[-1, y – 1] + 2) >> 2	otherwise
pf[x, -1]	= p[x, -1]	for x = 0 or x=PuPartSize*2 – 1
pf[x, -1]	= (p[x – 1, -1] + 2*p[x, -1] + p[x + 1, -1] + 2) >> 2	otherwise
pf[-1, -1]	= (p[-1, 0] + 2*p[-1, -1] + p[0, -1] + 2) >> 2	for x = 0 and y = 0

If intra_luma_filt_flag == 0, the filtered neighbouring samples equal the unfiltered neighbouring samples.

  pf[x, y] = p[x, y]

6.2.1.2.1 Specification of intra planar prediction

This intra prediction mode is invoked when planar_flag == 1. The values of the prediction samples $predPart_L$[x, y], with x, y = 0..PuPartSize – 1, are derived by

$predPart_L$[x, y] = ((PuPartSize – (y+1))*TR(x) + (y+1)*BR(x) + (PuPartSize – (x+1))*LC(y) + (x+1)*RC(y) + PuPartSize)/(PuPartSize*2)

where the values of TR pixel array are specified as

– If the samples p[x, -1] with x = 0..PuPartSize – 1 are marked as "available for intra prediction."
TR(i) = p[i,-1] with i = 0..PuPartSize - 1

– Otherwise if the samples p[-1, y] with x = 0..PuPartSize – 1 are marked as "available for intra prediction."
TR(i) = ((PuPartSize – (i+1))*M(-1,0) + (i+1)*BRS + (PuPartSize>>1))/PuPartSize with i = 0..PuPartSize - 1

– Otherwise
TR(i) = 128 with i = 0..PuPartSize - 1

where the values of LC pixel array are specified as

– If the samples p[-1, y] with y = 0..PuPartSize – 1 are marked as "available for intra prediction."
LC(i) = p[-1, i] with i = 0..PuPartSize - 1

– Otherwise if the samples p[x, -1] with x = 0..PuPartSize – 1 are marked as "available for intra prediction."
LC(j) = ((PuPartSize – (j+1))*M(0,-1) + (j+1)*BRS + (PuPartSize>>1))/PuPartSize

– Otherwise
LC(i) = 128 with i = 0..PuPartSize - 1

where the values of BR pixel array are specified as

BR(i) = ((PuPartSize – (i+1)*LC[PuPartSize-1] + (i+1)*BRS + (PuPartSize>>1))/PuPartSize with i = 0..PuPartSize - 1

where the values of RC pixel array are specified as

RC(i) = ((PuPartSize – (i+1))*TR[PuPartSize-1] + (i+1)*BRS + (PuPartSize>>1))/PuPartSize with i = 0..PuPartSize

where the BRS value is specified as
BRS = DC + planar_delta_y
where DC is calculated identically to the DC value in DC prediction as specified in subclause 6.2.1.2.2.2.

6.2.1.2.2 Specification of directional intra prediction

For prediction units with planar_flag == 0 either intra angular prediction mode or DC prediction mode is used depending on the value of IntraPredMode[puPartIdx] variable.

6.2.1.2.2.1 Specification of intra angular prediction mode

This intra prediction mode is invoked when IntraPredType[puPartIdx] is not equal to Intra_DC.

Let intraPredType be equal to IntraPredType[puPartIdx]. intraPredAngle variable is obtained from the table below as a function of IntraPredAngleID[puPartIdx].

IntraPredAngleID[puPartIdx]	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
intraPredAngle	-32	-26	-21	-17	-13	-9	-5	-2	0	2	5	9	13	17	21	26	32

The values of the prediction samples $predPart_L$[x, y], with x, y = 0..PuPartSize – 1, are derived by the following procedure

– For k=0..(PuPartSize – 1)
```
    deltaInt = ((k+1)* intraPredAngle)/32
    deltaFract = ((k+1)*abs(intraPredAngle))%32
    if(intraPredAngle < 0)
        deltaFract = (32 – deltaFract)%32
    for l = 0..(PuPartSize – 1)
        refMainInd = l+deltaInt+1
        if(intraPredAngle >= 0)
            if(deltaFract != 0)
                p(k,l) = ((32-deltaFract)*refMain[refMainInd]+deltaFract*refMain[refMainInd+1]+16) >>
```

```
5
                else
                    p(k,l) = refMain[refMainInd]
            else
                if(refMainInd >= 0)
                    p(k,l) = ((32-deltaFract)*refMain[refMainInd]+deltaFract*refMain[refMainInd+1]+16) >>
5
                else
                    deltaIntSide = (32*32*(l+1)/abs(IntraPredAngle))>>5
                    deltaFractSide = (32*32*(l+1)/abs(IntraPredAngle))%32
                    refSideIndex = k+l-deltaIntSide
                    if(deltaFractSide)
                      p(k,l) = ((32-deltaFractSide)*refSide[refSideIndex]+
                                deltaFractSide*refSide[refSideIndex-1]+16) >> 5
                    else
                      p(k,l) = refSide[refSideIndex]
            if(IntraPredType == Intra_Vertical)
                predPart_L[ x, y ] = p(k,l)
            else
                predPart_L[ y, x ] = p(k,l)
        end for
    end for
```

where the pixel arrays refMain and refSide are specified as

– If intraPredType == Intra_Vertical

  refMain[x] = p[x, -1], with x = 0..PuPartSize – 1
  refSide[x] = p[-1, x], with x = 0..PuPartSize – 1

– Otherwise

  refMain[x] = p[-1, x], with x = 0..PuPartSize – 1
  refSide[x] = p[x, -1], with x = 0..PuPartSize – 1

6.2.1.2.2.2 Specification of Intra_DC prediction mode

This intra prediction mode is invoked when IntraPredMode[puPartIdx] is equal to Intra_DC.
Depending on the variable edge_based_prediction_flag, the following applies.

– If the value of edge_based_prediction_flag is equal to 0, the DC prediction mode process is invoked as specified in subclause 6.2.1.2.2.2.2.
– If the value of edge_based_prediction_flag is equal to 1, the edge detection process is invoked as specified in subclause 6.2.1.2.2.2.1.
  – If the output value edge_detected of the edge detection process is equal to 0, the DC prediction mode process is invoked as specified in subclause 6.2.1.2.2.2.2.
  – If the output value edge_detected of the edge detection process is equal to 1, the edge based prediction process is invoked as specified in subclause 6.2.1.2.2.2.3.

6.2.1.2.2.2.1 Specification of edge detection process

The edge detection process using the Sobel operators shall be performed on the already decoded luma samples. This process is invoked when IntraPredMode[puPartIdx] is equal to Intra_DC and when the value of edge_based_prediction_flag is equal to 1.

Inputs to this process are:
– The index of the prediction unit partition puPartIdx.
– The value of threshold_edge.

Outputs of this process are:
– The value edge_detected equal to 0 or 1 depending on the result of the Sobel operators on the already reconstructed pixels as specified in the following.
– If the value edge_detected is equal to 1, the coordinates dir[i] with i=0,1 in the horizontal and vertical direction of the detected edge as specified in the following.

The position of the upper-left sample of the prediction unit partition with index puPartIdx inside the current prediction unit is derived by invoking the inverse prediction unit partition scanning process in subclause 4.2.4.2 with puPartIdx as the input and the output being assigned to (xO_E, yO_E).

Let PuPartSize be equal to max(PuPartWidth, PuPartHeight).

The 16*PuPartSize + 16 neighbouring decoded luma samples p_E[x, y] prior the deblocking filter process with x = -4..-1, y = -4..PuPartSize*2 - 1 and x = 0..PuPartSize*2 - 1, y = -4..-1, are derived as follows.

– The luma location (xN_E, yN_E) is specified by

$xN_E = xO_E + x$

$yN_E = yO_E + y$

– The derivation process for neighbouring locations in 4.3.5 is invoked for luma locations with (xN_E, yN_E) as input and $lcuAddrN_E$ and (xW_E, yW_E) as output.

– If $lcuAddN_E$ is available, the prediction unit partition index in the largest coding unit $lcuAddN_E$ covering the luma location (xW_E, yW_E) is assigned to $partIdxN_E$, otherwise, $partIdxN_E$ is marked as not available.

– Each sample p_E[x, y] with x = -4..-1, y = -4..PuPartSize*2 – 1 and x = 0..PuPartSize*2 – 1, y = -4..-1 is derived as follows.

  – If any of the following conditions is true, the sample p_E[x, y] is marked as "not available for intra prediction"

    – $lcuAddrN_E$ is not available,

    – $partIdxN_E$ is not available

  – Otherwise, the sample p_E[x, y] is marked as "available for intra prediction" and the luma sample at luma location (xW_E, yW_E) inside the largest coding unit $lcuAddrN_E$ is assigned to p_E[x, y].

– For each sample p_E[x, y] with x = -4..-1, y = -4..PuPartSize*2 – 1 and x = 0..PuPartSize*2 – 1, y = -4..-1 marked as "available for intra prediction", the following applies:

p_E[x, y] = (p_E[x, y] + (1 << (increased_bit_depth_luma – 1))) >> increased_bit_depth_luma

Let norm_max be an integer initialized to 0.

For x = -3, -2 and y = -3..2*PuPartSize-2, and for x = -1..2*PuPartSize-2 and y = -3,-2:

– If the samples p_E[x,y], p_E[x-1,y-1], p_E[x-1,y], p_E[x-1,y+1], p_E[x,y-1], p_E[x,y+1], p_E[x+1,y-1], p_E[x+1,y] and p_E[x+1,y+1] are marked as "available for intra prediction", the values $grad_X$ and $grad_Y$ are calculated as:

$grad_X = – p_E[x-1,y-1] – 2 * p_E[x-1,y] – p_E[x-1,y+1] + p_E[x+1,y-1] + 2 * p_E[x+1,y] + p_E[x+1,y+1]$

$grad_Y = – p_E[x-1,y-1] – 2 * p_E[x,y-1] – p_E[x+1,y-1] + p_E[x-1,y+1] + 2 * p_E[x,y+1] + p_E[x+1,y+1]$

– If $|grad_X|^2+|grad_Y|^2$ > norm_max, the process for extrapolating the edge described below is invoked with the values $grad_X$, $grad_Y$, x, y, PuPartWidth and PuPartHeight as inputs.

– If the process for extrapolating the edge is invoked and the output value cross_block is equal to 1, the value – $grad_Y$ is assigned to dir[0], the value $grad_X$ is assigned to dir[1] and the value $|grad_X|^2+|grad_Y|^2$ is assigned to norm_max.

If the final value of norm_max is superior or equal to the value of (threshold_edge << 8), the value of edge_detected is set equal to 1. Otherwise, the value of edge_detected is set equal to 0 and the values of dir[0] and dir[1] are both set to 0.

<u>Process for extrapolating the edge:</u>

Inputs to the process for extrapolating the edge are

– the gradient values $grad_X$ and $grad_Y$

– the values x and y.

– the width and height of the partition unit PuPartWidth, PuPartHeight

Output of the process for extrapolating the edge is the value cross_block.

The value of cross_block is initialized to 0.

If the value of $grad_X$ is negative, the following applies:
$grad_X = -grad_X$
$grad_Y = -grad_Y$
If the value of $grad_X$ is not equal to 0, the two values $\delta x1$ and $\delta x2$ are calculated as:

$$\delta x1 = (-y) * (-grad_Y)$$

$$\delta x2 = (-y + PuPartHeight - 1) * (-grad_Y)$$

If $x * grad_X + \delta x1 \geq 0$ and $x * grad_X + \delta x1 < PuPartWidth * grad_X$, the value of cross_block is set equal to 1.
If $x * grad_X + \delta x2 \geq 0$ and $x * grad_X + \delta x2 < PuPartWidth * grad_X$, the value of cross_block is set equal to 1;

If the value of $grad_Y$ is positive, the following applies:
$grad_X = -grad_X$
$grad_Y = -grad_Y$
If the value of $grad_Y$ is not equal to 0, the two values $\delta y1$ and $\delta y2$ are calculated as:

$$\delta y1 = (-x) * (grad_X)$$

$$\delta y2 = (-x + PuPartWidth - 1) * (grad_X)$$

If $y * (-grad_Y) + \delta y1 \geq 0$ and $y * (-grad_Y) + \delta y1 < PuPartHeight * (-grad_Y)$, the value of cross_block is set equal to 1.
If $y * (-grad_Y) + \delta y2 \geq 0$ and $y * (-grad_Y) + \delta y2 < PuPartHeight * (-grad_Y)$, the value of cross_block is set equal to 1.

6.2.1.2.2.2.2 Specification of DC prediction mode

This process is invoked

– when IntraPredMode[puPartIdx] is equal to Intra_DC and

– when the value of edge_based_prediction_flag is equal to 0 or when the value of edge_based_prediction_flag is equal to 1 and the ouput value edge_detected of the edge detection process specified in subclause 6.2.1.2.2.2.1 is equal to 0.

The values of the prediction samples $predPart_L$[x, y], with x, y = 0..PuPartSize – 1, are derived as follows.

– If all samples p[x, -1], with x = 0..PuPartSize-1 and p[-1, y], with y = 0..PuPartSize-1 are marked as "available for intra prediction", the values of the prediction samples $predPart_L$[x, y], with x, y = 0..PuPartSize-1, are derived by

$$predPart_L[\,x, y\,] = \left(\sum_{x'=0}^{PuPartSize-1} p[x', -1] + \sum_{y'=0}^{PuPartSize-1} p[-1, y'] + PuPartSize \right) >> (k+1), \text{ with } x, y = 0..PuPartSize - 1$$

where 2^k = PuPartSize

– Otherwise, if any samples p[x, -1], with x = 0..PuPartSize-1 are marked as "not available for intra prediction" and all samples p[-1, y], with y = 0..PuPartSize-1 are marked as "available for intra prediction", the values of the prediction samples $predPart_L$[x, y], with x, y = 0..PuPartSize-1, are derived by

$$predPart_L[\,x, y\,] = \left(\sum_{y'=0}^{PuPartSize-1} p[-1, y'] + PuPartSize/2 \right) >> k, \text{ with } x, y = 0..PuPartSize - 1$$

where 2^k = PuPartSize

– Otherwise, if all samples p[x, -1], with x = 0..PuPartSize-1 are marked as "available for intra prediction" and any samples p[-1, y], with y = 0..PuPartSize-1 are marked as "not available for intra prediction", the values of the prediction samples $predPart_L$[x, y], with x, y = 0..PuPartSize-1, are derived by

$$predPart_L[\,x, y\,] = \left(\sum_{x'=0}^{PuPartSize-1} p[x', -1] + PuPartSize/2 \right) >> k, \text{ with } x, y = 0..PuPartSize - 1$$

where 2^k = PuPartSize

– Otherwise, the values of the prediction samples $predPart_L[\ x, y\ ]$, with x, y = 0..PuPartSize-1, are derived by

$$predPart_L[\ x, y\ ] = 1 << (\ (\ BitDepth_Y + increased_bit_depth_luma\ ) - 1\ )$$

6.2.1.2.2.2.3 Specification of edge based prediction mode

This process is invoked when IntraPredMode[puPartIdx] is equal to Intra_DC, the value of edge_based_prediction_flag is equal to 1 and the ouput value edge_detected of the edge detection process specified in subclause 6.2.1.2.2.2.1 is equal to 1.

Inputs to this process are:
– the index of the prediction unit partition puPartIdx.
– the values dir[0] and dir[1] as calculated in the edge detection process specified in subclause 6.2.1.2.2.2.1.

Outputs of this process are the prediction samples $predPart_L[\ x, y\ ]$, with x = 0..PuPartWidth[puPartIdx] and y = 0..PuPartHeight[puPartIdx] for the prediction unit partition with index puPartIdx.

The position of the upper-left sample of the prediction unit partition with index puPartIdx inside the current prediction unit is derived by invoking the inverse prediction unit partition scanning process specified in subclause 4.2.4.2 with puPartIdx as the input and the output being assigned to (xO_{EP}, yO_{EP}).

Let PuPartSize be equal to max(PuPartWidth, PuPartHeight).

The PuPartSize*4 + 1 neighbouring samples $p_{EP}[\ x, y\ ]$, that are constructed luma samples prior to the deblocking filter process, with x = -1, y = -1..PuPartSize*2 - 1 and x = 0..PuPartSize*2 - 1, y = -1, are derived as follows.

– The luma location (xN_{EP}, yN_{EP}) is specified by

$$xN_{EP} = xO_{EP} + x$$

$$yN_{EP} = yO_{EP} + y$$

– The derivation process for neighbouring locations in 4.3.5 is invoked for luma locations with (xN_{EP}, yN_{EP}) as input and $lcuAddrN_{EP}$ and (xW_{EP}, yW_{EP}) as output.

– If $lcuAddN_{EP}$ is available, the prediction unit partition index in the largest coding unit $lcuAddN_{EP}$ covering the luma location (xW_{EP}, yW_{EP}) is assigned to $partIdxN_{EP}$, otherwise, $partIdxN_{EP}$ is marked as not available.

– Each sample $p_{EP}[\ x, y\ ]$ with x = -1, y = -1..PuPartSize*2 – 1 and x = 0..PuPartSize*2 – 1, y = -1 is derived as follows.

  – If any of the following conditions is true, the sample $p_{EP}[\ x, y\ ]$ is marked as "not available for intra prediction"

    $lcuAddrN_{EP}$ is not available,

    $partIdxN_{EP}$ is not available

  – Otherwise, the sample $p_{EP}[\ x, y\ ]$ is marked as "available for intra prediction" and the luma sample at luma location (xW_{EP}, yW_{EP}) inside the largest coding unit $lcuAddrN_{EP}$ is assigned to $p_{EP}[\ x, y\ ]$.

The values of the prediction samples $predPart_L[\ x, y\ ]$, with x, y = 0..PuPartSize – 1, are derived by linear interpolation of the surrounding samples using the values dir[0] and dir[1] to define the prediction direction. The calculation is performed as follows:

– If the value of dir[0] is not equal to 0 and if at least one sample $p_{EP}[z,-1]$ with z=-1..2*PuPartSize-1 is marked as "available for intra prediction", the value δx is derived by:

$$\delta x = ((y+1) * (dir[0] << 6) + (dir[1] >> 1)) / dir[1]$$

The following four values are calculated:

$$w_a = \delta x \,\&\, (2^6 - 1)$$

$$w_b = 2^6 - w_a$$

$$floor_\delta x = \delta x >> 6$$

$$ceil_\delta x = floor_\delta x \quad \text{if} \quad w_a \quad \text{is equal to 0}$$

$$ceil_\delta x = floor_\delta x + 1 \quad \text{if} \quad w_a \quad \text{is not equal to 0}$$

If both pixels $p_{EP}[x - ceil_\delta x, -1]$ and $p_{EP}[x - floor_\delta x, -1]$ are marked as "available for intra prediction", the value predup[x,y] is marked as "calculated" and is derived by (see Figure 1 for illustration):

$$predup[x, y] = (w_a \times p_{EP}[x - ceil_\delta x, -1] + w_b \times p_{EP}[x - floor_\delta x, -1] + 32) >> 6$$

Otherwise, the value predup[x,y] is marked as "not calculated".

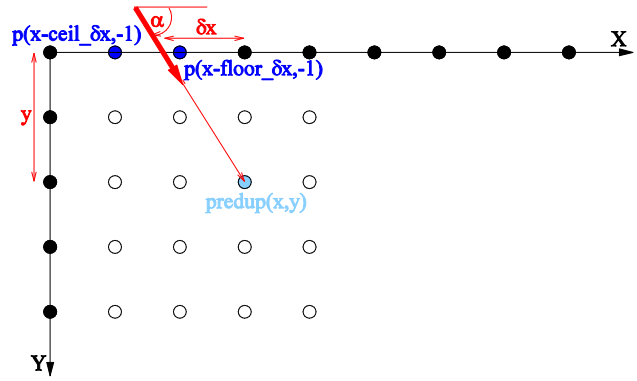


Figure 1: edge based prediction for intra coding.

– If the value of dir[1] is not equal to 0 and if at least one sample p_{EP}[-1,z] with z=-1..2*PuPartSize-1 is marked as "available for intra prediction", the value δy is derived by:

$$\delta y = ((x+1) \times (dir[1] << 6) + (dir[0] >> 1)) / dir[0]$$

The following four values are calculated:

$$w_c = \delta y \,\&\, (2^6 - 1)$$

$$w_d = 2^6 - w_c$$

$$floor_\delta y = \delta y >> 6$$

$$ceil_\delta y = floor_\delta y \quad \text{if } w_c \text{ is equal to 0}$$

$$ceil_\delta y = floor_\delta y + 1 \quad \text{if } w_c \text{ is not equal to 0}$$

If both pixels $p_{EP}[-1, y - ceil_\delta y]$ and $p_{EP}[-1, y - floor_\delta y]$ are marked as "available for intra prediction", the value predleft[x,y] is marked as "calculated" and is derived by:

$$predleft[x, y] = (w_c \times p_{EP}[-1, y - ceil_\delta y] + w_d \times p_{EP}[-1, y - floor_\delta y] + 32) >> 6$$

Otherwise, the value predleft[x,y] is marked as "not calculated".

– If both values predup[x,y] and predleft[x,y] are marked as "calculated", the prediction sample $predPart_L$[x, y] is derived by:
$predPart_L$[x, y] = (predup[x,y] + predleft[x,y]) >> 1

– If predup[x,y] is marked as "calculated" and predleft[x,y] is marked as "not calculated", the prediction sample $predPart_L$[x, y] is derived by:
$predPart_L$[x, y] = predup[x,y]

– If predleft[x,y] is marked as "calculated" and predup[x,y] is marked as "not calculated", the prediction sample $predPart_L$[x, y] is derived by:
$predPart_L$[x, y] = predleft[x,y]

– If both predup[x,y] and predleft[x,y] are marked as "not calculated", the prediction sample $predPart_L$[x, y] is derived by:
$predPart_L$[x, y] = p_{EP}[-1,-1] if x = 0, y = 0 and p_{EP}[-1,-1] is marked as "available for intra prediction"
$predPart_L$[x, y] = p_{EP}[0,-1] if x = 0, y = 0, p_{EP}[-1,-1] is marked "not available for intra prediction" and p_{EP}[0,-1] is marked as "available for intra prediction"
$predPart_L$[x, y] = p_{EP}[-1,0] if x = 0, y = 0, p_{EP}[-1,-1] is marked "not available for intra prediction" and p_{EP}[-1,0] is marked as "available for intra prediction"
$predPart_L$[x, y] = $predPart_L$[x, y-1] if x = 0
$predPart_L$[x, y] = $predPart_L$[x-1, y] if y = 0
$predPart_L$[x, y] = (2*$predPart_L$[x-1, y-1] + $predPart_L$[x, y-1] + $predPart_L$[x-1, y]) >> 2 for all other cases

6.2.2 Intra prediction process for chroma samples

This process is invoked for prediction units when PredMode is equal to MODE_INTRA.

Outputs of this process are intra prediction chroma samples for the current prediction unit partition $pred_{Cb}$[x, y] and $pred_{Cr}$[x, y].

Both chroma partitions (Cb and Cr) of the prediction unit shall use the same prediction mode. In the remainder of this subclause, chroma partition refers to one of the two chroma partitions and the subscript C is used as a replacement of the subscript Cb or Cr.

Table 6-3 specifies the associated meanings of the intra_chroma_pred_mode.

Table 6-3 Associated names of intra_chroma_pred_mode

intra_chroma_pred_mode	Name of intra_chroma_pred_mode
0	Intra_Vertical
1	Intra_Horizontal
2	Intra_DC
3	Intra_Plane
4	Intra_LumaMode

If planar_flag == 1 for the prediction unit, the intra prediction process specified in subclause 6.2.1.2.1 shall be invoked with reconstructed chroma samples as input and the intra chroma prediction samples $predPart_C$[x, y] as output.

Otherwise, if intra_chroma_pred_mode is equal to one of the values from 0 (Intra_Vertical) to 3 (Intra_Plane), the intra prediction processes specified in subclause 1.1.1.1.1.1 to 1.1.1 shall be invoked with intra_chroma_pred_mode and reconstructed chroma samples as input and the intra chroma prediction samples $predPart_C$[x, y] as output. [Ed.: intra sample prediction to be modified to allow the use chroma samples as an input]

Otherwise, if the PuPartSize equals to PU_8x8, the intra prediction processes specified in subclause 6.2.1.2.2 shall be invoked with IntraPredMode and reconstructed chroma samples as input and the intra chroma prediction samples $predPart_C$[x, y] as output.

Otherwise, if the PuPartSize > PU_8x8, the intra prediction processes specified in subclause 1.1.1.1.1.1 shall be invoked with IntraPredMode and reconstructed chroma samples as input and the intra chroma prediction samples $predPart_C$[x, y] as output.

Intra chroma prediction samples for the prediction unit are derived as follows.

– The position of the upper-left sample of a prediction unit partition with index puPartIdx inside the current prediction unit is derived by invoking the inverse prediction unit partition scanning process in subclause 4.2.4.2 with puPartIdx as the input and the output being assigned to (xO, yO).

  xC = xO / SubWidthC and yC = yO / SubHeightC

  $pred_C$[xC + x, yC + y] = $predPart_C$[x, y], x = 0..PuPartWidthC – 1 and y = 0..PuPartHeightC – 1

– The transform coefficient decoding process and picture construction process prior to deblocking filter process in subclause 6.4 is invoked with $pred_C$ and puPartIdx as the input and the constructed samples for the current prediction unit partition S'_C as the output.

6.3 Inter prediction process

This process is invoked when decoding prediction units whose PredMode is not equal to MODE_INTRA.

Outputs of this process are inter prediction samples for the current prediction unit that are a (PuSize)x(PuSize) array $pred_L$ of luma samples and when ChromaArrayType is not equal to 0 two (PuSizeC)x(PuSizeC) arrays $pred_{Cb}$ and $pred_{Cr}$ of chroma samples, one for each of the chroma components Cb and Cr.

The partitioning of a prediction unit is specified by PartMode. Each prediction unit partition is referred to by puPartIdx.

The following steps are specified for each prediction unit partition.

The functions PuPartWidth( ), PuPartHeight( ) describing the width and height of prediction unit partition are specified in Table 5-14.

The prediction unit partition index puPartIdx proceeds over values 0..NumPuParts – 1.

- If PredMode is not equal to MODE_SKIP or MODE_DIRECT, puPartWidth and puPartHeight are derived as

  puPartWidth = PuPartWidth(puPartIdx)

  puPartHeight = PuPartHeight(puPartIdx)

- Otherwise (PredMode is equal to MODE_SKIP or MODE_DIRECT), puPartIdx is set equal to 0, PartMode is set equal to PART_2Nx2N and puPartWidth and puPartHeight are derived as

  puPartWidth = PuPartWidth(0)

  puPartHeight = PuPartHeight(0)

When ChromaArrayType is not equal to 0, the variables puPartWidthC and puPartHeightC are derived as follows.

puPartWidthC = puPartWidth / SubWidthC

puPartHeightC = puPartHeight / SubHeightC

Let the variable MvCnt be initially set equal to 0 before any invocation of subclause 6.3.1 for the prediction unit.

[Ed.: are mvCnt and subMvCnt still needed?]

The inter prediction process for a prediction unit partition puPartIdx consists of the following ordered steps

1. Derivation process for motion vector components and reference indices as specified in subclause 6.3.1.

   Input to this process is a prediction unit partition puPartIdx,

   Outputs of this process are

   – luma motion vectors mvL0 and mvL1 and when ChromaArrayType is not equal to 0, the chroma motion vectors mvCL0 and mvCL1

   – reference indices refIdxL0 and refIdxL1

   – prediction list utilization flags predFlagL0 and predFlagL1

   – the prediction unit partition motion vector count subMvCnt.

2. The variable MvCnt is incremented by subMvCnt.

3. Derivation process for PBIC vector components are specified in subclause 6.3.3

   Inputs to this process are

   – the prediction unit partition index puPartIdx

   – reference indices refIdxL0 and refIdxL1

   – prediction list utilization flags predFlagL0 and predFlagL1

   Output of this process is the PBIC vector ic.

~~3.~~4. Decoding process for inter prediction samples as specified in subclause 6.3.2.

   Inputs to this process are

   – a prediction unit partition puPartIdx,

   – variables specifying partition width and height for luma and chroma (if available), partWidth, partHeight, partWidthC (if available), and partHeightC (if available)

   – luma motion vectors mvL0 and mvL1 and when ChromaArrayType is not equal to 0, the chroma motion vectors mvCL0 and mvCL1

   – PBIC vector ic

   – reference indices refIdxL0 and refIdxL1

– prediction list utilization flags predFlagL0 and predFlagL1

Outputs of this process are

– inter prediction samples (pred); which are a (puPartWidth)x(puPartHeight) array predPartL of prediction luma samples and when ChromaArrayType is not equal to 0 two (puPartWidthC)x(puPartHeightC) arrays predPartCr, and predPartCb of prediction chroma samples, one for each of the chroma components Cb and Cr.

For use in derivation processes of variables invoked later in the decoding process, the following assignments are made:

MvL0[puPartIdx] = mvL0

MvL1[puPartIdx] = mvL1

RefIdxL0[puPartIdx] = refIdxL0

RefIdxL1[puPartIdx] = refIdxL1

PredFlagL0[puPartIdx] = predFlagL0

PredFlagL1[puPartIdx] = predFlagL1

The location of the upper-left sample of the prediction unit partition to the upper-left sample of the prediction unit is derived by invoking the inverse prediction unit partition scanning process as described in subclause 4.2.4.2 with puPartIdx as the input and (xP, yP) as the output.

The prediction unit prediction is formed by placing the prediction unit partition samples in their correct relative positions in the prediction unit, as follows.

The variable $pred_L$[xP + x, yP + y] with x = 0..puPartWidth − 1, y = 0..puPartHeight − 1 is derived by

$pred_L$[xP + x, yP + y] = $predPart_L$[x, y]

When ChromaArrayType is not equal to 0, the variable $pred_C$ with x = 0.. puPartWidthC − 1, y = 0.. puPartHeightC − 1, and C in $pred_C$ and $predPart_C$ being replaced by Cb or Cr is derived by

$pred_C$[xP / SubWidthC + x, yP / SubHeightC + y] = $predPart_C$[x, y]

6.3.1 Derivation process for motion vector components and reference indices

Input to this process is a prediction partition index puPartIdx.

Outputs of this process are

- luma motion vectors mvL0 and mvL1 and when ChromaArrayType is not equal to 0, the chroma motion vectors mvCL0 and mvCL1,
- reference indices refIdxL0 and refIdxL1,
  - prediction list utilization flags predFlagL0 and predFlagL1,
  - a motion vector count variable subMvCnt.

For the derivation of the variables mvL0 and mvL1 as well as refIdxL0 and refIdxL1, the following applies.

- If PredMode is equal to MODE_SKIP or MODE_DIRECT and slice_type is equal to P, the derivation process for luma motion vectors for skipped and direct prediction units in P slices in subclause 6.3.1.1 is invoked with the output being the luma motion vectors mvL0 and reference indices refIdxL0, and predFlagL0 is set equal to 1. mvL1 and refIdxL1 are marked as not available and predFlagL1 is set equal to 0. The motion vector count variable subMvCnt is set equal to 1.
- Otherwise, if PredMode is equal to MODE_SKIP or MODE_DIRECT and slice_type is equal to B, the derivation process for luma motion vectors for skipped and direct PUs in B slices in subclause 6.3.1.2 is invoked with the output being the luma motion vectors mvL0, mvL1, the reference indices refIdxL0, refIdxL1, the motion vector count variable subMvCnt, and the prediction list utilization flags predFlagL0 and predFlagL1.

- Otherwise, if merge_flag is equal to 1, the derivation process for luma motion vectors for merge_flag equal to 1 as specified in subclause 6.3.1.3 is invoked with puPartIdx as input and the outputs being the luma motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1, the prediction utilization flags predFlagL0 and predFlagL1, and the motion vector count variable subMvCnt.
- Otherwise, for X being replaced by either 0 or 1 in the variables predFlagLX, mvLX, refIdxLX and in Pred_LX and in the syntax elements ref_idx_lX and mvd_lX, the following applies.
    1. The variables refIdxLX and predFlagLX are derived as follows.
        - If inter_pred_idc[puPartIdx] is equal to Pred_LX or to Pred_BI,

            refIdxLX = ref_idx_lX[puPartIdx]

            predFlagLX = 1
        - Otherwise, the variables refIdxLX and predFlagLX are specified by

            refIdxLX = −1

            predFlagLX = 0
    2. The motion vector count variable subMvCnt is set equal to predFlagL0 + predFlagL1.
    3. The variable mvresLX is derived as follows.

        mvresLX[0] = mvres_lX[puPartIdx][0]

        mvresLX[1] = mvres_lX[puPartIdx][1]
    4. The variable mvdLX is derived as follows.

        mvdLX[0] = mvd_lX[puPartIdx][0]

        mvdLX[1] = mvd_lX[puPartIdx][1]
    5. When predFlagLX is equal to 1, the variable mvpLX is derived as follows.
        - The derivation process for luma motion vector prediction in subclause 6.3.1.3 is invoked with puPartIdx, refIdxLX and mvdLX as the inputs and the output being mvpLX.
    6. When predFlagLX is equal to 1, the horizontal component of luma motion vector is derived as follows.
        - If mvresLX[0] is equal to 1, mvLX[0] = Round(mvpLX[0]) + mvdLX[0]*2
        - Otherwise, mvLX[0] = mvpLX[0] + mvdLX[0]
    7. When predFlagLX is equal to 1, the vertical component of luma motion vector is derived as follows.
        - If mvresLX[1] is equal to 1, mvLX[1] = Round(mvpLX[1]) + mvdLX[1]*2
        - Otherwise, mvLX[1] = mvpLX[1] + mvdLX[1]

    where Round(x) = Sign(x) * Floor(Abs(x) + 0.5).

When ChromaArrayType is not equal to 0 and predFlagLX (with X being either 0 or 1) is equal to 1, the derivation process for chroma motion vectors in subclause - is invoked with mvLX and refIdxLX as input and the output being mvCLX.

6.3.1.1 Derivation process for luma motion vectors for skipped and direct prediction units in P slices

This process is invoked when PredMode is equal to MODE_SKIP or MODE_DIRECT and slice_type is equal to P.

Outputs of this process are the motion vector mvL0 and the reference index refIdxL0.

The reference index refIdxL0 for a skipped or direct prediction unit is derived as

refIdxL0 = 0.

For the derivation of the motion vector mvL0 of a MODE_SKIP or MODE_DIRECT in P, the process specified in subclause 6.3.1.3 is invoked with puPartIdx set equal to 0, listSuffixFlag set equal to 0, the both components of mvdLX set equal to 0 as inputs and the output is assigned to mvpLX. The output is directly assigned to mvL0, since the predictor is equal to the actual motion vector.

6.3.1.2 Derivation process for luma motion vectors for skipped and direct prediction units in B slices

This process is invoked when PredMode is equal to MODE_SKIP or MODE_DIRECT and slice_type is equal to B.

Outputs of this process are the reference indices refIdxL0, refIdxL1, the motion vectors mvL0 and mvL1, the motion vector count variable subMvCnt, and the prediction list utilization flags, predFlagL0 and predFlagL1.

For X being replaced by either 0 or 1 in the variables predFlagLX, mvLX, refIdxLX, and in Pred_LX and in the syntax elements ref_idx_lX and mvd_lX, the following applies.

The variables refIdxLX and predFlagLX are derived as follows.

- If inter_pred_idc[puPartIdx] is equal to Pred_LX or to Pred_BI,

  refIdxLX = 0

  predFlagLX = 1

- Otherwise, the variables refIdxLX and predFlagLX are specified by

  refIdxLX = −1

  predFlagLX = 0

The motion vector count variable subMvCnt is set equal to predFlagL0 + predFlagL1.

When predFlagLX is equal to 1, the derivation process for luma motion vector prediction for rectangular partitions in subclause 6.3.1.3 is invoked with puPartIdx, refIdxLX, and the both components of mvdLX set equal to 0 as the inputs and the output being mvpLX. The output is directly assigned to mvLX, since the predictor is equal to the actual motion vector.

6.3.1.3 Derivation process for luma motion vectors for merge_flag equal to 1

This process is only invoked if merge_flag is equal to 1 for the current partition.

Input to this process is a prediction partition index puPartIdx.

Outputs of this process are

- the luma motion vectors mvL0 and mvL1,
- the reference indices refIdxL0 and refIdxL1,
- the prediction list utilization flags predFlagL0 and predFlagL1,
- a motion vector count variable subMvCnt.

The motion vectors mvLXA and mvLXB, the reference indices refIdxLXA and refIdxLXB, and the prediction utilization flag predFlagLXA and predFlagLXB (with X being 0 and 1) are derived as specified by the following ordered steps:

1. The derivation process for neighbouring partitions in subclause 4.3.4.4 is invoked with lcuAddr, puIdx and puPartIdx as input and the output is is assigned to lcuAddrN, puIdxN and puPartIdxN with N being replaced by A and B.
2. Depending on the partition lcuAddrN/puIdxN/puPartIdxN (with N being A and B), the following applies.
   - If the partition lcuAddrN/puIdxN/puPartIdxN is not available, predFlagL0N and predFlagL1N are set equal to 0.
   - Otherwise, predFlagL0N, predFlagL1N, mvL0N, mvL1N, refIdxL0N, and refIdxL1N are set equal to the variables predFlagL0, predFlagL1, mvL0, mvL1, refIdxL0, and refIdxL1, respectively, of the partition lcuAddrN/puIdxN/puPartIdxN.

The variable mergeLeft is derived as follows:

– If (predFlagL0B is equal to 0 and predFlagL1B is equal to 0) or merge_left_flag is equal to 1, mergeLeft is set equal to 1.

– Otherwise, mergeLeft is set equal to 0.

Depending on mergeLeft, the following applies:

– If mergeLeft is equal to 1, predFlagL0, predFlagL1, mvL0, mvL1, refIdxL0, and refIdxL1 are set equal to predFlagL0B, predFlagL1B, mvL0B, mvL1B, refIdxL0B, and refIdxL1B, respectively.

– Otherwise, predFlagL0, predFlagL1, mvL0, mvL1, refIdxL0, and refIdxL1 are set equal to predFlagL0A, predFlagL1A, mvL0A, mvL1A, refIdxL0A, and refIdxL1A, respectively.

The motion vector count variable subMvCnt is set equal to predFlagL0 + predFlagL1.

6.3.1.4 Derivation process for luma motion vector prediction

Inputs to this process are

- the prediction unit partition index puPartIdx,
- the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1),
- the motion vector difference mvdLX

Output of this process is the prediction mvpLX of the motion vector mvLX (with X being 0 or 1).

Depending on interleaved_motion_vector_predictor_flag, the following applies.

– If interleaved_motion_vector_predictor_flag is equal to 0, the following applies:

  – If mv_competition_flag is equal to 1, the derivation process for luma motion vector prediction in explicit mode in subclause 6.3.1.4.1 is invoked with puPartIdx, refIdxLX and mvdLX as the inputs and the output being mvpLX.

  – Otherwise (mv_competition_flag is equal to 0), the derivation process for luma motion vector prediction in implicit mode in subclause 6.3.1.4.6 is invoked with puPartIdx and refIdxLX as the inputs and the output being mvpLX.

– Otherwise (interleaved_motion_vector_predictor_flag is equal to 1), the following applies:

  – If mv_competition_flag is equal to 1, the derivation process for luma motion vector prediction in explicit mode and interleaved motion vector prediction in subclause 6.3.1.4.7 is invoked with puPartIdx, refIdxLX and mvdLX as the inputs and the output being mvpLX.

  – Otherwise (mv_competition_flag is equal to 0), the derivation process for luma motion vector prediction in implicit mode in subclause 6.3.1.4.6 is invoked with puPartIdx and refIdxLX as the inputs and the output being mvpLX. And the calculation of the horizontal component of mvpLX in subclause 6.3.1.4.9 is invoked with the vertical component of mvpLX, the vertical component of mvdLX as inputs and the output the horizontal component of mvpLX.

6.3.1.4.1 Derivation process for luma motion vector prediction with explicit mode

Inputs to this process are

- the PU partition index puPartIdx,
- the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1),
- the motion vector difference mvdLX

Output of this process is the prediction mvpLX of the motion vector mvLX (with X being 0 or 1).

The motion vector predictor mvpLX is derived in the following ordered steps.

1. The derivation process for motion vector predictor candidates from neighboring prediction unit partitions in subclause 6.3.1.4.2 is invoked with puPartIdx, refIdxLX (with X being 0 or 1) as the inputs and the

availability flags availableFlagLXN and the motion vectors mvLXN with N being replaced by A, B, or C as the output.

2. The derivation process for median luma motion vector prediction in subclause 6.3.1.4.3 is invoked with the motion vectors mvLXN and the availability flags availableFlagLXN with N being replaced by A, B, or C as the inputs and the output is assigned to the median motion vector predictor mvLXMed.
3. The motion vector predictor list, mvpListLX, is constructed of which elements are given as specified order:
   - mvLXMed, median motion vector of available motion vectors
   - mvLXA
   - mvLXB
   - mvLXC
4. If mv_competition_temporal_flag is equal to 1, the derivation process for temporal luma motion vector prediction in subclause 6.3.1.4.4 is invoked with puPartIdx, refIdxLX (with X being 0 or 1 for refIdxLX being refIdxL0 or refIdxL1, respectively) as the inputs and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol. If availableFlagLXCol is equal to 1, then mvLXCol is inserted at the end of mvpListLX.
5. If one of following conditions is true, mvLXMed is switched with mvLXN in the order of mvpListLX and mvLXN is derived as follows.
   - If PartMode is one of the values among PART_2NxN, PART_2NxnU and PART_2NxnD, the following applies.

     mvLXN = mvLXB for puPartIdx is equal to 0

     mvLXN = mvLXA otherwise
   - Otherwise, If PartMode is one of the values among PART_Nx2N, PART_nLx2N and PART_nRx2N, the following applies.

     mvLXN = mvLXA for puPartIdx is equal to 0

     mvLXN = mvLXC otherwise
6. If several motion vectors have same value, the motion vectors are removed from the list except the motion vector which has the smallest order in the mvpListLX
7. If the number of elements NumMVPCand(LX, puPartIdx) within the mvpListLX is greater than 1, the removal process of motion vector prediction in subclause 6.3.1.4.5 is invoked with mvpListLX and mvdLX as inputs and the output is modified mvpListLX.
8. If the number of elements NumMVPCand(LX, puPartIdx) within the mvpListLX is equal to 1, mvp_idx_lX is set by 0
9. The motion vector of mvListLX[mvp_idx_lX] is assigned to mvpLX

6.3.1.4.2 Derivation process for motion vector predictor candidates

Inputs to this process are

- the prediction unit partition index puPartIdx,
- the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1),

Outputs of this process are (with N being replaced by A, B, or C)

- the motion vectors mvLXN of the neighbouring partitions,
- the availability flags availableFlagLXN of the neighbouring partitions.

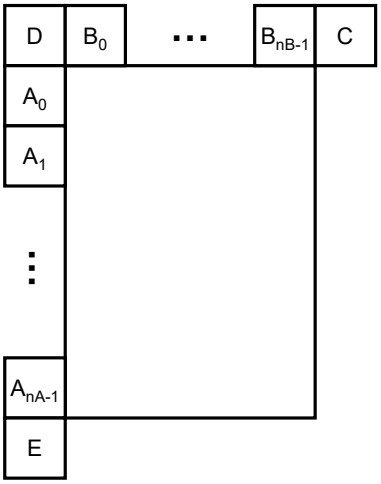


Fig. 6-1 Spatial motion vector neighbours

The motion vector mvLXA and the availability flag availableFlagLXA are derived in the following ordered steps:

1. Let a set of nA sample location (xA_k, yA_k), with k = 0..nA-1, express relative to the upper left corner of prediction unit the partition. The set of sample locations (xA_k, yA_k) represent the sample locations immediately to the left of the left partition boundary with the offset SuSize vertically. nA is set equal to puPartHeight / SuSize.

2. Let the availability flag availableFlagLXA be initially set equal to 0 and the both components of mvLXA are set equal to 0.

3. For (xA_k, yA_k) from (xA_0, yA_0) to (xA_{nN-1}, yA_{nN-1}) where $x_{Ak+1} = x_{Ak}$ and $y_{Ak+1} = y_{Ak}$ + SuSize, the following applies until availableFlagLXA is equal to 1:

   - The derivation process for the neighbouring locations specified in subclause 4.3.5 is invoked with (xA_k, yA_k) as input and the output is assigned to lcuAddrA and (xW, yW)

   - If the lcuAddrA is available,

     - The prediction unit index and the partition index in the largest coding unit lcuAddrA covering the luma location (xW, yW) shall be assigned to puIdxA and puPartIdxA.

     - If the prediction unit partition puPartIdxA is available, PredMode of puIdxA is not MODE_INTRA, predFlagLX of puIdxA is equal to 1, and the reference index refIdxLX of puIdxA is equal to the reference index refIdX of the current partition puPartIdx,

       - availableFlagLXA is set equal to 1 and the motion vector mvLXA is set equal to the motion vector mvLX which has been assigned to the prediction unit partition puPartIdxA

The motion vector mvLXB and the availability flag availableFlagLXB are derived in the following ordered steps:

1. Let a set of nB sample location (xB_k, yB_k), with k = 0..nB-1, express relative to the upper left corner of prediction unit the partition. The set of sample locations (xB_k, yB_k) represent the sample locations immediately to the above of the above partition boundary with the offset SuSize horizontally. nB is set equal to puPartWidth / SuSize.

2. Let the availability flag availableFlagLXB be initially set equal to 0 and the both components of mvLXB are set equal to 0.

3. For (xB_k, yB_k) from (xB_0, yB_0) to (xB_{nN-1}, yB_{nN-1}) where $x_{Bk+1} = x_{Bk}$ + SuSize and $y_{Bk+1} = y_{Bk}$, the following applies until availableFlagLXB is equal to 1:

   - The derivation process for the neighbouring locations specified in subclause 4.3.5 is invoked with (xB_k, yB_k) as input and the output is assigned to lcuAddrB and (xW, yW)

   - If the lcuAddrB is available,

- The prediction unit index and the partition index in the largest coding unit lcuAddrB covering the luma location (xW, yW) shall be assigned to puIdxB and puPartIdxB.
- If the prediction unit partition puPartIdxA is available, PredMode of puIdxB is not MODE_INTRA, predFlagLX of puIdxB is equal to 1, and the reference index refIdxLX of puIdxB is equal to the reference index refIdX of the current partition puPartIdx,
  - availableFlagLXB is set equal to 1 and the motion vector mvLXB is set equal to the motion vector mvLX which has been assigned to the prediction unit partition puPartIdxB

The motion vector mvLXC are derived in the following ordered steps:

3. The derivation process for neighbouring partitions in subclause 4.3.4.4 is invoked with lcuAddr, puIdx and puPartIdx as input and the output is is assigned to lcuAddrN, puIdxN and puPartIdxN with N being replaced by C, D or E.
4. Let the both components of mvLXC are set equal to 0.
5. For the derivation of the motion vectors mvLXC, the following applies:
   - If the prediction unit partition puPartIdxC is available, PredMode of puIdxC is not MODE_INTRA, predFlagLX of puPartIdxC is equal to 1, and the reference index refIdxLX of puPartIdxC is equal to the reference index refIdX of the current partition puPartIdx,
     - availableFlagLXC is set equal to 1 and mvLXC is set equal to the motion vector which has been assigned to the partition puPartIdxC
   - Otherwise, if the prediction unit partition puPartIdxD is available, PredMode of puIdxD is not MODE_INTRA, predFlagLX of puPartIdxD is equal to 1, and the reference index refIdxLX of puPartIdxD is equal to the reference index refIdX of the current partition puPartIdx,
     - availableFlagLXC is set equal to 1 and mvLXC is set equal to the motion vector which has been assigned to the partition puPartIdxD
   - Otherwise, if the prediction unit partition puPartIdxE is available, PredMode of puIdxE is not MODE_INTRA, predFlagLX of puPartIdxE is equal to 1, and the reference index refIdxLX of puPartIdxE is equal to the reference index refIdX of the current partition puPartIdx,
     - availableFlagLXC is set equal to 1 and mvLXC is set equal to the motion vector which has been assigned to the partition puPartIdxE

6.3.1.4.3 Derivation process for median luma motion vector prediction

Inputs to this process are

- the motion vectors mvLXN (with N being replaced by A, B, or C) of the neighbouring partitions,
- the availability flags availableFlagLXN with N being replaced by A, B, or C

Output of this process is the motion vector prediction mvpLXMed

For the derivation of the variables mvpLX, the following applies

- If all of the availability flags availableFlagLXN (with N being replaced by A, B, or C) are equal to 0, the both components of mvLXMed are set equal to 0.
- Otherwise, if one and only one of availableFlagLXA, availableFlagLXB and availableFlagLXC is equal to 1, availableFlagLXN is set equal to 1 and the motion vector mvLXN is assigned to the median motion vector prediction mvpLXMed

  mvLXMed = mvLXN
- Otherwise, each component of the median motion vector prediction mvLXM is given by the median of the corresponding vector components of the motion vector mvLXA, mvLXB, and mvLXC

  mvLXMed [0] = Median(mvLXA[0], mvLXB[0], mvLXC[0])

  mvLXMed [1] = Median(mvLXA[1], mvLXB[1], mvLXC[1])

6.3.1.4.4 Derivation process for temporal luma motion vector prediction

Inputs to this process are

- the prediction unit partition index puPartIdx,
- the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1),

Outputs of this process are

- the motion vector prediction mvLXCol,
- the availability flag availableFlagLXCol

The function RefPicOrderCnt(pic, refidx, LX) is specified by the value of PicOrderCnt of the picture that is the reference picture RefPicListX[refidx] of pic with X being 0 or 1. PicOrderCnt of reference pictures of pic shall be maintained until pic is not marked as "non-exisiting."

Depending on the values of slice_type and collocated_from_l0_flag, the following applies

- If slice_type is equal to B and collocated_from_l0_flag is equal to 1, mvLXCol is specified as follows.
  - The variable colPic specifies the picture that contains the co-located partition as specified by RefPicList0[0]
  - The co-located largest coding unit address lcuAddrCol, prediction unit index puIdxCol and prediction unit partition index puPartIdxCol are specified by

    lcuAddrCol = lcuAddr

    puIdxCol = puIdx

    puPartIdxCol = puPartIdx
  - If the prediction unit puIdxCol is coded in an intra prediction mode, both components of mvLXCol are set qual to 0 and availableFlagLXCol is set equal to 0.
  - Otherwise (the prediction unit puIdxCol is not coded in an intra prediction mode), the prediction utilization flag predFlagL1Col, the motion vector mvCol and the reference index refIdxCol are set equal to PredFlagL1[puPartIdxCol], MvL1[puPartIdxCol] and RefIdxL1[puPartidxCol], respectively, which are the prediction utilization flag, the motion vector mvL1 and the reference index refIdxL1 that have been assigned to the prediction unit partition puPartIdxCol.
    - If predFlagL1Col is equal to 0, both components of mvLXCol are set qual to 0 and availableFlagLXCol is set equal to 0.
    - Otherwise (predFlagL1Col is equal to 1), availableFlagLXCol is set equal to 1 and mvLXCol is derived as follows.
      - If PicOrderCnt(colPic) – RefPicOrderCnt(colPic, refIdxCol, L1)) is equal to PicOrderCnt(currPic) – RefPicOrderCnt(currPic, refIdxCol, LX)),

        mvLXCol = mvCol
      - Otherwise, mvLXCol is derived as scaled version of the motion vector mvCol as specified below

        tx = (16384 + Abs(td / 2)) / td

        DistScaleFactor = Clip3(-1024, 1023, (tb * tx + 32) >> 6)

        mvLXCol = (DistScaleFactor * mvCol + 128) >> 8

        where td and tb are derived as

        td = Clip3(-128, 127, PicOrderCnt(colPic) – RefPicOrderCnt(colPic, refIdxCol, L1))

        tb = Clip3(-128, 127, PicOrderCnt(currPic) – RefPicOrderCnt(currPic, refIdxCol, LX))
- Otherwise, if slice_type is equal to B and collocated_from_l0_flag is equal to 0, mvLXCol is specified as follows.

- The variable colPic specifies the picture that contains the co-located partition as specified by RefPicList1[0]
- The co-located largest coding unit address lcuAddrCol, prediction unit index puIdxCol and prediction unit partition index puPartIdxCol are specified by

  lcuAddrCol = lcuAddr

  puIdxCol = puIdx

  puPartIdxCol = puPartIdx
- If the prediction unit puIdxCol is coded in an intra prediction mode, both components of mvLXCol are set qual to 0 and availableFlagLXCol is set equal to 0.
- Otherwise (the prediction unit puIdxCol is not coded in an intra prediction mode), the prediction utilization flag predFlagL0Col, the motion vector mvCol and the reference index refIdxCol are set equal to PredFlagL0[puPartIdxCol], MvL0[puPartIdxCol] and RefIdxL0[puPartidxCol], respectively, which are the prediction utilization flag, the motion vector mvL0 and the reference index refIdxL0 that have been assigned to the prediction unit partition puPartIdxCol.
  - If predFlagL0Col is equal to 0, both components of mvLXCol are set qual to 0 and availableFlagLXCol is set equal to 0.
  - Otherwise (predFlagL0Col is equal to 1), availableFlagLXCol is set equal to 1 and mvLXCol is derived as follows.
    - If PicOrderCnt(colPic) – RefPicOrderCnt(colPic, refIdxCol, L0)) is equal to PicOrderCnt(currPic) – RefPicOrderCnt(currPic, refIdxCol, LX)),

      mvLXCol = mvCol
    - Otherwise, mvLXCol is derived as scaled version of the motion vector mvCol as specified below

      tx = (16384 + Abs(td / 2)) / td

      DistScaleFactor = Clip3(-1024, 1023, (tb * tx + 32) >> 6)

      mvLXCol = (DistScaleFactor * mvCol + 128) >> 8

      where td and tb are derived as

      td = Clip3(-128, 127, PicOrderCnt(colPic) – RefPicOrderCnt(colPic, refIdxCol, L0))

      tb = Clip3(-128, 127, PicOrderCnt(currPic) – RefPicOrderCnt(currPic, refIdxCol, LX))

- Otherwise (slice_type is equal to P), mvLXCol is specified as follows.
  - The variable colPic specifies the picture that contains the co-located partition as specified by RefPicList0[0]
  - The co-located largest coding unit address lcuAddrCol, prediction unit index puIdxCol and prediction unit partition index puPartIdxCol are specified by

    lcuAddrCol = lcuAddr

    puIdxCol = puIdx

    puPartIdxCol = puPartIdx
  - If the prediction unit puIdxCol is coded in an intra prediction mode, both components of mvLXCol are set qual to 0 and availableFlagLXCol is set equal to 0.
  - Otherwise (the prediction unit puIdxCol is not coded in an intra prediction mode), the prediction utilization flag predFlagL0Col, the motion vector mvCol and the reference index refIdxCol are set equal to PredFlagL0[puPartIdxCol], MvL0[puPartIdxCol] and RefIdxL0[puPartidxCol], respectively, which are the prediction utilization flag, the motion vector mvL0 and the reference index refIdxL0 that have been assigned to the prediction unit partition puPartIdxCol.
    - If predFlagL0Col is equal to 0, both components of mvLXCol are set qual to 0 and availableFlagLXCol is set equal to 0.

– Otherwise (predFlagL0Col is equal to 1), availableFlagLXCol is set equal to 1 and mvLXCol is derived as follows.

  – If PicOrderCnt(colPic) – RefPicOrderCnt(colPic, refIdxCol, L0)) is equal to PicOrderCnt(currPic) – RefPicOrderCnt(currPic, refIdxCol, LX)),

    mvLXCol = mvCol

  – Otherwise, mvLXCol is derived as scaled version of the motion vector mvCol as specified below

    tx = (16 384 + Abs(td / 2)) / td

    DistScaleFactor = Clip3(-1024, 1023, (tb * tx + 32) >> 6)

    mvLXCol = (DistScaleFactor * mvCol + 128) >> 8

  where td and tb are derived as

    td = Clip3(-128, 127, PicOrderCnt(colPic) – RefPicOrderCnt(colPic, refIdxCol, L0))

    tb = Clip3(-128, 127, PicOrderCnt(currPic) – RefPicOrderCnt(currPic, refIdxCol, LX))

6.3.1.4.5 Removal process for motion vector prediction

Inputs to this process are

- the motion vector predictor list mvpListLX,
- the motion resolution mvresLX,
- the motion vector difference mvdLX

Output of this process is the modified motion vector predictor list mvpListLX.

The function EstMvdBits(mvd) of a motion vector difference mvd is specified as follows:

EstMvdBits(mvd) = EstMvdCompBits(mvd [0]) + EstMvdCompBits(mvd [1])

The function EstMvdCompBits(mvd[c]) of a component of a motion vector difference mvd (with c being 0 or 1), used in the above formula is defined as the value of variable estMvdCompBits derived as specified by the following pseudo-code: [Ed.: to be tabularized]

```
estMvdCompBits = 1
uiTemp = ( mvd[ c ] <= 0 ) ? ( -mvd[ c ] << 1 ) + 1 : ( mvd[ c ] << 1 )
while ( 1 != uiTemp ) {
    uiTemp >>= 1
    estMvdCompBits += 2
}
```

The variable currentEstMvdBits is set equal to EstMvdBits(mvdLX).

For each mvpCandidate in mvpListLX, the following applies

- If mvresLX is equal to 0, the variable currentEstMvdBits is set equal to EstMvdBits(mvdLX) and for each mvpCandidate in mvpListLX, the following applies

  - The variable motion vector mvCandidate is derived as

    mvCandidate[0] = Round(mvpCandidate[0])/2 + mvdLX[0]

    mvCandidate[1] = Round(mvpCandidate[1])/2+ mvdLX[1]

  - For each motion vector canndidate mvpOtherCandidate in mvpListLX except mvpCandidate, the variable motion vector difference mvdOtherCandidate is derived as

    mvdOtherCandidate [0] =(mvCandidate [0] - Round(mvpOtherCandidate [0])/2)

    mvdOtherCandidate [1] = (mvCandidate [1] - Round(mvpOtherCandidate [1])/2)

- If any of EstMvdBits(mvdOtherCandidate) in mvpListX is smaller than currentEstMvdBits, mvpCandidate is removed from mvpListLX

- Otherwise, the variable currentEstMvdBits is set equal to EstMvdBits(mvdLX) and for each mvpCandidate in mvpListLX, the following applies
  - The variable motion vector mvCandidate is derived as

    mvCandidate[0] = mvpCandidate[0] + mvdLX[0]

    mvCandidate[1] = mvpCandidate[1] + mvdLX[1]
  - For each motion vector canndidate mvpOtherCandidate in mvpListLX except mvpCandidate, the variable motion vector difference mvdOtherCandidate is derived as

    mvdOtherCandidate [0] = mvCandidate [0] - mvpOtherCandidate [0]

    mvdOtherCandidate [1] = mvCandidate [1] - mvpOtherCandidate [1]
  - If any of EstMvdBits(mvdOtherCandidate) in mvpListX is smaller than currentEstMvdBits, mvpCandidate is removed from mvpListLX

6.3.1.4.6 Derivation process for luma motion vector prediction with implicit mode

[Ed.: motion vector prediction when mv_competition_flag = 0]

6.3.1.4.7 Derivation process for luma motion vector prediction with explicit mode and interleaved motion vector prediction

Inputs to this process are

- the PU partition index puPartIdx,
- the reference index of the current prediction unit partition refIdxLX (with X being 0 or 1),
- the motion vector difference mvdLX

Output of this process is the prediction mvpLX of the motion vector mvLX (with X being 0 or 1).

The motion vector predictor mvpLX is derived in the following ordered steps.

1. The derivation process for motion vector predictor candidates from neighboring prediction unit partitions in subclause 6.3.4.4.5 is invoked with puPartIdx, refIdxLX (with X being 0 or 1) as the inputs and the availability flags availableFlagLXN and the motion vectors mvLXN with N being replaced by A, B, or C as the output.
2. The derivation process for median luma motion vector prediction in subclause 6.3.4.4.6 is invoked with the motion vectors mvLXN and the availability flags availableFlagLXN with N being replaced by A, B, or C as the inputs and the output is assigned to the median motion vector predictor mvLXMed.
3. The motion vector predictor list, mvpListLX, is constructed of which elements are given as specified order:
   - mvLXMed, median motion vector of available motion vectors
   - mvLXA
   - mvLXB
   - mvLXC
4. When mv_competition_temporal_flag is equal to 1, the derivation process for temporal luma motion vector prediction in subclause 6.3.4.4.7 is invoked with puPartIdx, refIdxLX (with X being 0 or 1 for refIdxLX being refIdxL0 or refIdxL1, respectively) as the inputs and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol. If availableFlagLXCol is equal to 1, then mvLXCol is inserted at the end of mvpListLX.
5. When one of following conditions is true, mvLXMed is switched with mvLXN in the order of mvpListLX and mvLXN is derived as follows.

- If PartMode is one of the values among PART_2NxN, PART_2NxnU and PART_2NxnD, the following applies.

  mvLXN = mvLXB for puPartIdx is equal to 0

  mvLXN = mvLXA otherwise

- Otherwise, If PartMode is one of the values among PART_Nx2N, PART_nLx2N and PART_nRx2N, the following applies.

  mvLXN = mvLXA for puPartIdx is equal to 0

  mvLXN = mvLXC otherwise

6. When several motion vectors have same value, the motion vectors are removed from the list except the motion vector which has the smallest order in the mvpListLX
7. When the number of elements NumMVPCand(LX, puPartIdx) within the mvpListLX is greater than 1, the removal process of vertical component of motion vector prediction in subclause 6.3.1.4.8 is invoked with mvpListLX and mvdLX as inputs and the output is modified mvpListLX.
8. When the number of elements NumMVPCand(LX, puPartIdx) within the mvpListLX is equal to 1, mvp_idx_lX is set by 0
9. The calculation of the horizontal component of mvpLX in subclause 6.3.1.4.9 is invoked with the vertical component of mvpLX, the vertical component of mvdLX as inputs and the output the horizontal component of mvpLX.
10. The motion vector of mvListLX[mvp_idx_lX] is assigned to mvpLX

6.3.1.4.8 Removal process for vertical component of motion vector prediction

Inputs to this process are

- the motion vector predictor list mvpListLX,
- the motion resolution mvresLX,
- the motion vector difference mvdLX

Output of this process is the modified motion vector predictor list mvpListLX.

The function EstMvdCompBits(mvd[c]) of a component of a motion vector difference mvd (with c being 0 or 1), used in the above formula is defined as the value of variable estMvdCompBits derived as specified by the following pseudo-code: [Ed.: to be tabularized]

```
estMvdCompBits = 1
uiTemp = ( mvd[ c ] <= 0 ) ? ( -mvd[ c ] << 1 ) + 1 : ( mvd[ c ] << 1 )
while ( 1 != uiTemp ) {
    uiTemp >>= 1
    estMvdCompBits += 2
}
```

The variable currentEstMvdVerBits is set equal to EstMvdCompBits(mvdLXver).

For each vertical component mvpCandidateVer in mvpListLX, the following applies

- If the vertical component of mvresLX is equal to 0, the variable currentEstMvdVerBits is set equal to EstMvdCompBits (vertical component of mvdLX) and for each mvpCandidate in mvpListLX, the following applies
  - The variable vertical component of motion vector mvCandidateVer is derived as

    mvCandidateVer [0] = Round(mvpCandidateVer [0])/2 + mvdLXver[0]

    mvCandidateVer [1] = Round(mvpCandidateVer [1])/2+ mvdLXver[1]

– For each vertical component of motion vector candidate mvpOtherCandidateVer in mvpListLX except mvCandidateVer, the variable vertical component of motion vector difference mvdOtherCandidateVer is derived as

mvdOtherCandidateVer [0] =(mvCandidateVer [0] - Round(mvpOtherCandidateVer [0])/2)

mvdOtherCandidateVer [1] = (mvCandidateVer [1] - Round(mvpOtherCandidateVer [1])/2)

– When any of EstMvdCompBits(mvpOtherCandidateVer) in mvpListX is smaller than currentEstMvdVerBits, mvpCandidateVer is removed from mvpListLX

– Otherwise, the variable currentEstMvdVerBits is set equal to EstMvdCompBits(mvdLXver) and for each mvpCandidateVer in mvpListLX, the following applies

– The variable vertical component of motion vector mvCandidateVer is derived as

mvCandidateVer [0] = mvCandidateVer [0] + mvdLXver[0]

mvCandidateVer [1] = mvCandidateVer [1] + mvdLXver[1]

– For each vertical component of motion vector candidate mvpOtherCandidateVer in mvpListLX except mvpCandidate, the variable vertical component of motion vector difference mvpOtherCandidateVer is derived as

mvpOtherCandidateVer [0] = mvCandidateVer [0] - mvpOtherCandidateVer [0]

mvpOtherCandidateVer [1] = mvCandidate Ver[1] - mvpOtherCandidateVer [1]

– When any of EstMvdCompBits(mvpOtherCandidateVer) in mvpListX is smaller than currentEstMvdVerBits, mvpCandidate is removed from mvpListLX.

6.3.1.4.9 Derivation process for interleaved motion vector prediction

Inputs to this process are

– the vertical component of motion vector predictor list mvpLXver,

– the vertical component of the motion vector difference mvdLXver.

Output of this process the horizontal component of mvpLXhor.

The location of the top-left luma sample of the prediction block, relative to the top-left luma sample of the picture, is denoted by (x0, y0). The width and height of the prediction block are given by currW and currH, respectively.

Given the predictor mvpLX for the vertical motion vector component, the vertical motion vector component mvpLXver of the current motion vector is reconstructed by setting currMvVer equal to (mvpLXver + mvdLXver). Then, given the reconstructed vertical motion vector component currMvVer, the predictor mvpLXhor for the horizontal motion vector component is derived as follows:

1. Let setOfBlks be the set of neighboring prediction blocks that are contained in the same slice as the current block, precede the current block in coding order, and contain a luma sample at any of the following locations:

   – (x0 − 1, y) with y = 0..currH

   – (x, y0 − 1) with x = −1..currW + 1

2. A list listMv of motion vectors is derived as follows:

   For each block blkX of the set setOfBlks, the following applies with mvX being the motion vector that is associated with blkX:

   The motion vector mvX is scaled as described above and inserted into the list listMv.

3. Let numMv be the number of entries in the list listMv. If numMv is equal to 0, the predictor mvpLXhor for the horizontal motion vector component is set equal to 0; otherwise, mvpLXhor is derived as specified in the following with listMv[k] (k = 0..numMv − 1) representing the k-th entry in the list listMv:

   a. The variable compY is initially set equal to listMv[0][1] and the variable minDelta is set equal to abs(listMv[0][1] − currMvVer), where abs( ) specifies the absolute value of the argument.

   b. For i proceeding over the range of 1 to numMv − 1, inclusive, the following applies:

When abs(listMv[i][1] − currMvVer) is less than minDelta or (abs(listMv[i][1] − currMvVer) is equal to minDelta and listMv[i][1] is less than compY), compY is set equal to listMv[i][1].

c. The list listMvCompX is derived as follows.

For i proceeding over the range of 0 to numMv − 1, inclusive, if listMv[i][1] is equal to compY, the value listMv[i][0] is inserted into the list listMvCompX.

d. Let numCompX be the number of entries in the list listMvCompX and let listMvCompX[k], with k = 0..numCompX − 1, be the entries of the list listMvCompX. Depending on numCompX, the horizontal component mvpLXhor of the motion vector predictor is derived as follows.

– If numCompX is equal to 1, mvpLXhor is set equal to listMvCompX[0].

– Otherwise, if numCompX is equal to 2, mvpLXhor is set equal to (listMvCompX[0] + listMvCompX[1] + 1) >> 1.

– Otherwise (numCompX is greater than 2), the entries in listMvCompX are sorted in increasing order and then mvpLXhor is set equal to listMvCompX[numCompX >> 1].

6.3.1.5 Derivation process chroma motion vectors

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are a luma motion vector mvLX and a reference index refIdxLX.

Output of this process is a chroma motion vector mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

The precision of the chroma motion vector components is 1 ÷ (4 * SubWidthC) horizontally and 1 ÷ (4 * SubHeightC) vertically.

NOTE – For example, when using the 4:2:0 chroma format, since the units of luma motion vectors are one-quarter luma sample units and chroma has half horizontal and vertical resolution compared to luma, the units of chroma motion vectors are one-eighth chroma sample units, i.e., a value of 1 for the chroma motion vector refers to a one-eighth chroma sample displacement. For example, when the luma vector applies to 8x16 luma samples, the corresponding chroma vector in 4:2:0 chroma format applies to 4x8 chroma samples and when the luma vector applies to 4x4 luma samples, the corresponding chroma vector in 4:2:0 chroma format applies to 2x2 chroma samples.

For the derivation of the motion vector mvCLX, the following applies.

mvCLX[0] = mvLX[0]/2

mvCLX[1] = mvLX[1]/2

6.3.2 Decoding process for inter prediction sample

Inputs to this process are

- the prediction unit partition index puPartIdx,
- variables specifyng prediction unit partition width and height for luma and chroma, puPartWidth, puPartHeight, puPartWidthC and puPartHeightC
- luma motion vectors mvL0 and mvL1 and when ChromaArrayType is not equal to 0 chroma motion vectors mvCL0 and mvCL1
- PBIC vector ic
- reference indices refIdxL0 and refIdxL1,
- prediction list utilization flags, predFlagL0 and predFlagL1

Outputs to this process are

- inter prediction samples predPart, which are a (puPartWidth)x(puPartHeight) array $predPart_L$ of prediction luma samples, and when ChromaArrayType is not equal to 0 two (puPartWidthC)x(puPartHeightC) arrays $predPart_{Cb}$, $predPart_{Cr}$ of prediction chroma samples, one for each of the chroma components Cb and Cr.

Let $predPartL0_L$ and $predPartL1_L$ be (puPartWidth)x(puPartHeight) arrays of predicted luma sample values and when ChromaArrayType is not equal to 0 $predPartL0_{Cb}$, $predPartL1_{Cb}$, $predPartL0_{Cr}$, and $predPartL1_{Cr}$ be (puPartWidthC)x(puPartHeightC) arrays of predicted chroma sample values.

When predFlagLX is equal to 1, the following applies.

- The reference picture consisting of an ordered two-dimensional array $refPicLX_L$ of luma samples and when ChromaArrayType is not equal to 0 two ordered two-dimensional arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$ of chroma samples is derived by invoking the process specified in subclause 6.3.2.1 with refIdxLX and RefPicListX given as input.
- The array $predPartLX_L$ and when ChromaArrayType is not equal to 0 the arrays $predPartLX_{Cb}$ and $predPartLX_{Cr}$ are derived by invoking the process specified in subclause 6.3.2.2 with the current partition specified by puPartIdx, the motion vectors mvLX, mvCLX (if available), and the reference arrays with $refPicLX_L$, $refPicLX_{Cb}$ (if available), and $refPicLX_{Cr}$ (if available) given as input.
- The array $predPartLX_L$ and when ChromaArrayType is not equal to 0 the arrays $predPartLX_{Cb}$ and $predPartLX_{Cr}$ are modified by invoking the process specified in clause 6.3.4 with the current partition specified by puPartIdx, the PBIC vector ic, the prediction list utilization flags predFlagL0 and predFlagL1 given as input.

For C being replaced by L, Cb (if available), or Cr (if available), the array $predPart_C$ of the prediction samples of component C is derived by invoking the process specified in subclause **Error! Reference source not found.** with the current partition specified by puPartIdx, prediction utilization flags predFlagL0 and predFlagL1, the arrays $predPartL0_C$ and $predPartL1_C$ given as input.

6.3.2.1 Reference picture selection process

[Ed.: same as H.264/AVC]

6.3.2.2 Fractional sample interpolation process

Inputs to this process are

- the prediction unit partition index puPartIdx
- the width and height puPartWidth, puPartHeight of this prediction unit partition in luma-sample units
- a luma motion vector mvLX given in one-eighth sample units.
- when ChromaArrayType is not equal to 0, a chroma motion vector mvCLX with a precision of one-(-(4*SubWidthC)-th chroma-sample units horizontally and one-(4*SubHeightC)-th chroma-sample units vertically, and
- the selected reference picture sample arrays $refPicLX_L$, and when ChromaArrayType is not equal to 0, $refPicLX_{Cb}$, and $refPicLX_{Cr}$.

Outputs of this process are

- a puPartWidth*puPartHeight array $predPartLX_L$ of prediction luma sample values
- when ChromaArrayType is not equal to 0, two (puPartWidthC)x(puPartHeightC) arrays $predPartLX_{Cb}$, and $predPartLX_{Cr}$ of prediction chroma sample values

Let (xA_L, yA_L) be the location given in full-sample units of the upper-left luma sample of the current partition given by puPartIdx relative to the upper-left luma sample location of the given two-dimensional array of luma samples.

Let (xIntL, yIntL) be a luma location given in full-sample units and (xFracL, yFracL) be an offset given in one-eighth sample units. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays $refPicLX_L$, $refPicLX_{Cb}$, and $refPicLX_{Cr}$.

For each luma sample location (0<=xL < puPartWidth, 0<=yL < puPartHeight) inside the prediction luma sample array $predPartLX_L$, the corresponding predicted luma sample value $predPartLX_L[x_L, y_L]$ is derived as specified by the following ordered steps.

- The variables $xInt_L$, $yInt_L$, $xFrac_L$, and $yFrac_L$ are derived by

$$xInt_L = xA_L + (mvLX[0] >> 3) + x_L$$

$yInt_L = yA_L + (mvLX[1] >> 3) + y_L$

$xFrac_L = mvLX[0] \& 7$

$yFrac_L = mvLX[1] \& 7$

– The luma sample value $predPartLX_L[x_L, y_L]$ is derived by invoking the process specified in subclause 6.3.2.2.1 with ($xInt_L$, $yInt_L$), ($xFrac_L$, $yFrac_L$) and $refPicLX_L$ given as input.

When ChromaArrayType is not equal to 0, the following applies.

Let ($xInt_CC$, $yInt_CC$) be a chroma location given in full-sample units and ($xFrac_CC$, $yFrac_CC$) be an offset given in one-(-(4*SubWidthC)-th chroma-sample units horizontally and one-(4*SubHeightC)-th chroma-sample units vertically. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

For each chroma sample location ($0 <= x_C < puPartWidthC$, $0 <= y_C < puPartHeightC$) inside the prediction chroma sample arrays $predPartLX_{Cb}$ and $predPartLX_{Cr}$, the corresponding prediction chroma sample values $predPartLX_{Cb}[x_C, y_C]$ and $predPartLX_{Cr}[x_C, y_C]$ are derived as specified by the following ordered steps.

1. Depending on ChromaArrayType, the variables $xInt_C$, $yInt_C$, $xFrac_C$, and $yFrac_C$ are derived as follows

   – If ChromaArrayType is equal to 1,

   $xInt_C = (xA_L / SubWidthC) + (mvCLX[0] >> 3) + x_C$

   $yInt_C = (yA_L / SubHeightC) + (mvCLX[1] >> 3) + y_C$

   $xFrac_C = mvCLX[0] \& 7$

   $yFrac_C = mvCLX[1] \& 7$

   – Otherwise, if ChromaArrayType is equal to 2,

   $xInt_C = (xA_L / SubWidthC) + (mvCLX[0] >> 3) + x_C$

   $yInt_C = (yA_L / SubHeightC) + (mvCLX[1] >> 2) + y_C$

   $xFrac_C = mvCLX[0] \& 7$

   $yFrac_C = (mvCLX[1] \& 3)<<1$

   – Otherwise, if ChromaArrayType is equal to 3,

   $xInt_C = (xA_L / SubWidthC) + (mvCLX[0] >> 2) + x_C$

   $yInt_C = (yA_L / SubHeightC) + (mvCLX[1] >> 2) + y_C$

   $xFrac_C = (mvCLX[0] \& 3)<<1$

   $yFrac_C =(mvCLX[1] \& 3)<<1$

2. Depending on ChromaArrayType, the following applies.

   – If ChromaArrayType is not equal to 3, the following applies

   - The prediction sample value $predPartLX_{Cb}[x_C, y_C]$ is derived by invoking the process specified in subclause 6.3.2.2.2 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cb}$ given as input.

   - The prediction sample value $predPartLX_{Cr}[x_C, y_C]$ is derived by invoking the process specified in subclause 6.3.2.2.2 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cr}$ given as input

   – Otherwise (ChromaArrayType is equal to 3), the following applies..

   - The prediction sample value $predPartLX_{Cb}[x_C, y_C]$ is derived by invoking the process specified in subclause 6.3.2.2.1 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cb}$ given as input.

   - The prediction sample value $predPartLX_{Cr}[x_C, y_C]$ is derived by invoking the process specified in subclause 6.3.2.2.1 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cr}$ given as input.

6.3.2.2.1 Luma sample interpolation process

Inputs to this process are

– a luma location in full-sample units ($xInt_L$, $yInt_L$),

– a luma location offset in fractional-sample units ($xFrac_L$, $yFracy_L$), and

– the luma sample array of the selected reference picture $refPicLX_L$

Outputs of this process is a predicted luma sample value $predPartLX_L[\ x_L, y_L\ ]$.

6.3.2.2.1.1 Luma sample interpolation process for mc_interpolation_idc equal to 0 and 1

Table 6-4 and Table 6-5 specify the 12-tap filter coefficients and 6-tap filter coefficients for each quarter sample position, respectively.

Table 6-4 12-tap filter coefficients for each quarter sample position

Position (xFracL or yFracL)	filterCoeff[Position][-5...6]
1	-1, 4, -7, 13, -27, 249, 36, -16, 9, -6, 3, -1
2	-1, 5, -12, 20,-40, 229, 76, -32, 16, -8, 4,-1
3	-3, 9, -15, 27,-51, 200,119,-44, 24, -14, 7, -3
4	-1, 8, -16, 24,-48, 161, 161, -48, 24, -16, 8,-1
5	-1,6,-12, 20, -40,119, 195, -44,22,-14, 6, -1
6	-1, 4, -8, 16, -32, 76, 229, -40, 20, -12, 5,-1
7	-1, 3, -6, 9, -16, 36,249, -27, 13,-7, 4, -1

Table 6-5 6-tap filter coefficients for each quarter sample position

Position (xFracL or yFracL)	filterCoeff[Position][-2...3]
1	5, -22, 247, 35, -13, 4
2	8, -32, 224, 72, -24, 8
3	11, -42, 196, 118, -36, 10
4	8, -40, 160, 160, -40, 8
5	10, -36, 196, 118, -42, 11
6	8, -24, 72, 224, -32, 8
7	4, -13, 35,247, -22, 5

If mc_interpolation_idc is equal to 0, the 12-tap filter specified in Table 6-4 shall be used and the variable M is set equal to 6, otherwise if mc_interpolation_idc is equal to 1, the 6-tap filter specified in Table 6-5 shall be used and the variable M is set equal to 3. [Ed.: other values of mc_interpolation_idc to be added]

In the type-1 filtering process in horizontal direction, the predicted value x with quarter sample position ($xFrac_L$, 0) is derived as follows where x_l are full samples aligned with x in horizontal direction.

$$x = \left(\sum_{l=-M+1}^{M} filterCoeff\,[xFrac_L][l] * x_l + 128 \right) >> 8$$

In the type-1 filtering process in vertical direction, the predicted value x with quarter sample position (0, $yFracy_L$) is derived as follows where x_l are full samples aligned with x in vertical direction.

$$x = \left(\sum_{l=-M+1}^{M} filterCoeff\,[yFrac_L][l] * x_l + 128 \right) >> 8$$

In the type-2 filtering process in horizontal direction, the predicted value x with quarter sample position ($xFrac_L$, 0) is derived as follows where x_l are full samples aligned with x in horizontal direction.

$$x = \sum_{l=-M+1}^{M} filterCoeff\,[xFrac_L][l] * x_l$$

In the type-2 filtering process in vertical direction, the predicted value x with quarter sample position ($xFrac_L$, $yFrac_L$) is derived as follows where x_l are input quarter samples with ($xFrac_L$, 0) which are aligned with x in vertical direction.

$$x = \left(\sum_{l=-M+1}^{M} filterCoeff[yFrac_L][l] * x_l + 32768 \right) >> 16$$

Depending on ($xFrac_L$, $yFracy_L$), the following applies.

– If both $xFrac_L$ and $yFracy_L$ are 0, no filtering process is performed.

– Otherwise, if $xFrac_L$ is not 0 and $yFracy_L$ is 0, type-1 filtering process in horizontal direction is performed with full samples aligned with in horizontal direction as inputs.

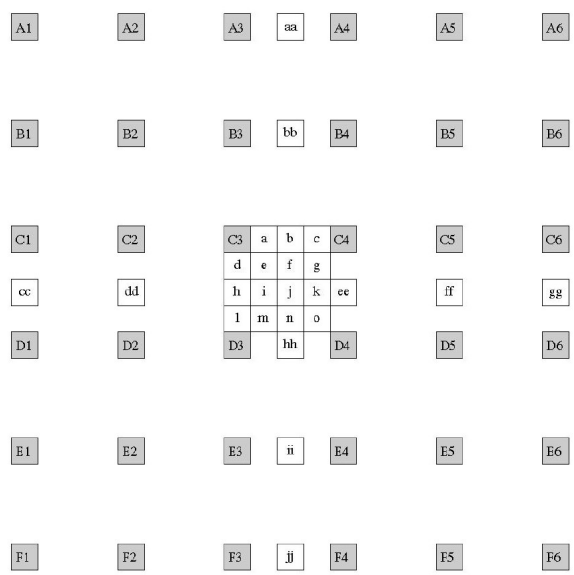
– Otherwise, if $xFrac_L$ is 0 and $yFracy_L$ is not 0, type-1 filtering process in vertical filtering is performed with full samples aligned with in vertical directions as inputs.

– Otherwise, type-2 filtering process in vertical direction is performed using quarter samples with ($xFracy_L$, 0) which were derived by type-2 filtering process in horizontal direction.

– Filtered output x shall be assigned to the prediction unit partition as

$$predPartLX_L[\, x_L, y_L \,] = Clip1_Y(\, x \,)$$

6.3.2.2.1.2 Luma sample interpolation process for mc_interpolation_idc equal to 2



Notation of image samples[Ed.: notations need to be unified between different interpolation filters]

If mc_interpolation_idc is equal to 2, the predicted value x with quarter sample position ($yFrac_L$, $xFrac_L$) is derived as follows:

($yFrac_L$,$xFrac_L$)	predicted value, x
(0,2)	(3*C1 – 15*C2 + 111*C3 + 37*C4 – 10*C5 + 2*C6 + 64)>>7
(0,4)	(3*C1 – 17*C2 + 78*C3 + 78*C4 – 17*C5 + 3*C6 + 64)>>7
(0,6)	(2*C1 – 10*C2 + 37*C3 + 111*C4 – 15*C5 + 3*C6 + 64)>>7
(2,0)	(3*A3 – 15*B3 + 111*C3 + 37*D3 – 10*E3 + 2*F3 + 64)>>7
(2,2)	(3*A1 – 15*B2 + 111*C3 + 37*D4 – 10*E5 + 2*F6 + 64)>>7
(2,4)	(3*(A1+A6) – 15*(B2+B5) + 111*(C3+C4) + 37*(D4+D3) – 10*(E5+E2) + 2*(F6+F1) + 128)>>8

(2,6)	(3*A6 – 15*B5 + 111*C4 + 37*D3 – 10*E2 + 2*F1 + 64)>>7
(4,0)	(3*A3 – 17*B3 + 78*C3 + 78*D3 – 17*E3 + 3* F3 + 64)>>7
(4,2)	(3*(A1+F1) – 15*(B2+E2) + 111*(C3+D3) + 37*(D4+C4) – 10*(E5+B5) + 2*(F6+A6) + 128)>>8
(4,4)	(5*(B3+B4+C2+C5+D2+D5+E3+E4) + 22*(C3+C4+D3+D4) + 64)>>7
(4,6)	(2*(A1+F1) – 10*(B2+E2) + 37*(C3+D3) + 111*(D4+C4) – 15*(E5+B5) + 3*(F6+A6) + 128)>>8
(6,0)	(2*A3 – 10*B3 + 37*C3 + 111*D3 – 15*E3 + 3* F3 + 64)>>7
(6,2)	(2*A6 – 10*B5 + 37*C4 + 111*D3 – 15*E2 + 3*F1 + 64)>>7
(6,4)	(2*(A1+A6) – 10*(B2+B5) + 37*(C3+C4) + 111*(D4+D3) – 15*(E5+E2) + 3*(F6+F1) + 128)>>8
(6,6)	(2*A1 – 10*B2 + 37*C3 + 111*D4 – 15*E5 + 3*F6 + 64)>>7

– Filtered output x shall be assigned to the prediction unit partition as

$$predPartLX_L[\ x_L, y_L\ ] = Clip1_Y(\ x\ )$$

For each one-eighth sample position, refer to 5.2.2.2.1.1.

6.3.2.2.1.3 Luma sample interpolation process for mc_interpolation_idc equal to 3 and 4

Table 5-5, Table 5-6 and Table 5-7 specify 6-tap filter coefficients, 8-tap filter coefficients and 12-tap filter coefficients for each quarter sample position, respectively. For each one-eighth sample position, refer to 5.2.2.2.1.1.

Table 6-5 6-tap filter coefficients for each quarter sample position

filtIdx	Position (xFracL or yFracL)	filterCoeff[Position][-2...3]
0	2	8, -32, 224, 72, -24, 8
	4	8, -40, 160, 160, -40, 8
	6	8, -24, 72, 224, -32, 8
1	2	6, -30, 222, 74, -20, 4
	4	6, -34, 156, 156, -34, 6
	6	4, -20, 74, 222, -30, 6
2	2	-6, 8, 182, 86, -12, -2
	4	-8, -8, 144, 144, -8, -8
	6	-2, -12, 86, 182, 8, -6
3	2	12, -36, 226, 70, -24, 8
	4	14, -46, 160, 160, -46, 14
	6	8, -24, 70, 226, -36, 12

Table 6-6 8-tap filter coefficients for each quarter sample position

filtIdx	Position (xFracL or yFracL)	filterCoeff[Position][-3...4]
4	2	-3, 12, -37, 229, 71, -21, 6, -1
	4	-3, 12, -39, 158, 158, -39, 12, -3
	6	-1, 6, -21, 71, 229, -37, 12, -3
5	2	-4, 16, -32, 228, 68, -28, 12, -4
	4	-1, 9, -40, 160, 160, -40, 9, -1
	6	-4, 12, -28, 68, 228, -32, 16, -4

Table 6-7 12-tap filter coefficients for each quarter sample position

filtIdx	Position (xFracL or yFracL)	filterCoeff[Position][-5...6]
6	2	-1, 5, -12, 20, -40, 229, 76, -32, 16, -8, 4, -1
	4	-1, 8, -16, 24, -48, 161, 161, -48, 24, -16, 8, -1
	6	-1, 4, -8, 16, -32, 76, 229, -40, 20, -12, 5, -1

If mc_interpolation_idc is equal to 3, 12-tap filter specified in Table 5-7 and 8-tap filter specified in Table 5-6 shall be used, otherwise if mc_interpolation_idc is equal to 4, the 6-tap filter specified in Table 5-5 shall be used.

filterCoeff points to the filtIdx according to sifo_filters[i], where i = ($xFrac_L$+($yFracy_L$<<2))>>1.

If mc_interpolation_idc is equal to 4, then filtIdx is derived as follows:

sifo_filter[i]	i=1,2,3,4,8,12	i=5,6,7,9,10,11,13,14,15
	filtIdx	(filtIdxV, filtIdxH)
0	0	0,0
1	1	1,1
2	2	2,2
3	3	3,3
4	-	0,1
5	-	0,2
6	-	0,3
7	-	1,0
8	-	1,2
9	-	1,3
10	-	2,0
11	-	2,1
12	-	2,3
13	-	3,0
14	-	3,1
15	-	3,2

If mc_interpolation_idc is equal to 3, then filtIdx is derived as follows:

sifo_filter[i]	i=1,2,3,4,8,12	i=5,6,7,9,10,11,13,14,15
	filtIdx	(filtIdxV, filtIdxH)
0	6	5,5
1	4	4,4
2	-	5,4
3	-	4,5

If mc_interpolation_idc is equal to 4, the variable M is set to 3. If mc_interpolation_idc is equal to 3 and if filtIdx is equal to 0, the variable M is set to 6, otherwise if filtIdx is greater than 0, the variable M is set to 4.

In the type-1 filtering process in horizontal direction, the predicted value x with quarter sample position ($xFrac_L$, 0) is derived as follows where x_l are full samples aligned with x in horizontal direction.

$$x = \left(\sum_{l=-M+1}^{M} \text{filterCoeff}[\text{filtIdx}][\text{xFrac}_L][l] * x_l + 128 \right) >> 8$$

In the type-1 filtering process in vertical direction, the predicted value x with quarter sample position (0, $yFracy_L$) is derived as follows where x_l are full samples aligned with x in vertical direction.

$$x = \left(\sum_{l=-M+1}^{M} \text{filterCoeff}[\text{filtIdx}][\text{yFrac}_L][l] * x_l + 128 \right) >> 8$$

In the type-2 filtering process in horizontal direction, the predicted value x with quarter sample position ($xFrac_L$, 0) is derived as follows where x_l are full samples aligned with x in horizontal direction.

$$x = \sum_{l=-M+1}^{M} \text{filterCoeff}[\text{filtIdxH}][\text{xFrac}_L][l] * x_l$$

In the type-2 filtering process in vertical direction, the predicted value x with quarter sample position ($xFrac_L$, $yFrac_L$) is derived as follows where x_l are input quarter samples with ($xFrac_L$, 0) which are aligned with x in vertical direction.

$$x = \left(\sum_{l=-M+1}^{M} filterCoeff[filtIdxV][yFrac_L][l] * x_l + 32768 \right) >> 16$$

Depending on ($xFrac_L$, $yFracy_L$), the following applies.

– If both $xFrac_L$ and $yFracy_L$ are 0, no filtering process is performed.

– Otherwise, if $xFrac_L$ is not 0 and $yFracy_L$ is 0, type-1 filtering process in horizontal direction is performed with full samples aligned with in horizontal direction as inputs.

– Otherwise, if $xFrac_L$ is 0 and $yFracy_L$ is not 0, type-1 filtering process in vertical filtering is performed with full samples aligned with in vertical directions as inputs.

– Otherwise, type-2 filtering process in vertical direction is performed using quarter samples with ($xFracy_L$, 0) which were derived by type-2 filtering process in horizontal direction.

– Filtered output x shall be assigned to the variable x0 as

$$x0 = Clip1_Y(x)$$

Depending on zero_offset_flag, the following applies.

– If zero_offset_flag is 0, the variable sifo_offset is set to 0.

– Otherwise, (if zero_offset_flag is not 0), and if predFlagLX is equal to 1, and

- if refIdxLX is equal to 0, the variable sifo_offset is set to sifo_offset_lX[i], where i = ($xFrac_L$+($yFracy_L$<<2))>>1.

- if refIdxLX is greater than 0, the variable sifo_offset is set to img_offset_lX[refIdxLX]

– Final output x0 shall be assigned to the prediction unit partition as

$$predPartLX_L[x_L, y_L] = Clip1_Y(x0 + sifo_offset).$$

6.3.2.2.1.4 Luma sample interpolation process for mc_interpolation_idc equal to 5 and 6

When the mc_interpolation_idc is equal to 5 or 6, the interpolation process consists of an IIR filtering of the reference samples followed by an FIR filtering.

Define PF(z) as a series of sub-stages:

i. F1 in Horizontal direction,
ii. F2 in Horizontal direction,
iii. F1 in Vertical direction,
iv. F2 in Vertical direction,

The sub-stages F1 and F2 are defined as,

Notation	Operation
F1	input[n] + xMult16x16(output[n-1], z), n=0,1…M-1
F2	xMult16x16(output[n+1] - input[n], z), n=M-1,..1,0
xMult16x16	(i16A * i16B + (1<<14)) >> 15

where, M is the total number of samples in the reference picture in the considered direction and z is the pole value. specified below.

When the mc_interpolation_idc is equal to 5, the IIR filtering is PF(z) with z=-12854.

When the mc_interpolation_idc is equal to 6, the IIR filtering is PF(z1) followed by PF(z2) with z1= -19612 and z2=-3067.

Next, the FIR filtering generates the interpolated value required according to the sub-sample position (xFracL, yFracL) pointed to by the motion vector.

Table 6-8 and 6-9 specify 4-tap and 6-tap filter coefficients, for each one-eighth sample position, respectively.

Table 6-8 Coefficients for 4-tap FIR filter for each one-eighth sample position for mc_interpolation_idc=5

Position (xFracL or yFracL)	filterCoeff[Position][-1...2]
0	8192, 24097, 8192, 0
1	5784, 24046, 10457,194
2	3930, 22967, 13117,467
3	2550, 21098, 15935,898
4	1566, 18674, 18674,1566
5	898, 15935, 21098, 2550
6	467, 13117, 22967,3930
7	194, 10457, 24046,5784

Table 6-9 Coefficients for 6-tap FIR filter for each one-eighth sample position for mc_interpolation_idc=6

Position (xFracL or yFracL)	filterCoeff[Position][-2...3]
0	601, 7837, 15892, 7837, 601, 0
1	351, 6418, 15743, 9292, 969, -5
2	189, 5105, 15262, 10746, 1471, -5
3	90, 3940, 14467, 12140, 2125, 6
4	33, 2946, 13405, 13405, 2946, 33
5	6, 2125, 12140, 14467, 3940, 90
6	-5, 1471, 10746, 15262, 5105, 189
7	-5, 969, 9292, 15743, 6418, 351

6.3.2.2.2 Chroma sample interpolation process

[Ed.: same as H.264/AVC]

Depending on mc_interpolation_idc, the following applies:

– If mc_interpolation_idc is equal to 5 or 6, the chroma interpolation process consists of the same steps as described in section 6.3.2.2.1.4 but operating on the chroma reference samples.

– Otherwise, the chroma interpoltaion process as specified in H.264/AVC is applied.

6.3.3 Derivation process for PBIC vector

Inputs to this process are

- the prediction unit partition index puPartIdx
- the prediction list utilization flags of the current prediction unit partition predFlagL0 and predFlagL1
- the reference indices of the current prediction unit partition refIdxL0 and refIdxL1

Output of this process is the PBIC vector ic.

For the derivation of the PBIC vector ic, the following applies:

- If PredMode is equal to MODE_SKIP or MODE_DIRECT, the derivation process for PBIC prediction in subclause 6.3.3.1 is invoked with puPartIdx, refIdxL0, refIdxL1, predFlagL0 and predFlagL1 as the inputs and icp as the output. The output icp is directly assigned to ic, since the predictor is equal to the actual PBIC vector.
- Otherwise, if merge_flag is equal to 1, the derivation process for PBIC vector when merge_flag is equal to 1 as specified in subclause 6.3.3.2 is invoked with puPartIdx as input and the PBIC vector ic as the output.

- Otherwise, the following applies:
    1. The derivation process for PBIC prediction in subclause 6.3.3.1 is invoked with puPartIdx, refIdxL0, refIdxL1, predFlagL0 and predFlagL1 as the inputs and icp as the output.
    2. The vector ic is derived as follows:
        i. ic[0] = icp[0] + icd[puPartIdx][0]
        ii. ic[1] = icp[1] + icd[puPartIdx][1]
        iii. ic[2] = icp[2] + icd[puPartIdx][2]

6.3.3.1 Derivation process for PBIC prediction

Inputs to this process are

- the prediction unit partition index puPartIdx
- the prediction list utilization flags of the current prediction unit partition predFlagL0 and predFlagL1
- the reference indices of the current prediction unit partition refIdxL0 and refIdxL1

Output of this process is the prediction icp of the PBIC vector ic.

The PBIC vector predictor icp is derived in the following ordered steps:

1. The derivation process for PBIC vector predictor candidates from neighboring prediction unit partitions in subclause 6.3.3.1.1 is invoked with puPartIdx, refIdxL0, refIdxL1, predFlagL0 and predFlagL1 as the inputs and the availability flags availableFlagN and the PBIC vectors icN with N being replaced by A, B, or C as the output.
2. A PBIC vector icDefault is constructed such that all of its elements are set to zero.
3. The PBIC vector predictor list, icpList, is constructed of which elements are given as specified order:
    - icDefault
    - icA
    - icB
    - icC
4. If both icA and icB are available and one of the following conditions is true, icA is switched with icB in the order of icpList.
    - If puPartIdx is equal to 0 and PartMode is one of the values among PART_2NxN, PART_2NxnU and PART_2NxnD.
    - If puPartIdx is equal to 1 and PartMode is one of the values among PART_Nx2N, PART_nLx2N and PART_nRx2N.
5. If several PBIC vectors have same value, the PBIC vectors are removed from the list except the PBIC vector which has the smallest order in the icpList
6. If the number of elements NumICPCand(puPartIdx) within the icpList is equal to 1, icp_idx[puPartIdx] is set to 0
7. The PBIC vector of icpList [icp_idx[puPartIdx]] is assigned to icp

6.3.3.1.1 Derivation process for PBIC vector predictor candidates

Inputs to this process are

- the prediction unit partition index puPartIdx
- the prediction list utilization flags of the current prediction unit partition predFlagL0 and predFlagL1
- the reference indices of the current prediction unit partition refIdxL0 and refIdxL1

Outputs of this process are (with N being replaced by A, B, or C)

- the PBIC vectors icN of the neighbouring partitions
- the availability flags availableFlagN of the neighbouring partitions

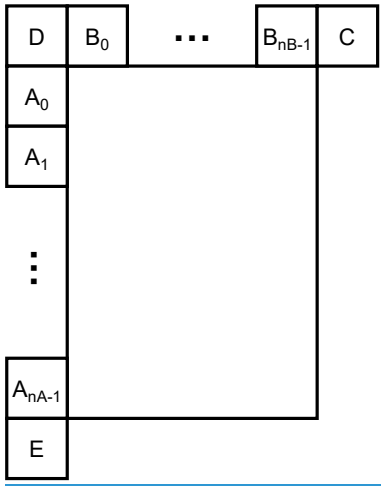


Fig. 6-1 Spatial motion vector neighbours

The PBIC vector icA and the availability flag availableFlagA are derived in the following ordered steps:

1. Let a set of nA sample location (xA_k, yA_k), with k = 0..nA-1, express relative to the upper left corner of the prediction unit partition. The set of sample locations (xA_k, yA_k) represent the sample locations immediately to the left of the left partition boundary with the offset SuSize vertically. nA is set equal to puPartHeight / SuSize.
2. Let the availability flag availableFlagA be initially set equal to 0 and all the components of icA be set equal to 0.
3. For (xA_k, yA_k) from (xA_0, yA_0) to (xA_{nN-1}, yA_{nN-1}) where $x_{Ak+1} = x_{Ak}$ and $y_{Ak+1} = y_{Ak}$ + SuSize, the following applies until availableFlagA is equal to 1:
   - The derivation process for the neighbouring locations specified in subclause 4.3.5 is invoked with (xA_k, yA_k) as input and the output is assigned to lcuAddrA and (xW, yW)
   - If the lcuAddrA is available,
     - The prediction unit index and the partition index in the largest coding unit lcuAddrA covering the luma location (xW, yW) shall be assigned to puIdxA and puPartIdxA.
     - If the prediction unit partition puPartIdxA is available, PredMode of puIdxA is not MODE_INTRA, predFlagL0 of puIdxA is equal to predFlagL0 of the current partition puPartIdx, reference index refIdxL0 of puIdxA is equal to reference index refIdxL0 of the current partition puPartIdx, predFlagL1 of puIdxA is equal to predFlagL1 of the current partition puPartIdx, reference index refIdxL1 of puIdxA is equal to reference index refIdxL1 of the current partition puPartIdx,
       - availableFlagA is set equal to 1 and the PBIC vector icA is set equal to the PBIC vector ic which has been assigned to the prediction unit partition puPartIdxA

The PBIC vector icB and the availability flag availableFlagB are derived in the following ordered steps:

4. Let a set of nB sample location (xB_k, yB_k), with k = 0..nB-1, express relative to the upper left corner of the prediction unit partition. The set of sample locations (xB_k, yB_k) represent the sample locations immediately to the above of the above partition boundary with the offset SuSize horizontally. nB is set equal to puPartWidth / SuSize.
5. Let the availability flag availableFlagB be initially set equal to 0 and all the components of icB be set equal to 0.
6. For (xB_k, yB_k) from (xB_0, yB_0) to (xB_{nN-1}, yB_{nN-1}) where $x_{Bk+1} = x_{Bk}$ + SuSize and $y_{Bk+1} = y_{Bk}$, the following applies until availableFlagLXB is equal to 1:

- The derivation process for the neighbouring locations specified in subclause 4.3.5 is invoked with (xB_k, yB_k) as input and the output is assigned to lcuAddrB and (xW, yW)
- If the lcuAddrB is available,
  - The prediction unit index and the partition index in the largest coding unit lcuAddrB covering the luma location (xW, yW) shall be assigned to puIdxB and puPartIdxB.
  - If the prediction unit partition puPartIdxB is available, PredMode of puIdxB is not MODE_INTRA, predFlagL0 of puIdxB is equal to predFlagL0 of the current partition puPartIdx, reference index refIdxL0 of puIdxB is equal to reference index refIdxL0 of the current partition puPartIdx, predFlagL1 of puIdxB is equal to predFlagL1 of the current partition puPartIdx, reference index refIdxL1 of puIdxB is equal to reference index refIdxL1 of the current partition puPartIdx,
    - availableFlagB is set equal to 1 and the PBIC vector icB is set equal to the PBIC vector ic which has been assigned to the prediction unit partition puPartIdxB

The PBIC vector icC and the availability flag availableFlagC are derived in the following ordered steps:

6. The derivation process for neighbouring partitions in subclause 4.3.4.4 is invoked with lcuAddr, puIdx and puPartIdx as input and the output is is assigned to lcuAddrN, puIdxN and puPartIdxN with N being replaced by C, D or E.
7. Let the availability flag availableFlagC be initially set equal to 0 and all the components of icC be set equal to 0.
8. For the derivation of the availability flag availableFlagC and the PBIC vector icC, the following applies:
   - If the prediction unit partition puPartIdxC is available, PredMode of puIdxC is not MODE_INTRA, predFlagL0 of puIdxC is equal to predFlagL0 of the current partition puPartIdx, reference index refIdxL0 of puIdxC is equal to reference index refIdxL0 of the current partition puPartIdx, predFlagL1 of puIdxC is equal to predFlagL1 of the current partition puPartIdx, reference index refIdxL1 of puIdxC is equal to reference index refIdxL1 of the current partition puPartIdx,
     - availableFlagC is set equal to 1 and the PBIC vector icC is set equal to the PBIC vector ic which has been assigned to the prediction unit partition puPartIdxC
   - Otherwise, if the prediction unit partition puPartIdxE is available, PredMode of puIdxE is not MODE_INTRA, predFlagL0 of puIdxE is equal to predFlagL0 of the current partition puPartIdx, reference index refIdxL0 of puIdxE is equal to reference index refIdxL0 of the current partition puPartIdx, predFlagL1 of puIdxE is equal to predFlagL1 of the current partition puPartIdx, reference index refIdxL1 of puIdxE is equal to reference index refIdxL1 of the current partition puPartIdx,
     - availableFlagC is set equal to 1 and the PBIC vector icC is set equal to the PBIC vector ic which has been assigned to the prediction unit partition puPartIdxE
   - Otherwise, if the prediction unit partition puPartIdxD is available, PredMode of puIdxD is not MODE_INTRA, predFlagL0 of puIdxD is equal to predFlagL0 of the current partition puPartIdx, reference index refIdxL0 of puIdxD is equal to reference index refIdxL0 of the current partition puPartIdx, predFlagL1 of puIdxD is equal to predFlagL1 of the current partition puPartIdx, reference index refIdxL1 of puIdxD is equal to reference index refIdxL1 of the current partition puPartIdx,
     - availableFlagC is set equal to 1 and the PBIC vector icC is set equal to the PBIC vector ic which has been assigned to the prediction unit partition puPartIdxD

6.3.3.2 Derivation process for PBIC vector when merge_flag is equal to 1

This process is only invoked if merge_flag is equal to 1 for the current partition.

Input to this process is the prediction unit partition index puPartIdx.

Output of this process is the PBIC vector ic.

The PBIC vector ic is derived as specified by the following ordered steps:

1. The derivation process for neighbouring partitions in subclause 4.3.4.4 is invoked with lcuAddr, puIdx and puPartIdx as input and the output is is assigned to lcuAddrN, puIdxN and puPartIdxN with N being replaced by A and B.

2. Depending on the partition lcuAddrN/puIdxN/puPartIdxN (with N being A and B), the following applies.

   – If the partition lcuAddrN/puIdxN/puPartIdxN is not available, predFlagL0N and predFlagL1N are set equal to 0.

   – Otherwise, predFlagL0N, predFlagL1N and icN are set equal to the variables predFlagL0, predFlagL1 and ic, respectively, of the partition lcuAddrN/puIdxN/puPartIdxN.

3. The variable mergeLeft is derived as follows:

   – If (predFlagL0B is equal to 0 and predFlagL1B is equal to 0) or merge_left_flag is equal to 1, mergeLeft is set equal to 1.

   – Otherwise, mergeLeft is set equal to 0.

4. Depending on mergeLeft, the following applies:

   – If mergeLeft is equal to 1, icB is assigned to ic.

   – Otherwise, icA is assigned to ic.

6.3.4 Application of PBIC

Inputs to this process are

– the prediction unit partition index puPartIdx

– a PBIC vector ic

– prediction list utilization flags, predFlagL0 and predFlagL1

– a puPartWidth*puPartHeight array $predPartLX_L$ of prediction luma sample values

– when ChromaArrayType is not equal to 0, two (puPartWidthC)x(puPartHeightC) arrays $predPartLX_{Cb}$, and $predPartLX_{Cr}$ of prediction chroma sample values

Outputs of this process are

– modified array $predPartLX_L$ of prediction luma sample values

– when ChromaArrayType is not equal to 0, modified arrays $predPartLX_{Cb}$ and $predPartLX_{Cr}$ of prediction chroma sample values

The application of PBIC is done as specified by the following ordered steps:

1. IC_SCALE_PREC is set to 6

2. The icScaleLX and icOffsetLX are derived as follows:

   – If both predFlagL0 and predFlagL1 are equal to 1, then the following applies:

      o icScaleLX = ((1<< IC_SCALE_PREC) + ic[0]) + (ic[1] << 1), if X = 0

      o icScaleLX = ((1<< IC_SCALE_PREC) + ic[0]) – (ic[1] << 1), if X = 1

      o icOffsetLX = ic[2]

   – Otherwise, the following applies:

      o icScaleLX = ((1<< IC_SCALE_PREC) + ic[0])

      o icOffsetLX = ic[2]

~~1.~~3. For each luma sample location (0 <= xL < puPartWidth, 0 <= yL < puPartHeight) inside the prediction luma sample array $predPartLX_L$, the corresponding predicted luma sample value $predPartLX_L[x_L, y_L]$ is modified as specified by the following ordered steps:

   a. $predPartLX_L[x_L, y_L]$ = (icScaleLX * $predPartLX_L[x_L, y_L]$ + (1 << (IC_SCALE_PREC – 1))) >> IC_SCALE_PREC

b. predPartLX$_L$[x$_L$, y$_L$] = predPartLX$_L$[x$_L$, y$_L$] + icOffsetLX

c. predPartLX$_L$[x$_L$, y$_L$] = Clip1$_Y$(predPartLX$_L$[x$_L$, y$_L$])

4. When ChromaArrayType is not equal to 0, the following applies:

– For each chroma sample location (0 <= x$_C$ < puPartWidthC, 0 <= y$_C$ < puPartHeightC) inside the prediction chroma sample array predPartLX$_N$ (for both N being Cb and Cr), the corresponding prediction chroma sample value predPartLX$_N$[x$_C$, y$_C$] is modified as specified by the following ordered steps:

a. predPartLX$_N$[x$_C$, y$_C$] = predPartLX$_N$[x$_C$, y$_C$] – (1 << (BitDepth$_C$ – 1))

b. predPartLX$_N$[x$_C$, y$_C$] = (icScaleLX * predPartLX$_N$[x$_C$, y$_C$] + (1 << (IC_SCALE_PREC – 1))) >> IC_SCALE_PREC

c. predPartLX$_N$[x$_C$, y$_C$] = predPartLX$_N$[x$_C$, y$_C$] + (1 << (BitDepth$_C$ – 1))

d. predPartLX$_N$[x$_C$, y$_C$] = Clip1$_C$(predPartLX$_N$[x$_C$, y$_C$])

6.4 Transform coefficient decoding process and picture construction process

Inputs to this process are LumaLevel (if available), ChromaLevel (if available), and available inter or intra prediction sample arrays for the current transform unit for the applicable components pred$_L$, pred$_{Cb}$, or pred$_{Cr}$.

Outputs of this process are the constructed sample arrays prior to the deblocking filter process for the applicable components S'$_L$, S'$_{Cb}$, or S'$_{Cr}$.

This subclause specifies transform coefficient decoding and picture construction prior to the deblocking filter process.

When the prediction unit which current transform unit belongs to is coded as MODE_SKIP, all values of LumaLevel and ChromaLevel are set equal to 0 for the current transform unit.

When the prediction unit which current transform unit belongs to is coded as MODE_INTRA and the value of planar_flag equals to 1 for the corresponding prediction unit, all values of LumaLevel and ChromaLevel are set equal to 0 for the current transform unit.

6.4.1 Specification of transform decoding process for luma transform unit

The variable LumaLevel contains the levels for the luma transform coefficients for the luma block with index tuIdx.

For a luma block indexed by tuIdx, the following ordered steps are specified:

1. The inverse scanning process for the luma transform coefficients as described specified in subclause 6.4.3 is invoked with LumaLevel as the input and the two-dimensional array c as the output.
2. The scaling and transformation process for the residual block as specified in subclause 6.4.5 is invoked with c as the input and r as the output.
3. The position of the upper-left luma sample of the current largest coding unit is derived by invoking the inverse largest coding unit scanning process in subclause 4.2.1 with address lcuAddr as input and the output being assigned to (xL, yL).
4. The position of the upper-left luma sample of the current prediction unit puIdx inside the largest coding unit is derived by invoking the inverse prediction unit scanning process in subclause 4.2.4.1 with puIdx as input and the output being assigned to (xP, yP).
5. The position of the upper-left sample of the transform unit with index tuIdx inside the prediction unit is derived by invoking the inverse transform unit scanning process in subclause 4.2.4.3 with tuIdx as the input and the output being assigned to (xO, yO).
6. The NxN array u with elements u$_{ij}$ for i, j = 0..N-1 is derived in raster order as

   – Let xS be xL + xP + xT and yS be yL + yP + yT.

   – If PredMode is equal to MODE_INTRA and combined_intra_pred_flag is equal to 1,

$openPred_{0,0}$=(3*S'$_L$[xS–1, yS] + 3*S'$_L$[xS , yS–1] + 2*S'$_L$[xS–1, yS–1] + 4) >> 3

if i>0, $openPred_{i,0}$ = (3*S'$_L$[xS–1, yS+i] + 3*$u_{i-1,0}$ + 2*S'$_L$[xS–1, yS+i–1] + 4) >> 3

if j>0, $openPred_{0,j}$ = (3*$u_{0,j-1}$ + 3*S'$_L$[xS+j , yS–1] + 2*S'$_L$[xS+j–1, yS–1] + 4) >> 3

if i>0 and j>0, $openPred_{i,j}$ = (3*$u_{i,j-1}$ + 3*$u_{i-1,j}$ + 2*$u_{i-1,j-1}$ + 4) >> 3

[Ed.: availability check of S'$_L$ is needed.]

u_{ij} = Clip1$_Y$((($pred_L$[xO + j, yO + i] + $openPred_{i,j}$ + 1) >> 1) + r_{ij})

- Otherwise

  u_{ij} = Clip1$_Y$($pred_L$[xO + j, yO + i] + r_{ij})

7. The picture construction process prior to deblocking filter process in subclause 6.4.6 is invoked with u and tuIdx as the inputs.

6.4.2 Specification of transform decoding process for chroma transform unit

This process is invoked for each chroma component Cb and Cr separately when ChromaArrayType is not equal to 0.

For each chroma component, the variable ChromaLevel[iCbCr], with iCbCr set equal to 0 for Cb and iCbCr set equal to 1 for Cr, contain the levels for both components of the chroma transform coefficients.

For each chroma component, the transform decoding proceeds separately in the following ordered steps:

1. The inverse scanning process for chroma transform coefficients as described specified in subclause 6.4.3 is invoked with ChromaLevel[iCbCr] as the input and the two-dimensional array c as the output.
2. The scaling and transformation process for residual block as specified in subclause 6.4.5 is invoked with c as the input and r as the output.
3. The position of the upper-left sample of the transform unit with index tuIdx inside the prediction unit is derived by invoking the inverse transform unit scanning process in subclause 4.2.4.3 with tuIdx as the input and the output being assigned to (xO, yO).

   xO_C = xO / SubWidthC

   yO_C = yO / SubHeightC

4. The NxN array u with elements u_{ij} for i, j = 0..N-1 is derived as

   u_{ij} = Clip1$_C$($pred_C$[xO_C + j, yO_C + i] + r_{ij})

5. The picture construction process prior to deblocking filter process in subclause 6.4.6 is invoked with u and tuIdx as the inputs.

6.4.3 Inverse scanning process for transform coefficients

Input to this process is a list of NxN values listTrCoeffs.

Output of this process is a variable c containing a two-dimensional array of NxN values. In the case of transform coefficients, these NxN values represent levels assigned to locations in the transform unit.

The variable c_{ij} which is located in the (j, i) position in the array c is derived as follows.

c_{ij} = ZigzagToBlockPosition(listTrCoeffs, N, idx).

If adaptive coefficient scanning is used, adaptive coefficient scanning order is determined by the mddtIdx parameter. The scan order associated by mddtIdx is used for inverse mapping from scan order to block position.

The variable c_{ij} which is located in the (j, i) position in the array c is derived as follows.

c_{ij} = AdaptScanOrderToBlockPosition(listTrCoeffs, N, mddtIdx, idx).

6.4.4 Adaptive Coefficient Scanning

[Ed: needs more work]

1. At the beginning of each video slice, initialize the coefficient scanning order for each mddtIdx;
2. When a block is entropy-coded, for each non-zero coefficient coded, increment the count at the corresponding position by one;
3. After each macroblock is coded, update the coefficient scanning order according to the statistics collected;
4. Normalize the collected statistics if needed;
5. Use the updated order for coding of future blocks. Go back to 2 until slice is finished.

If no MDDT is used, e.g. for block size larger than 8x8, and entropySymbolMode != 0, an adaptive scan is used for parsing the significance map. The adaptive scan for significance map is further described in 7.5.1.

6.4.5 Scaling and transformation process for transform unit

Input to this process is an NxN array c with elements cij which is either an array relating to the residual block of the luma component or, when ChromaArrayType is equal to 3, an array relating to the residual block of a chroma component.

NOTE 1 – When separate_colour_plane_flag is equal to 1, all residual blocks are considered to be associated with the luma component for purposes of the decoding process of each coded picture (prior to the final assignment of the decoded picture to a particular luma or chroma picture array according to the value of colour_plane_id).

Outputs of this process are residual sample values as NxN array r with elements r_{ij}.

The variables bitDepth and qP are derived as follows.

- If the input array c relates to a luma residual block, bitDepth is set equal to $BitDepth_Y$ and qP is set equal to QP'_Y.
- Otherwise (the input array c relates to a chroma residual block), bitDepth is set equal to $BitDepth_C$ and qP is set equal to QP'_C.

NOTE 2 – When separate_colour_plane_flag is equal to 1, all residual blocks are considered to be associated with the luma component for purposes of the decoding process of each colour component of a picture.

If PredMode is MODE_INTRA and N is equal to 4 or 8, mode depedent directiona transform process as specified in subclause 6.4.5.1 is invoked for luma residual block array c.

Depending on the values of rotational_transform_idx, the following applies.

- If rotational_transform_idx is equal to 1, the following ordered steps are specified:
  1. The rotational transformation process as specified in subclause 6.4.5.2 is invoked with bitDepth, qP, and c as the inputs and the output is assigned to the NxN array d of scaled transform coefficients with elements d_{ij}.
  2. The scaling process for residual blocks as specified in subclause 6.4.5.3 is invoked with bitDepth, qP, and d as the inputs and the output is assigned to the NxN array e of scaled transform coefficients with elements e_{ij}.
  3. The transformation process for residual blocks as specified in subclause 6.4.5.4 is invoked with bitDepth and e as the inputs and the output is assigned to the NxN array r of residual sample values with elements r_{ij}.
- Otherwise (rotational_transform_idx is equal to 0), the following ordered steps are specified:
  1. The scaling process for residual NxN blocks as specified in subclause 6.4.5.3 is invoked with bitDepth, qP, and c as the inputs and the output is assigned to the NxN array d of scaled transform coefficients with elements d_{ij}.
  2. The transformation process for residual blocks as specified in subclause 6.4.5.4 is invoked with bitDepth and d as the inputs and the output is assigned to the NxN array r of residual sample values with elements r_{ij}.

6.4.5.1 Mode dependent directional transformation process for intra prediction residuals

Inputs of this process are

- the variable bitDepth,
- an NxN array of scaled transform coefficients e with elements e_{ij}.
- IntraPredMode
- Prediction unit size PU Size

Outputs of this process are residual sample values as NxN array r with elements r_{ij}.

The transformation process shall convert the block of scaled transform coefficients to a block of output samples in a manner mathematically equivalent to the following.

The final constructed residual samples values are derived as specified in the following ordered steps:

1. Mode dependent directional transform index is derived from IntraPredMode and PU Size according to Table 6-6.
2. If N is equal to 4 block transformation process described in subclause 6.4.5.1.1 is invoked on 4x4 array of scaled transform coefficients e with elements e_{ij}, else block transformation process described in subclause 6.4.5.1.2 is invoked on 8x8 array of scaled transform coefficients e with elements e_{ij}, resulting in residual matrix g_{ij} (i, j = 0…N-1).
3. The residual sample value r_{ij} is derived by

   $r_{ij} = (g_{ij} + 2^{19}) >> 20$ with i, j = 0..N-1

Table 6-6 Specification of IntraPredMode[puPartIdx] and associated mddtIdx

IntraPredMode [puPartIdx]	mddtIdx	
	PU Size != PU_8x8	PU Size == PU_8x8
0	0	0
1	1	1
2	2	2
3	2	4
4	3	5
5	4	7
6	5	3
7	6	6
8	7	8
9	8	3
10	0	4
11	3	0
12	3	0
13	6	3
14	4	4
15	0	1
16	3	1
17	8	3
18	0	4
19	7	5
20	7	5
21	3	0
22	3	0
23	8	7
24	1	7

25	1	3
26	6	4
27	6	6
28	4	6
29	5	1
30	3	1
31	3	8
32	2	8
33	-	3

6.4.5.1.1 Mode dependent directional transform for 4x4 block input

Inputs of this process are 4x4 array e of scaled transform coefficients e_{ij} and mode dependent directional transform index, mddtIdx.

Outputs of this process are residual sample values a 4x4 block array g with elements g_{ij}.

The 4x4 block inverse transform for a given input X is defined as

$$Y = C' X R$$

where C = kltCol4x4[mddtIdx] and R=kltRow4x4[mddtIdx].
The transform matrices kltRow4x4[i], kltCol4x4[i] (i = 0,1,3,4,5,6,7,8) are defined as:

```
kltRow4x4[0]={{40,  64,  78,  68},{ -82, -60,  31,  70},{  76, -52, -56,  69}, {47, -77,  79, -46}}
kltRow4x4[1]={{30,  55,  74,  84},{ -67, -76,  -4,  78},{ -83,  16,  81, -52},{ -64,  85, -66,  25}}
kltRow4x4[3]={{50,  69,  72,  62},{ -89, -43,  42,  70},{  69, -68, -43,  71},{  36, -71,  87, -50}}
kltRow4x4[4]={{26,  59,  81,  76},{ -63, -80,   6,  77},{  86, -15, -72,  60},{  66, -79,  68, -33}}
kltRow4x4[5]={{22,  52,  82,  81},{ -61, -86,   0,  73},{  88, -13, -72,  58},{  67, -78,  67, -36}}
kltRow4x4[6]={{26,  57,  78,  79},{ -65, -77,  -1,  79},{ -87,  15,  75, -56},{ -63,  83, -68,  29}}
kltRow4x4[7]={{50,  70,  72,  62},{ -87, -45,  45,  69},{  69, -67, -44,  72},{  39, -71,  85, -51}}
kltRow4x4[8]={{29,  59,  77,  78},{ -71, -73,   3,  77},{ -85,  25,  72, -58},{ -57,  84, -72,  30}}

kltCol4x4[0]={{32,  56,  74,  82},{ -69, -74,  -3,  79},{ -81,  17,  82, -53},{ -63,  87, -65,  24}}
kltCol4x4[1]={{29,  55,  80,  78},{ -78, -73,  11,  69},{  81, -35, -67,  64},{  55, -82,  73, -37}}
kltCol4x4[3]={{21,  54,  80,  81},{ -64, -82,  -2,  74},{  88, -15, -71,  57},{  64, -80,  70, -32}}
kltCol4x4[4]={{24,  56,  81,  78},{ -62, -81,   0,  77},{  87, -12, -73,  58},{  66, -81,  67, -32}}
kltCol4x4[5]={{25,  56,  79,  80},{ -64, -79,  -3,  78},{ -86,  11,  76, -56},{ -65,  84, -66,  28}}
kltCol4x4[6]={{10,  32,  73, 100},{ -50, -92, -38,  63},{  89,  15, -80,  44},{  77, -82,  57, -24}}
kltCol4x4[7]={{26,  57,  78,  80},{ -64, -78,  -2,  79},{ -85,  13,  77, -55},{ -65,  83, -67,  28}}
kltCol4x4[8]={{40,  38,  60,  99},{ -97, -62,  15,  54},{  65, -80, -63,  43},{  34, -69,  93, -44}}
```

If mmdtIdx = 2, then the transformation process defined in subclause 6.4.5.4 is invoked.

6.4.5.1.2 Mode dependent directional transform for 8x8 block input

Inputs of this process are 4x4 array e of scaled transform coefficients e_{ij} and mode dependent directional transform index, mddtIdx.

Outputs of this process are residual sample values a 8x8 block array g with elements g_{ij}.

The 4x4 block inverse transform for a given input X is defined as

$$Y = C' X R$$

where C = kltCol8x8[mddtIdx] and R=kltRow8x8[mddtIdx].
The transform matrices kltRow8x8[i], kltCol8x8[i] (i = 0,1,3,4,5,6,7,8) are defined as:

```
kltRow8x8[0]={ { 31,  40,  45,  49,  51,  51,  49,  43},{ -52, -59, -47, -23,   6,  35,  54,  57},
               { -58, -42,  10,  57,  60,  21, -30, -56},{  55,   5, -57, -42,  34,  64,   5, -56},
               { -55,  33,  53, -37, -45,  45,  41, -49},{  47, -59,   4,  52, -50, -11,  61, -40},
```

```
                { -31,  60, -57,  21,  25, -57,  59, -28},{ -15,  36, -53,  63, -63,  52, -35,  14}}
kltRow8x8[1]={  { 18,  28,  37,  45,  51,  55,  57,  56},{  42,  58,  57,  41,  14, -19, -46, -61},
                { -56, -53, -11,  41,  61,  38, -16, -56},{  58,  20, -49, -55,  19,  64,  16, -50},
                { -58,  22,  59, -29, -54,  38,  45, -42},{  53, -56,  -3,  54, -44, -18,  63, -36},
                { -37,  62, -53,  15,  28, -57,  59, -26},{  16, -38,  55, -64,  62, -50,  33, -12}
kltRow8x8[3]={  { 37,  45,  49,  51,  50,  47,  43,  38},{  61,  59,  38,   8, -21, -43, -52, -52},
                { -63, -27,  31,  63,  47,   3, -38, -56},{  55, -15, -62, -18,  52,  54,  -8, -57},
                { -48,  49,  34, -56, -26,  58,  28, -51},{  36, -61,  26,  38, -61,   9,  59, -45},
                { -23,  53, -60,  35,   9, -52,  66, -33},{  11, -31,  48, -60,  64, -57,  42, -17}}
kltRow8x8[4]={  { 10,  21,  33,  45,  54,  60,  59,  53},{  25,  49,  61,  54,  24, -17, -49, -59},
                { -40, -60, -36,  22,  61,  45, -16, -59},{  53,  47, -23, -62,  -2,  62,  20, -53},
                { -61,  -9,  63,  -2, -60,  26,  49, -44},{  64, -38, -29,  60, -31, -27,  60, -34},
                { -51,  63, -41,   1,  37, -57,  54, -24},{  27, -46,  58, -61,  57, -46,  31, -12}}
kltRow8x8[5]={  { 11,  22,  34,  45,  53,  59,  59,  54},{  25,  49,  61,  53,  24, -16, -49, -60},
                { -38, -60, -37,  21,  62,  46, -15, -57},{  54,  48, -25, -62,  -1,  61,  20, -51},
                { -65,  -7,  62,  -7, -57,  29,  47, -44},{  64, -43, -22,  58, -35, -23,  59, -35},
                { -49,  64, -48,   9,  31, -55,  54, -25},{  22, -41,  54, -61,  61, -50,  35, -14}}
kltRow8x8[6]={  { 10,  21,  33,  44,  53,  59,  60,  55},{  27,  50,  59,  53,  26, -13, -46, -63},
                { -45, -60, -33,  20,  58,  47, -10, -60},{  59,  42, -28, -60,  -3,  59,  26, -50},
                { -62,   2,  61,  -7, -61,  22,  53, -42},{  62, -48, -21,  61, -32, -29,  58, -30},
                { -43,  64, -49,   7,  36, -59,  53, -22},{  21, -43,  57, -62,  59, -48,  31, -12}}
kltRow8x8[7]={  { 36,  43,  48,  50,  50,  48,  45,  40},{ -58, -60, -42, -13,  16,  41,  53,  52},
                { -60, -33,  23,  63,  53,   8, -37, -54},{  56,  -7, -61, -27,  47,  57,  -6, -57},
                { -52,  44,  41, -49, -31,  55,  31, -51},{  39, -61,  18,  44, -58,   3,  59, -45},
                { -26,  56, -59,  30,  15, -54,  64, -32},{ -13,  33, -50,  61, -63,  55, -40,  17}}
kltRow8x8[8]={  { 10,  21,  34,  44,  53,  58,  59,  55},{  33,  54,  63,  50,  19, -17, -47, -56},
                { -54, -59, -21,  34,  61,  34, -20, -56},{  63,  28, -41, -51,  17,  62,  15, -54},
                { -66,  20,  57, -29, -50,  37,  42, -44},{  52, -58,   3,  54, -47, -14,  60, -37},
                { -33,  59, -53,  19,  27, -59,  60, -28},{  17, -38,  55, -64,  62, -51,  32, -11}}

kltCol8x8[0]={  { 17,  28,  37,  45,  51,  55,  57,  56},{  36,  54,  58,  45,  17, -18, -48, -62},
                { -49, -54, -22,  32,  64,  44, -14, -58},{  63,  29, -40, -57,   7,  60,  24, -50},
                { -62,  15,  58, -17, -55,  29,  52, -45},{  53, -54, -12,  61, -38, -27,  61, -32},
                { -36,  62, -50,   8,  38, -62,  54, -21},{  20, -44,  59, -63,  59, -47,  29, -10}}
kltCol8x8[1]={  { 27,  36,  41,  46,  49,  51,  53,  52},{ -50, -57, -47, -30,  -8,  20,  50,  66},
                { -56, -48,  -3,  47,  64,  36, -17, -55},{  50,  11, -51, -54,  14,  67,  27, -53},
                { -59,  24,  61, -21, -55,  27,  51, -43},{  48, -53, -10,  60, -38, -30,  64, -33},
                { -36,  64, -50,   7,  40, -62,  51, -20},{ -20,  43, -58,  64, -60,  46, -28,  10}}
kltCol8x8[3]={  {  9,  21,  34,  45,  54,  59,  58,  53},{  24,  49,  63,  54,  22, -18, -48, -58},
                { -42, -62, -34,  27,  63,  38, -19, -56},{  58,  44, -30, -56,   9,  61,  17, -55},
                { -66,  -1,  59, -14, -54,  32,  47, -47},{  60, -44, -18,  59, -37, -24,  62, -35},
                { -46,  62, -46,   7,  35, -60,  55, -23},{  24, -44,  57, -62,  59, -48,  31, -11}}
kltCol8x8[4]={  {  9,  20,  33,  45,  55,  60,  59,  51},{  23,  48,  62,  54,  24, -17, -50, -60},
                { -41, -62, -37,  21,  60,  42, -17, -58},{  56,  46, -27, -59,   2,  59,  20, -54},
                { -64,  -5,  61,  -9, -57,  28,  48, -45},{  61, -40, -24,  61, -34, -27,  61, -35},
                { -49,  63, -42,   3,  37, -59,  53, -23},{  27, -46,  57, -61,  57, -47,  31, -12}}
kltCol8x8[5]={  {  9,  19,  30,  42,  52,  59,  61,  57},{  24,  46,  60,  56,  31,  -8, -45, -63},
                { -45, -61, -39,  14,  58,  48,  -8, -57},{  61,  43, -26, -58,  -6,  58,  28, -51},
                { -65,   1,  59,  -5, -59,  19,  55, -43},{  58, -46, -24,  63, -28, -34,  60, -30},
                { -42,  64, -44,  -2,  44, -62,  50, -19},{  25, -48,  61, -63,  56, -43,  26,  -9}}
kltCol8x8[6]={  { 10,  21,  33,  44,  53,  59,  60,  54},{  23,  47,  60,  54,  27, -12, -48, -63},
                { -39, -60, -39,  17,  59,  48,  -9, -59},{  56,  49, -24, -62,  -5,  58,  25, -50},
                { -64,  -6,  63,  -7, -58,  23,  51, -42},{  63, -42, -23,  60, -32, -29,  60, -33},
                { -49,  64, -46,   6,  35, -58,  52, -22},{  24, -42,  55, -62,  60, -49,  31, -12}}
kltCol8x8[7]={  { 10,  22,  34,  44,  53,  58,  59,  56},{  28,  52,  62,  51,  22, -15, -47, -60},
                { -47, -59, -29,  27,  62,  41, -14, -58},{  62,  36, -34, -55,   5,  60,  23, -53},
                { -67,  10,  58, -17, -54,  27,  51, -44},{  55, -50, -13,  62, -39, -26,  60, -33},
                { -38,  62, -50,   8,  37, -62,  53, -21},{  23, -44,  58, -62,  59, -48,  30, -11}}
kltCol8x8[8]={  { 32,  40,  44,  45,  44,  47,  53,  53},{ -69, -67, -39,  -9,  13,  30,  45,  50},
                { -60, -21,  42,  68,  47,   6, -34, -51},{  53, -21, -66,  -5,  62,  51,  -8, -47},
                { -45,  48,  24, -61, -13,  65,  27, -51},{  39, -63,  31,  29, -62,  19,  54, -46},
                { -24,  52, -63,  49,  -9, -39,  63, -34},{  -8,  22, -36,  51, -64,  66, -53,  23}}
```

If mmdtIdx = 2, then the transformation process defined in subclause 6.4.5.4 is invoked.

6.4.5.2 Rotational transformation process

Inputs of this process are

- the variable bitDepth and qP,
- an NxN array of transform coefficients c with elements c_{ij}

Outputs of this process are transform coefficients as NxN array d with elements d_{ij}.

The transform process shall convert the block of scaled transform coefficients to a block of output samples in a manner mathematically equivalent to the following.

Depending on the values of N, the following applies.

- If N is equal to 4, the following ordered steps are specified:
  1. The scaled transform coefficients s with elements s_{ij} is derived as

     $s_{ij} = c_{ij} * (1<< (qP / 6))$ with i, j = 0…3.
  2. The rotational transformation process for 4x4 transform unit as specified in subclause 6.4.5.2.1 is invoked with bitDepth, qP, and s as the inputs and the output is assigned to the 4x4 array d of transform coefficients with elements d_{ij}.
- Otherwise (N is larget than 4), the following ordered steps are specified:
  1. The scaled transform coefficients s with element s_{ij} is derived as

     $s_{ij} = c_{ij} * (1<< (qP / 6))$ with i, j = 0…7.
  2. The rotational transformation process for 8x8 transform unit as specified in subclause 6.4.5.2.2 is invoked with bitDepth and s_{ij} as the inputs and the output is assigned to the 8x8 array d8x8 of transform coefficients with elements $d8x8_{ij}$.
  3. The NxN array d of transform coefficients with element d_{ij} is derived by

     $d_{ij} = d8x8_{ij}$ with i, j = 0..7,

     $d_{ij} = s_{ij}$ otherwise

6.4.5.2.1 Rotational transformation process for 4x4 transform coefficients

Inputs of this process is a 4x4 array of transform coefficients s with elements s_{ij}

Outputs of this process are transform coefficients as a 4x4 array d with elements d_{ij}.

The variable addShift and addOffset is derived as follows.

shift = Min(Max(0, 4 – increased_bit_depth_luma), 8)

if (shift > 0) offset = 1 << (shift – 1), otherwise offset = 0

When the value of variable idx is set to rotational_transform_idx – 1, output element d_{ij} is specified as follows.

- For j=0..3, a set of intermediate values t_{ij} is derived by

  $t_{ij} = INV_ROT_4[idx][9]*s_{ij} + INV_ROT_4[idx][12]*s_{i+1,j} + INV_ROT_4[idx][15]*s_{i+2,j}$

  $t_{i+1,j} = INV_ROT_4[idx][10]*s_{ij}+ INV_ROT_ 4[idx][13]*s_{i+1,j}+ INV_ROT_4[idx][16]*s_{i+2,j}$

  $t_{i+2,j} = INV_ROT_ 4[idx][11]*s_{ij}+ INV_ROT_4[idx][14]*s_{i+1,j}+ INV_ROT_4[idx][17]*s_{i+2,j}$

  $t_{ij} = Sign(t_{ij})*(Abs(t_{ij}) >> (8-Shift))$

  $t_{i+1,j} = Sign(t_{i+1,j})*(Abs(t_{i+1,j}) >> (8-shift))$

  $t_{i+2,j} = Sign(t_{i+2,j})*(Abs(t_{i+2,j}) >> (8-shift))$

  $t_{i+3,j} = t_{i+3,j} << shift$
- For i=0..3, the transformed result d_{ij} is computed from these intermediate values t_{ij} as

  $d_{ij} = INV_ROT_4[idx][0]*t_{ij} + INV_ROT_4[idx][3]*t_{i,j+1}+ INV_ROT_4[idx][6]*t_{i,j+2}$

  $d_{i,j+1} = INV_ROT_4[idx][1]*t_{ij} + INV_ROT_4[idx][4]*t_{i,j+1}+ INV_ROT_4[idx][7]*t_{i,j+2}$

  $d_{i,j+2} = INV_ROT_4[idx][2]*t_{ij} + INV_ROT_4[idx][5]*t_{i,j+1} + INV_ROT_4[idx][8]*t_{i,j+2}$

  $d_{i,j+3} = t_{i, j+3}$

  $d_{ij} = Sign(d_{ij})*((Abs(d_{ij}) + offset) >> (8+shift))$

$d_{i,j+1}$ = Sign($d_{i,j+1}$)*((Abs($d_{i,j+1}$) + (offset << 8)) >> (8+shift))

$d_{i,j+2}$ = Sign($d_{i,j+2}$)*((Abs($d_{i,j+2}$) + (offset << 8)) >> (8+shift))

$d_{i,j+3}$ = Sign($d_{i,j+3}$)*((Abs($d_{i,j+3}$) + offset) >> shift)

The matrix INV_ROT_4 is specified in Table 6-7.

Table 6-7 Specification of transpose of NV_ROT_4 matrix

	idx = 0	idx = 1	idx = 2	idx = 3
0	248	255	237	250
1	-55	22	-96	-18
2	-28	-8	-9	-54
3	-60	-22	90	-13
4	-244	254	229	220
5	-52	-17	-73	-131
6	-16	6	35	55
7	57	17	64	130
8	-249	255	245	213
9	250	241	243	212
10	-49	-84	-80	-143
11	-26	-18	-6	9
12	-54	69	70	134
13	-240	221	222	192
14	-71	-110	-107	-103
15	-11	52	38	51
16	75	98	100	90
17	-245	231	233	234

6.4.5.2.2 Rotational transformation process for 8x8 transform coefficients

Inputs of this process is a 8x8 array of transform coefficients s with elements s_{ij}

Outputs of this process are transform coefficients as a 8x8 array d with elements d_{ij}.

The variable addShift and addOffset is derived as follows.

shift = Min(Max(0, 4 – increased_bit_depth_luma), 8)

if (shift > 0) offset = 1 << (shift – 1), otherwise offset = 0

When the value of variable idx is set to rotational_transform_idx – 1, output element d_{ij} is specified as follows.

- For j=0..7, a set of intermediate values t_{ij} is derived by

t_{ij} = INV_ROT_8[idx][9]*s_{ij} + INV_ROT_8[idx][12]*$s_{i+1,j}$+ INV_ROT_8[idx][15]*$s_{i+2,j}$

$t_{i+1,j}$ = INV_ROT_8[idx][10]*s_{ij} + INV_ROT_8[idx][13]*$s_{i+1,j}$+ INV_ROT_8[idx][16]* $s_{i+2,j}$

$t_{i+2,j}$ = INV_ROT_8[idx][11]*s_{ij} + INV_ROT_8[idx][14]*$s_{i+1,j}$+ INV_ROT_8[idx][17]* $s_{i+2,j}$

$t_{i+3,j}$ = INV_ROT_8[idx][27]*$s_{i+3,j}$ + INV_ROT_8[idx][30]*$s_{i+4,j}$ + INV_ROT_8[idx][33]*$s_{i+5,j}$

$t_{i+4,j}$ = INV_ROT_8[idx][28]*$s_{i+3,j}$ + INV_ROT_8[idx][31]*$s_{i+4,j}$ + INV_ROT_8[idx][34]*$s_{i+5,j}$

$t_{i+5,j}$ = INV_ROT_8[idx][29]*$s_{i+3,j}$ + INV_ROT_8[idx][32]*$s_{i+4,j}$ + INV_ROT_8[idx][35]*$s_{i+5,j}$

t_{ij} = Sign(t_{ij})*(Abs(t_{ij}) >> (8-shift))

$t_{i+1,j}$ = Sign($t_{i+1,j}$)*(Abs($t_{i+1,j}$) >> (8-shift))

$t_{i+2,j}$ = Sign($t_{i+2,j}$)*(Abs($t_{i+2,j}$) >> (8-shift))

$t_{i+3,j}$ = Sign($t_{i+3,j}$)*(Abs($t_{i+3,j}$) >> (8-shift))

$t_{i+4,j}$ = Sign($t_{i+4,j}$)*(Abs($t_{i+4,j}$) >> (8-shift))

$t_{i+5,j}$ = Sign($t_{i+5,j}$)*(Abs($t_{i+5,j}$) >> (8-shift))

$t_{i+6,j}$ = $s_{i+6,j}$ << shift

$t_{i+7,j}$ = $s_{i+7,j}$ << shift

- For i=0..7, the transformed result d_{ij} is computed from these intermediate values t_{ij} as

d_{ij} = INV_ROT_8[idx][0] * t_{ij} + INV_ROT_8[index][3] * $t_{i,j+1}$ + INV_ROT_8[idx][6] * $t_{i,j+2}$

$d_{i,j+1}$ = INV_ROT_8[idx][1] * t_{ij} + INV_ROT_8[index][4] * $t_{i,j+1}$ + INV_ROT_8[idx][7] * $t_{i,j+2}$

$d_{i,j+2}$ = INV_ROT_8[idx][2] * t_{ij} + INV_ROT_8[index][5] * $t_{i,j+1}$ + INV_ROT_8[idx][8] * $t_{i,j+2}$

$d_{i,j+3}$ = INV_ROT_8[idx][18] * $t_{i,j+3}$ + INV_ROT_8[idx][21] * $t_{i,j+4}$ + INV_ROT_8[idx][24] * $t_{i,j+5}$

$d_{i,j+4}$ = INV_ROT_8[idx][19] * $t_{i,j+3}$ + INV_ROT_8[idx][22] * $t_{i,j+4}$ + INV_ROT_8[idx][25] * $t_{i,j+5}$

$d_{i,j+5}$ = INV_ROT_8[idx][20] * $t_{i,j+3}$ + INV_ROT_8[idx][23] * $t_{i,j+4}$ + INV_ROT_8[idx][26] * $t_{i,j+5}$

$d_{i,j+6}$ = $t_{i,j+6}$

$d_{i,j+7}$ = $t_{i,j+7}$

d_{ij} = Sign(d_{ij})*((Abs(d_{ij}) + (offset << 8)) >> (8+shift))

$d_{i,j+1}$ = Sign($d_{i,j+1}$)*((Abs($d_{i,j+1}$) + (offset << 8)) >> (8+shift))

$d_{i,j+2}$ = Sign($d_{i,j+2}$)*((Abs($d_{i,j+2}$) + (offset << 8)) >> (8+shift))

$d_{i,j+3}$ = Sign($d_{i,j+3}$)*((Abs($d_{i,j+3}$) + (offset << 8)) >> (8+shift))

$d_{i,j+4}$ = Sign($d_{i,j+4}$)*((Abs($d_{i,j+4}$) + (offset << 8)) >> (8+shift))

$d_{i,j+5}$ = Sign($d_{i,j+5}$)*((Abs($d_{i,j+5}$) + (offset << 8)) >> (8+shift))

$d_{i,j+6}$ = Sign($d_{i,j+6}$)*((Abs($d_{i,j+6}$) + offset) >> shift)

$d_{i,j+7}$ = Sign($d_{i,j+7}$)*((Abs($d_{i,j+7}$) + offset) >> shift)

The matrix INV_ROT_8 is specified in Table 6-8.

Table 6-8 Specification of transpose of INV_ROT_8 matrix

	idx = 0	idx = 1	idx = 2	idx = 3
0	251	243	243	174
1	-36	-61	-81	-186
2	-31	-50	12	23
3	23	-75	81	181
4	237	-230	232	158
5	-94	-83	-70	-87
6	42	-25	11	49
7	90	94	71	75
8	236	-237	246	240
9	236	254	252	236
10	-100	-27	-41	-99
11	3	-18	-9	-10
12	100	22	-42	-99
13	234	249	-251	-235
14	-25	-56	-28	-25
15	7	24	-4	1
16	24	54	29	27
17	255	249	-254	-255
18	251	243	243	174
19	-36	-61	-81	-186
20	-31	-50	12	23
21	23	-75	81	181
22	237	-230	232	158
23	-94	-83	-70	-87
24	42	-25	11	49

25	90	94	71	75
26	236	-237	246	240
27	236	254	252	236
28	-100	-27	-41	-99
29	3	-18	-9	-10
30	100	22	-42	-99
31	234	249	-251	-235
32	-25	-56	-28	-25
33	7	24	-4	1
34	24	54	29	27
35	255	249	-254	-255

6.4.5.3 Scaling process for residual blocks

Inputs of this process are

- the variables bitDepth and qP,
- an NxN array d with elements d_{ij} which is either an array relating to a residual block of luma component or an array relating to a residual block of a chroma component.

Output of this process is an NxN array of scaled transform coefficients e with elements e_{ij}.

The scaling process for NxN block transform coefficient levels d_{ij} proceeds as follows.

- If rotational_transform_idx is not equal to 0, the scaled result is derived as

  e_{ij} = (d_{ij} * LevelScaleNxN[qP % 6][i][j]) , with i, j = 0..N-1

  where N in the LevelScaleNxN is replaced with 4, 8, 16, 32 and 64 depending on the value of N.

- Otherwise (rotational_transform_idx is equal to 0), the scaled result is derived as

  e_{ij} = (d_{ij} * LevelScaleNxN[qP % 6][i][j]) << (qP / 6), with i, j = 0..N-1

  where N in the LevelScaleNxN is replaced with 4, 8, 16, 32 and 64 depending on the value of N.

If mode dependent directional transform is invoked, LevelScaleNxN[m][i][j] = w[m] for m=0..5 and i, j = 0..N-1 where w = [40 45 50 57 63 71].

6.4.5.4 Transformation process for residual blocks

Inputs of this process are

- the variable bitDepth,
- an NxN array of scaled transform coefficients e with elements e_{ij}.

Outputs of this process are residual sample values as NxN array r with elements r_{ij}.

The transformation process shall convert the block of scaled transform coefficients to a block of output samples in a manner mathematically equivalent to the following.

The final constructed residual samples values are derived as specified in the following ordered steps:

4. Each (horizontal) row of scaled transform coefficients e_{ij} (i, j = 0…N-1) is transformed to f_{ij} (i, j = 0…N-1) by invoking the one-dimensional transformation process as specified in subclause 6.4.5.4.1.
5. Each (vertical) column of the resulting matrix f_{ij} (i, j = 0…N-1) is transformed by invoking the one-dimensional transformation process as specified in subclause 6.4.5.4.1 where resulting matrix is g_{ij} (i, j = 0…N-1).
6. Depending on the value of N, following is applied.
   - If N is equal to 4 or 8 or 64, the residual sample value r_{ij} is derived by

     $r_{ij} = (g_{ij} + 2^8) >> 9$ with i, j = 0..N-1

- Otherwise (N is equal to 16 or 32), the residual sample value r_{ij} is derived by

$$r_{ij} = (g_{ij} + 2^9) >> 10 \qquad \text{with } i, j = 0..N-1$$

6.4.5.4.1 Transformation process for N-point input

Inputs of this process are an N-point array of scaled transform coefficients X with elements Xi.

Outputs of this process are residual sample values as N-point array Y with elements Yi.

Depending on the values of N, the following applies.

- If N is equal to 4, the transformation process for 4-point input as specified in subclause 6.4.5.4.2 is invoked with X as the input and Y as the output.
- Otherwise, if N is equal to 8, the transformation process for 8-point input as specified in subclause 6.4.5.4.3 is invoked with X as the input and Y as the output.
- Otherwise, if N is equal to 16, the transformation process for 16-point input as specified in subclause 6.4.5.4.4 is invoked with X as the input and Y as the output.
- Otherwise, if N is equal to 32, the transformation process for 32-point input as specified in subclause 6.4.5.4.5 is invoked with X as the input and Y as the output.
- Otherwise, if N is equal to 64, the transformation process for 64-point input as specified in subclause 6.4.5.4.6 is invoked with X as the input and Y as the output.

6.4.5.4.2 Transformation process for 4-point input

[Ed.: same as H.264/AVC]

6.4.5.4.3 Transformation process for 8-point input

[Ed.: same as H.264/AVC]

6.4.5.4.4 Transformation process for 16-point input

Inputs of this process are a 16-point array of scaled transform coefficients X with elements Xi.

Outputs of this process are residual sample values as a 16-point array Y with elements Yi.

16-point one-dimensional inverse transform process is defined as following 6 stages. Xi (i = 0...15) are elements of input vector and Yi (i=0...15) are elements of output vector. In each stage, sets of intermediate values Fi, Ei, Di, Ci, Bi (i = 0...15) are derived and fed into the next stage.

Stage 1	Stage 2	Stage 3
F8 = (6 * X1 - 63 * X15) >> 6	E4 = (12 * X2 - 62 * X14) >> 6	D0 = (45 * (X0 + X8)) >> 6
F9 = (49 * X9 - 40 * X7) >> 6	E5 = (53 * X10 - 35 * X6) >> 6	D1 = (45 * (X0 - X8)) >> 6
F10 = (30 * X5 - 56 * X11) >> 6	E6 = (53 * X6 + 35 * X10) >> 6	D2 = (24 * X4 - 59 * X12) >> 6
F11 = (61 * X13 - 18 * X3) >> 6	E7 = (12 * X14 + 62 * X2) >> 6	D3 = (59 * X4 + 24 * X12) >> 6
F12 = (61 * X3 + 18 * X13) >> 6	E8 = F8 + F9	D4 = E4 + E5
F13 = (30 * X11 + 56 * X5) >> 6	E9 = F8 - F9	D5 = E4 - E5
F14 = (49 * X7 + 40 * X9) >> 6	E10 = F11 - F10	D6 = E7 - E6
F15 = (6 * X15 + 63 * X1) >> 6	E11 = F11 + F10	D7 = E7 + E6
	E12 = F12 + F13	D9 = (24 * E14 - 59 * E9) >> 6
	E13 = F12 - F13	D10 = (- 59 * E13 - 24 * E10) >> 6
	E14 = F15 - F14	D13 = (24 * E13 - 59 * E10) >> 6
	E15 = F15 + F14	D14 = (59 * E14 + 24 * E9) >> 6

Stage 4	Stage 5	Stage 6
C0 = D0 + D3	B0 = C0 + D7	Y0 = B0 + C15
C3 = D0 - D3,	B7 = C0 - D7	Y15 = B0 - C15
C8 = E8 + E11	B1 = C1 + C6	Y1 = B1 + C14
C11 = E8 - E11	B6 = C1 - C6	Y14 = B1 - C14
C12 = E15 - E12	B2 = C2 + C5	Y2 = B2 + B13
C15 = E15 + E12	B5 = C2 - C5	Y13 = B2 - B13
C1 = D1 + D2	B3 = C3 + D4	Y3 = B3 + B12
C2 = D1 - D2	B4 = C3 - D4	Y12 = B3 - B12

C9 = D9 + D10 C10 = D9 - D10 C13 = D14 - D13 C14 = D14 + D13 C5 = (45 * (D6 - D5)) >> 6 C6 = (45 * (D6 + D5)) >> 6	B10 = (45 * (C13 - C10)) >> 6 B13 = (45 * (C13 + C10)) >> 6 B11 = (45 * (C12 - C11)) >> 6 B12 = (45 * (C12 + C11)) >> 6	Y4 = B4 + B11 Y11 = B4 - B11 Y5 = B5 + B10 Y10 = B5 - B10 Y6 = B6 + C9 Y9 = B6 - C9 Y7 = B7 + C8 Y8 = B7 - C8

6.4.5.4.5 Transformation process for 32-point input

Inputs of this process are a 32-point array of scaled transform coefficients X with elements Xi.

Outputs of this process are residual sample values as a 32-point array Y with elements Yi.

32-point one-dimensional inverse transform process is defined as following 8 stages. Xi (i = 0...31) are elements of input vector and Yi (i=0...31) are elements of output vector. In each stage, sets of intermediate values Gi, Fi, Ei, Di, Ci, Bi, Ai (i = 0...31) are derived and fed into next stage.

Stage 1	Stage 2	Stage 3
G16 = (12 * X1 - 255 * X31) >> 8 G17 = (189 * X17 - 171 * X15) >> 8 G18 = (109 * X9 - 231 * X23) >> 8 G19 = (241 * X25 - 86 * X7) >> 8 G20 = (62 * X5 - 248 * X27) >> 8 G21 = (219 * X21 - 131 * X11) >> 8 G22 = (152 * X13 - 205 * X19) >> 8 G23 = (253 * X29 - 37 * X3) >> 8 G24 = (253 * X3 + 37 * X29) >> 8 G25 = (152 * X19 + 205 * X13) >> 8 G26 = (219 * X11 + 131 * X21) >> 8 G27 = (62 * X27 + 248 * X5) >> 8 G28 = (241 * X7 + 86 * X25) >> 8 G29 = (109 * X23 + 231 * X9) >> 8 G30 = (189 * X15 + 171 * X17) >> 8 G31 = (12 * X31 + 255 * X1) >> 8	F8 = (25 * X2 - 254 * X30) >> 8 F9 = (197 * X18 - 162 * X14) >> 8 F10 = (120 * X10 - 225 * X22) >> 8 F11 = (244 * X26 - 74 * X6) >> 8 F12 = (244 * X6 + 74 * X26) >> 8 F13 = (120 * X22 + 225 * X10) >> 8 F14 = (197 * X14 + 162 * X18) >> 8 F15 = (25 * X30 + 254 * X2) >> 8 F16 = G16 + G17 F17 = G16 - G17 F18 = G19 - G18 F19 = G19 + G18 F20 = G20 + G21 F21 = G20 - G21 F22 = G23 - G22 F23 = G23 + G22 F24 = G24 + G25 F25 = G24 - G25 F26 = G27 - G26 F27 = G27 + G26 F28 = G28 + G29 F29 = G28 - G29 F30 = G31 - G30 F31 = G31 + G30	E4 = (49 * X4 - 251 * X28) >> 8 E5 = (212 * X20 - 142 * X12) >> 8 E6 = (212 * X12 + 142 * X20) >> 8 E7 = (49 * X28 + 251 * X4) >> 8 E8 = F8 + F9 E9 = F8 - F9 E10 = F11 - F10 E11 = F11 + F10 E12 = F12 + F13 E13 = F12 - F13 E14 = F15 - F14 E15 = F15 + F14 E17 = (49 * F30 - 251 * F17) >> 8 E18 = (- 251 * F29 - 49 * F18) >> 8 E21 = (212 * F26 - 142 * F21) >> 8 E22 = (- 142 * F25 - 212 * F22) >> 8 E25 = (212 * F25 - 142 * F22) >> 8 E26 = (142 * F26 + 212 * F21) >> 8 E29 = (49 * F29 - 251 * F18) >> 8 E30 = (251 * F30 + 49 * F17) >> 8

Stage 4	Stage 5	Stage 6
D0 = (181 * (X0 + X16)) >> 8 D1 = (181 * (X0 - X16)) >> 8 D2 = (97 * X8 - 236 * X24) >> 8 D3 = (236 * X8 + 97 * X24) >> 8 D4 = E4 + E5 D5 = E4 - E5 D6 = E7 - E6 D7 = E7 + E6 D9 = (97 * E14 - 236 * E9) >> 8 D10 = (- 236 * E13 - 97 * E10) >> 8 D13 = (97 * E13 - 236 * E10) >> 8 D14 = (236 * E14 + 97 * E9) >> 8 D16 = F16 + F19 D19 = F16 - F19 D20 = F23 - F20 D23 = F23 + F20 D24 = F24 + F27 D27 = F24 - F27 D28 = F31 - F28 D31 = F31 + F28 D17 = E17 + E18 D18 = E17 - E18 D21 = E22 - E21 D22 = E22 + E21 D25 = E25 + E26 D26 = E25 - E26 D29 = E30 - E29	C0 = D0 + D3 C3 = D0 - D3 C8 = E8 + E11 C11 = E8 - E11 C12 = E15 - E12 C15 = E15 + E12 C1 = D1 + D2 C2 = D1 - D2 C9 = D9 + D10 C10 = D9 - D10 C13 = D14 - D13 C14 = D14 + D13 C5 = (181 * (D6 - D5)) >> 8 C6 = (181 * (D6 + D5)) >> 8 C18 = (97 * D29 - 236 * D18) >> 8 C20 = (- 236 * D27 - 97 * D20) >> 8 C26 = (- 236 * D21 + 97 * D26) >> 8 C28 = (97 * D19 + 236 * D28) >> 8 C19 = (97 * D28 - 236 * D19) >> 8 C21 = (- 236 * D26 - 97 * D21) >> 8 C27 = (- 236 * D20 + 97 * D27) >> 8 C29 = (97 * D18 + 236 * D29) >> 8	B7 = C0 - D7 B1 = C1 + C6 B6 = C1 - C6 B2 = C2 + C5 B5 = C2 - C5 B3 = C3 + D4 B4 = C3 - D4 B10 = (181 * (C13 - C10)) >> 8 B13 = (181 * (C13 + C10)) >> 8 B11 = (181 * (C12 - C11)) >> 8 B12 = (181 * (C12 + C11)) >> 8 B16 = D16 + D23 B23 = D16 - D23 B24 = D31 - D24 B31 = D31 + D24 B17 = D17 + D22 B22 = D17 - D22 B25 = D30 - D25 B30 = D30 + D25 B18 = C18 + C21 B21 = C18 - C21 B26 = C29 - C26 B29 = C29 + C26 B19 = C19 + C20 B20 = C19 - C20 B27 = C28 - C27 B28 = C28 + C27

D30 = E30 + E29		

Stage 7	Stage 8	
A0 = B0 + C15 A15 = B0 - C15 A1 = B1 + C14 A14 = B1 - C14 A2 = B2 + B13 A13 = B2 - B13 A3 = B3 + B12 A12 = B3 - B12 A4 = B4 + B11 A11 = B4 - B11 A5 = B5 + B10 A10 = B5 - B10 A6 = B6 + C9 A9 = B6 - C9 A7 = B7 + C8 A8 = B7 - C8 A20 = (181 * (B27 - B20)) >> 8 A27 = (181 * (B27 + B20)) >> 8 A21 = (181 * (B26 - B21)) >> 8 A26 = (181 * (B26 + B21)) >> 8 A22 = (181 * (B25 - B22)) >> 8 A25 = (181 * (B25 + B22)) >> 8 A23 = (181 * (B24 - B23)) >> 8 A24 = (181 * (B24 + B23)) >> 8	Y0 = (A0 + B31) Y31 = (A0 - B31) Y1 = (A1 + B30) Y30 = (A1 - B30) Y2 = (A2 + B29) Y29 = (A2 - B29) Y3 = (A3 + B28) Y28 = (A3 - B28) Y4 = (A4 + A27) Y27 = (A4 - A27) Y5 = (A5 + A26) Y26 = (A5 - A26) Y6 = (A6 + A25) Y25 = (A6 - A25) Y7 = (A7 + A24) Y24 = (A7 - A24) Y8 = (A8 + A23) Y23 = (A8 - A23) Y9 = (A9 + A22) Y22 = (A9 - A22) Y10 = (A10 + A21) Y21 = (A10 - A21) Y11 = (A11 + A20) Y20 = (A11 - A20) Y12 = (A12 + B19) Y19 = (A12 - B19) Y13 = (A13 + B18) Y18 = (A13 - B18) Y14 = (A14 + B17) Y17 = (A14 - B17) Y15 = (A15 + B16) Y16 = (A15 - B16)	

6.4.5.4.6 Transformation process for 64-point input

Inputs of this process are a 64-point array of scaled transform coefficients X with elements Xi.

Outputs of this process are residual sample values as a 64-point array Y with elements Yi.

64-point one-dimensional inverse transform process is defined as following 10 stages. Xi (i = 0...63) are elements of input vector and Yi (i=0...63) are elements of output vector. In each stage, sets of intermediate values Hi,Gi, Fi, Ei, Di, Ci, Bi, Ai, Oi (i = 0...63) are derived and fed into next stage.

Stage 1	Stage 2	Stage 3
H32 = (25*X1-1024*X63) >> 10 H33 = (742*X33-706*X31) >> 10 H34 = (415*X17-936*X47) >> 10 H35 = (955*X49-369*X15) >> 10 H36 = (224*X9-999*X55) >> 10 H37 = (865*X41-548*X23) >> 10 H38 = (590*X25-837*X39) >> 10 H39 = (1009*X57-175*X7) >> 10 H40 = (125*X5-1016*X59) >> 10 H41 = (807*X37-630*X27) >> 10 H42 = (505*X21-891*X43) >> 10 H43 = (987*X53-273*X11) >> 10 H44 = (321*X13-972*X51) >> 10 H45 = (915*X45-460*X19) >> 10 H46 = (669*X29-775*X35) >> 10 H47 = (1021*X61-75*X3) >> 10 H48 = (1021*X3+75*X61) >> 10 H49 = (669*X35+775*X29) >> 10 H50 = (915*X19+460*X45) >> 10 H51 = (321*X51+972*X13) >> 10 H52 = (987*X11+273*X53) >> 10 H53 = (505*X43+891*X21) >> 10 H54 = (807*X27+630*X37) >> 10 H55 = (125*X59+1016*X5) >> 10 H56 = (1009*X7+175*X57) >> 10 H57 = (590*X39+837*X25) >> 10	G16 = (50*X2-1023*X62) >> 10 G17 = (759*X34-688*X30) >> 10 G18 = (438*X18-926*X46) >> 10 G19 = (964*X50-345*X14) >> 10 G20 = (249*X10-993*X54) >> 10 G21 = (878*X42-526*X22) >> 10 G22 = (610*X26-822*X38) >> 10 G23 = (1013*X58-150*X6) >> 10 G24 = (1013*X6+150*X58) >> 10 G25 = (610*X38+822*X26) >> 10 G26 = (878*X22+526*X42) >> 10 G27 = (249*X54+993*X10) >> 10 G28 = (964*X14+345*X50) >> 10 G29 = (438*X46+926*X18) >> 10 G30 = (759*X30+688*X34) >> 10 G31 = (50*X62+1023*X2) >> 10 G32 = H32+H33 G33 = H32-H33 G34 = H35-H34 G35 = H35+H34 G36 = H36+H37 G37 = H36-H37 G38 = H39-H38 G39 = H39+H38 G40 = H40+H41 G41 = H40-H41	F8 = (100*X4-1019*X60) >> 10 F9 = (792*X36-650*X28) >> 10 F10 = (483*X20-903*X44) >> 10 F11 = (980*X52-297*X12) >> 10 F12 = (980*X12+297*X52) >> 10 F13 = (483*X44+903*X20) >> 10 F14 = (792*X28+650*X36) >> 10 F15 = (100*X60+1019*X4) >> 10 F16 = G16+G17 F17 = G16-G17 F18 = G19-G18 F19 = G19+G18 F20 = G20+G21 F21 = G20-G21 F22 = G23-G22 F23 = G23+G22 F24 = G24+G25 F25 = G24-G25 F26 = G27-G26 F27 = G27+G26 F28 = G28+G29 F29 = G28-G29 F30 = G31-G30 F31 = G31+G30 F33 = (100*G62-1019*G33) >> 10 F34 = (-1019*G61-100*G34) >> 10

H58 = (865*X23+548*X41) >> 10 H59 = (224*X55+999*X9) >> 10 H60 = (955*X15+369*X49) >> 10 H61 = (415*X47+936*X17) >> 10 H62 = (742*X31+706*X33) >> 10 H63 = (25*X63+1024*X1) >> 10	G42 = H43-H42 G43 = H43+H42 G44 = H44+H45 G45 = H44-H45 G46 = H47-H46 G47 = H47+H46 G48 = H48+H49 G49 = H48-H49 G50 = H51-H50 G51 = H51+H50 G52 = H52+H53 G53 = H52-H53 G54 = H55-H54 G55 = H55+H54 G56 = H56+H57 G57 = H56-H57 G58 = H59-H58 G59 = H59+H58 G60 = H60+H61 G61 = H60-H61 G62 = H63-H62 G63 = H63+H62	F37 = (792*G58-650*G37) >> 10 F38 = (-650*G57-792*G38) >> 10 F41 = (483*G54-903*G41) >> 10 F42 = (-903*G53-483*G42) >> 10 F45 = (980*G50-297*G45) >> 10 F46 = (-297*G49-980*G46) >> 10 F49 = (980*G49-297*G46) >> 10 F50 = (297*G50+980*G45) >> 10 F53 = (483*G53-903*G42) >> 10 F54 = (903*G54+483*G41) >> 10 F57 = (792*G57-650*G38) >> 10 F58 = (650*G58+792*G37) >> 10 F61 = (100*G61-1019*G34) >> 10 F62 = (1019*G62+100*G33) >> 10

Stage 4	Stage 5	Stage 6
E4 = (200*X8-1004*X56) >> 10 E5 = (851*X40-569*X24) >> 10 E6 = (851*X24+569*X40) >> 10 E7 = (200*X56+1004*X8) >> 10 E8 = F8+F9 E9 = F8-F9 E10 = F11-F10 E11 = F11+F10 E12 = F12+F13 E13 = F12-F13 E14 = F15-F14 E15 = F15+F14 E17 = (200*F30-1004*F17) >> 10 E18 = (-1004*F29-200*F18) >> 10 E21 = (851*F26-569*F21) >> 10 E22 = (-569*F25-851*F22) >> 10 E25 = (851*F25-569*F22) >> 10 E26 = (569*F26+851*F21) >> 10 E29 = (200*F29-1004*F18) >> 10 E30 = (1004*F30+200*F17) >> 10 E32 = G32+G35 E33 = F33+F34 E34 = F33-F34 E35 = G32-G35 E36 = G39-G36 E37 = F38-F37 E38 = F38+F37 E39 = G39+G36 E40 = G40+G43 E41 = F41+F42 E42 = F41-F42 E43 = G40-G43 E44 = G47-G44 E45 = F46-F45 E46 = F46+F45 E47 = G47+G44 E48 = G48+G51 E49 = F49+F50 E50 = F49-F50 E51 = G48-G51 E52 = G55-G52 E53 = F54-F53 E54 = F54+F53 E55 = G55+G52 E56 = G56+G59 E57 = F57+F58 E58 = F57-F58 E59 = G56-G59 E60 = G63-G60 E61 = F62-F61	D0 = (724*(X0+X32)) >> 10 D1 = (724*(X0-X32)) >> 10 D2 = (392*X16-946*X48) >> 10 D3 = (946*X16+392*X48) >> 10 D4 = E4+E5 D5 = E4-E5 D6 = E7-E6 D7 = E7+E6 D9 = (392*E14-946*E9) >> 10 D10 = (-946*E13-392*E10) >> 10 D13 = (392*E13-946*E10) >> 10 D14 = (946*E14+392*E9) >> 10 D16 = F16+F19 D19 = F16-F19 D20 = F23-F20 D23 = F23+F20 D24 = F24+F27 D27 = F24-F27 D28 = F31-F28 D31 = F31+F28 D17 = E17+E18 D18 = E17-E18 D21 = E22-E21 D22 = E22+E21 D25 = E25+E26 D26 = E25-E26 D29 = E30-E29 D30 = E30+E29 D34 = (200*E61-1004*E34) >> 10 D35 = (200*E60-1004*E35) >> 10 D36 = (-1004*E59-200*E36) >> 10 D37 = (-1004*E58-200*E37) >> 10 D42 = (851*E53-569*E42) >> 10 D43 = (851*E52-569*E43) >> 10 D44 = (-569*E51-851*E44) >> 10 D45 = (-569*E50-851*E45) >> 10 D50 = (851*E50-569*E45) >> 10 D51 = (851*E51-569*E44) >> 10 D52 = (569*E52+851*E43) >> 10 D53 = (569*E53+851*E42) >> 10 D58 = (200*E58-1004*E37) >> 10 D59 = (200*E59-1004*E36) >> 10 D60 = (1004*E60+200*E35) >> 10 D61 = (1004*E61+200*E34) >> 10	C0 = D0+D3 C3 = D0-D3 C8 = E8+E11 C11 = E8-E11 C12 = E15-E12 C15 = E15+E12 C1 = D1+D2 C2 = D1-D2 C9 = D9+D10 C10 = D9-D10 C13 = D14-D13 C14 = D14+D13 C5 = (724*(D6-D5)) >> 10 C6 = (724*(D6+D5)) >> 10 C18 = (392*D29-946*D18) >> 10 C20 = (-946*D27-392*D20) >> 10 C26 = (-946*D21+392*D26) >> 10 C28 = (392*D19+946*D28) >> 10 C19 = (392*D28-946*D19) >> 10 C21 = (-946*D26-392*D21) >> 10 C27 = (-946*D20+392*D27) >> 10 C29 = (392*D18+946*D29) >> 10 C32 = E32+E39 C39 = E32-E39 C40 = E47-E40 C47 = E47+E40 C48 = E48+E55 C55 = E48-E55 C56 = E63-E56 C63 = E63+E56 C33 = E33+E38 C38 = E33-E38 C41 = E46-E41 C46 = E46+E41 C49 = E49+E54 C54 = E49-E54 C57 = E62-E57 C62 = E62+E57 C34 = D34+D37 C37 = D34-D37 C42 = D45-D42 C45 = D45+D42 C50 = D50+D53 C53 = D50-D53 C58 = D61-D58 C61 = D61+D58 C35 = D35+D36 C36 = D35-D36 C43 = D44-D43 C44 = D44+D43

E62 = F62+F61 E63 = G63+G60		C51 = D51+D52 C52 = D51-D52 C59 = D60-D59 C60 = D60+D59

Stage 7	Stage 8	Stage 9
B0 = C0+D7 B7 = C0-D7 B1 = C1+C6 B6 = C1-C6 B2 = C2+C5 B5 = C2-C5 B3 = C3+D4 B4 = C3-D4 B10 = (724*(C13-C10)) >> 10 B13 = (724*(C13+C10)) >> 10 B11 = (724*(C12-C11)) >> 10 B12 = (724*(C12+C11)) >> 10 B16 = D16+D23 B23 = D16-D23 B24 = D31-D24 B31 = D31+D24 B17 = D17+D22 B22 = D17-D22 B25 = D30-D25 B30 = D30+D25 B18 = C18+C21 B21 = C18-C21 B26 = C29-C26 B29 = C29+C26 B19 = C19+C20 B20 = C19-C20 B27 = C28-C27 B28 = C28+C27 B36 = (392*C59-946*C36) >> 10 B40 = (-946*C55-392*C40) >> 10 B52 = (-946*C43+392*C52) >> 10 B56 = (392*C39+946*C56) >> 10 B37 = (392*C58-946*C37) >> 10 B41 = (-946*C54-392*C41) >> 10 B53 = (-946*C42+392*C53) >> 10 B57 = (392*C38+946*C57) >> 10 B38 = (392*C57-946*C38) >> 10 B42 = (-946*C53-392*C42) >> 10 B54 = (-946*C41+392*C54) >> 10 B58 = (392*C37+946*C58) >> 10 B39 = (392*C56-946*C39) >> 10 B43 = (-946*C52-392*C43) >> 10 B55 = (-946*C40+392*C55) >> 10 B59 = (392*C36+946*C59) >> 10	A0 = B0+C15 A15 = B0-C15 A1 = B1+C14 A14 = B1-C14 A2 = B2+B13 A13 = B2-B13 A3 = B3+B12 A12 = B3-B12 A4 = B4+B11 A11 = B4-B11 A5 = B5+B10 A10 = B5-B10 A6 = B6+C9 A9 = B6-C9 A7 = B7+C8 A8 = B7-C8 A20 = (724*(B27-B20)) >> 10 A27 = (724*(B27+B20)) >> 10 A21 = (724*(B26-B21)) >> 10 A26 = (724*(B26+B21)) >> 10 A22 = (724*(B25-B22)) >> 10 A25 = (724*(B25+B22)) >> 10 A23 = (724*(B24-B23)) >> 10 A24 = (724*(B24+B23)) >> 10 A32 = C32+C47 A47 = C32-C47 A48 = C63-C48 A63 = C63+C48 A33 = C33+C46 A46 = C33-C46 A49 = C62-C49 A62 = C62+C49 A34 = C34+C45 A45 = C34-C45 A50 = C61-C50 A61 = C61+C50 A35 = C35+C44 A44 = C35-C44 A51 = C60-C51 A60 = C60+C51 A36 = B36+B43 A43 = B36-B43 A52 = B59-B52 A59 = B59+B52 A37 = B37+B42 A42 = B37-B42 A53 = B58-B53 A58 = B58+B53 A38 = B38+B41 A41 = B38-B41 A54 = B57-B54 A57 = B57+B54 A39 = B39+B40 A40 = B39-B40 A55 = B56-B55 A56 = B56+B55	O0 = A0+B31 O31 = A0-B31 O1 = A1+B30 O30 = A1-B30 O2 = A2+B29 O29 = A2-B29 O3 = A3+B28 O28 = A3-B28 O4 = A4+A27 O27 = A4-A27 O5 = A5+A26 O26 = A5-A26 O6 = A6+A25 O25 = A6-A25 O7 = A7+A24 O24 = A7-A24 O8 = A8+A23 O23 = A8-A23 O9 = A9+A22 O22 = A9-A22 O10 = A10+A21 O21 = A10-A21 O11 = A11+A20 O20 = A11-A20 O12 = A12+B19 O19 = A12-B19 O13 = A13+B18 O18 = A13-B18 O14 = A14+B17 O17 = A14-B17 O15 = A15+B16 O16 = A15-B16 O40 = (724*(A55-A40)) >> 10 O55 = (724*(A55+A40)) >> 10 O41 = (724*(A54-A41)) >> 10 O54 = (724*(A54+A41)) >> 10 O42 = (724*(A53-A42)) >> 10 O53 = (724*(A53+A42)) >> 10 O43 = (724*(A52-A43)) >> 10 O52 = (724*(A52+A43)) >> 10 O44 = (724*(A51-A44)) >> 10 O51 = (724*(A51+A44)) >> 10 O45 = (724*(A50-A45)) >> 10 O50 = (724*(A50+A45)) >> 10 O46 = (724*(A49-A46)) >> 10 O49 = (724*(A49+A46)) >> 10 O47 = (724*(A48-A47)) >> 10 O48 = (724*(A48+A47)) >> 10

Stage 10

Y0 = O0+A63 Y63 = O0-A63 Y1 = O1+A62 Y62 = O1-A62 Y2 = O2+A61 Y61 = O2-A61 Y3 = O3+A60 Y60 = O3-A60 Y4 = O4+A59 Y59 = O4-A59 Y5 = O5+A58 Y58 = O5-A58 Y6 = O6+A57 Y57 = O6-A57 Y7 = O7+A56 Y56 = O7-A56 Y8 = O8+O55 Y55 = O8-O55 Y9 = O9+O54 Y54 = O9-O54 Y10 = O10+O53 Y53 = O10-O53	Y11 = O11+O52 Y52 = O11-O52 Y12 = O12+O51 Y51 = O12-O51 Y13 = O13+O50 Y50 = O13-O50 Y14 = O14+O49 Y49 = O14-O49 Y15 = O15+O48 Y48 = O15-O48 Y16 = O16+O47 Y47 = O16-O47 Y17 = O17+O46 Y46 = O17-O46 Y18 = O18+O45 Y45 = O18-O45 Y19 = O19+O44 Y20 = O20+O43 Y43 = O20-O43 Y21 = O21+O42 Y42 = O21-O42 Y22 = O22+O41	Y41 = O22-O41 Y23 = O23+O40 Y40 = O23-O40 Y24 = O24+A39 Y39 = O24-A39 Y25 = O25+A38 Y38 = O25-A38 Y26 = O26+A37 Y37 = O26-A37 Y27 = O27+A36 Y36 = O27-A36 Y28 = O28+A35 Y35 = O28-A35 Y29 = O29+A34 Y34 = O29-A34 Y30 = O30+A33 Y33 = O30-A33 Y31 = O31+A32 Y32 = O31-A32

6.4.6 Picture construction process prior to deblocking filter process

Inputs to this process are

- a sample array u with elements u_{ij} which is either a NxN luma block or NxN chroma block.
- largest coding unit address lcuAddr, prediction unit index puIdx and transform unit index tuIdx.

The position of the upper-left luma sample of the current largest coding unit is derived by invoking the inverse largest coding unit scanning process in subclause 4.2.1 with address lcuAddr as input and the output being assigned to (xL, yL).

The position of the upper-left luma sample of the current prediction unit puIdx inside the largest coding unit is derived by invoking the inverse prediction unit scanning process in subclause 4.2.4.1 with puIdx as input and the output being assigned to (xP, yP).

The position of the upper-left luma sample of the current transform unit tuIdx inside the prediction unit is derived by invoking the inverse transform unit scanning process in subclause 4.2.4.3 with tuIdx as input and the output being assigned to (xO, yO).

Let xS be xL + xP + xO and yS be yL + yP + yO.

When u is a luma block, for each sample u_{ij} of the luma block, the following applies.

$$S'_L[\ xS + j, yS + i\ ] = u_{ij} \quad \text{with } i, j = 0..N-1$$

When u is a chroma block, for each sample u_{ij} of the chroma block, the following ordered steps are specified:

1. The subscript C in the variable S'C is replaced with Cb for the Cb chroma component and with Cr for the Cr chroma component.

   $xS_C = xS / SubWidthC$

   $yS_C = yS / SubHeightC$

2. Then, the following applies.

   $S'_C[\ xS_C + j, yS_C + i\ ] = u_{ij}$ with i, j = 0..N-1

6.5 In-loop filter process

6.5.1 Deblocking filter process

The deblocking filter process shall be applied to all prediction unit partition edges and transform unit edges of a picture, except edges at the boundary of the picture, and edges of prediction unit partitions and transform unit partitions smaller than 8 pixels in either vertical or horizontal direction. For the transform units and prediction units with edges smaller than 8 pixels, only the edges lying on the 8x8 grid are filtered.

The deblocking filter process is invoked for the luma and chroma components separately. The filter process is in LCU order. Each LCU is processed in CU order. For each CU, vertical edges are filtered first, starting with the edge on the left-hand side of the prediction unit proceeding through the edges towards the right-hand side of the prediction unit in their geometrical order. Then, horizontal edges are filtered starting with the edge on the top of the CU proceeding through the edges towards the bottom of the CU in their geometrical order.

The variables filterInternalEdgesFlag, filterLeftPuEdgeFlag, filterTopPuEdgeFlag are derived as specified by the following ordered steps.

1. The derivation process for neighbouring largest coding units specified in subclause 4.3.4.1 is invoked and the output is assigned to lcuAddrA and lcuAddrB
2. The variables filterInternalEdgesFlag, filterLeftPuEdgeFlag and filterTopPuEdgeFlag are derived as specified by the following ordered steps:
   a. The variable filterInternalEdgesFlag is derived as follows.
      – the variable filterInternalEdgesFlag is set equal to 1
   b. The variable filterLeftPuEdgeFlag is derived as follows.
      – If any of the following conditions is true, the variable filterLeftPuEdgeFlag is set equal to 0:
        – CurrLcuAddr % PicWidthInLCUs is equal to 0
        – the largest coding unit lcuAddA is not available
      – Otherwise, the variable filterLeftPuEdgeFlag is set equal to 1
   c. The variable filterTopPuEdgeFlag is derived as follows.
      – If any of the following conditions is true, the variable filterTopPuEdgeFlag is set equal to 0:
        – CurrLcuAddr is less than PicWidthInLCUs
        – the largest coding unit lcuAddB is not available
      – Otherwise, the variable filterTopPuEdgeFlag is set equal to 1

Fig. 6-2 shows the possible edges from the partition boundaries inside the prediction unit which can be interpreted as luma or chroma edges.

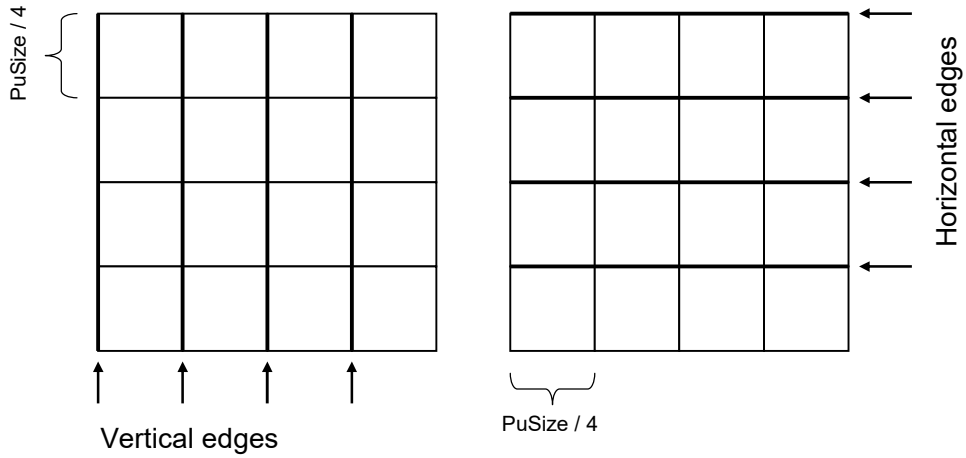


Fig. 6-2 Partition boundaries in a prediction unit to be filtered

When interpreting the edges in Fig. 6-2 as luma edges and chroma edges, the partition mode PartMode is considered as well as the value of the variables filterInternalEdgesFlag, filterLeftPUEdgeFlag and filterTopPUEdgeFlag.

The vertical edge interpretation depending on the partition mode is specified as follows:

- The first vertical edge can be filtered when filterLeftPuEdgeFlag is equal to 1 and the width of the prediction unit is equal or larger than 8 pixels.

- If PartMode is PART_nLx2N, the second vertical edge can be filtered when filterInternalEdgesFlag is equal to 1 and the distance between the first and the second vertical edges is equal or larger than 8 pixels.
- Otherwise, if PartMode is PART_Nx2N or PART_NxN, the third vertical edge can be filtered when filterInternalEdgesFlag is equal to 1 and the distance between the first and the third vertical edges is equal or larger than 8 pixels.
- Otherwise, if PartMode is PART_nRx2N, the fourth vertical edge can be filtered when filterInternalEdgesFlag is equal to 1 and the distance between the fourth vertical edges and the next PU or the picture boundary is equal or larger than 8 pixels.

The horizontal edge interpretation depending on the partition mode is specified as follows:

- The first horizontal edge can be filtered when filterTopPuEdgeFlag is equal to 1 and the height of the prediction unit is equal or larger than 8 pixels.
- If PartMode is PART_2NxnU, the second horizontal edge can be filtered when filterInternalEdgesFlag is equal to 1 and the distance between the first and the second horizontal edges is equal or larger than 8 pixels.
- Otherwise, if PartMode is PART_2NxN or PART_NxN, the third horizontal edge can be filtered when filterInternalEdgesFlag is equal to 1 and the distance between the first and the third horizontal edges is equal or larger than 8 pixels.
- Otherwise, if PartMode is PART_2NxnD, the fourth horizontal edge can filtered when filterInternalEdgesFlag is equal to 1 and the distance between the fourth horizontal edges and the next PU or the picture boundary is equal or larger than 8 pixels.

The vertical and horizontal edges interpretation depending on transform unit size is specified as follows:

Fig. 5-3 shows the possible edges from the partition boundaries inside the transform unit which can be interpreted as luma or chroma edges.

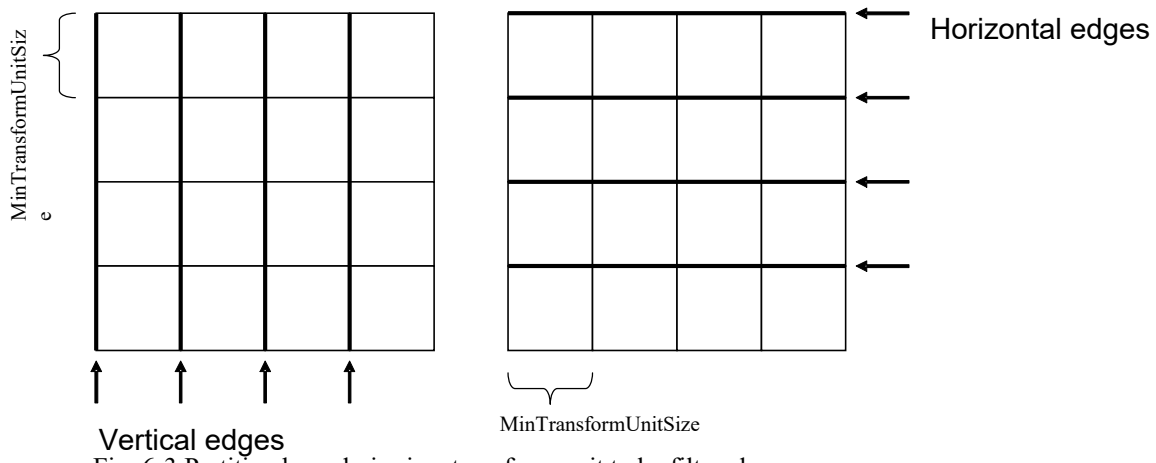


Fig. 6-3 Partition boundaries in a transform unit to be filtered

Only vertical edges in the transform unit with a distance equal to 8 or larger can be filtered. This means that if MinTransformUnitSize=4 only every second vertical edge can be filtered. Only horizontal edges in the transform unit with a distance equal to 8 or larger can be filtered. This means that if MinTransformUnitSize=4 only every second horizontal edge can be filtered.

When a horizontal or vertical edge from either partitions of a prediction unit or from partitions of a transform unit can be filtered according to above processes, another process follows to check if parts of respective edge can be filtered. The deblocking filter operates on k parts, where k ranges 0 to K-1, of edges for luma as well as for chroma, where K is the width or height of the current CU divided by 8 for horisontal edges and vertical edges respectively. The size of each part is 8 pixel. The pixels on respective side of parts of the edges are in the following denoted A and B where B corresponds to a part of the partition at largest coding unit address lcuAddr, prediction unit index

puIdx and transform unit index tuIdx. In the following, A corresponds to the left neighbouring partition to B for vertical edges and to the above neighbouring partition to B for horizontal edges.

$p3_0$ $p2_0$ $p1_0$ $p0_0$	$q0_0$ $q1_0$ $q2_0$ $q3_0$
$p3_1$ $p2_1$ $p1_1$ $p0_1$	$q0_1$ $q1_1$ $q2_1$ $q3_1$
$p3_2$ $p2_2$ $p1_2$ $p0_2$	$q0_2$ $q1_2$ $q2_2$ $q3_2$
A $p3_3$ $p2_3$ $p1_3$ $p0_3$	$q0_3$ $q1_3$ $q2_3$ $q3_3$ **B**
$P3_4$ $p2_4$ $p1_4$ $p0_4$	$q0_4$ $q1_4$ $q2_4$ $q3_4$
$P3_5$ $p2_5$ $p1_5$ $p0_5$	$q0_5$ $q1_5$ $q2_5$ $q3_5$
$P3_6$ $p2_6$ $p1_6$ $p0_6$	$q0_6$ $q1_6$ $q2_6$ $q3_6$
$P3_7$ $p2_7$ $p1_7$ $p0_7$	$q0_7$ $q1_7$ $q2_7$ $q3_7$

Notation of an 8 pixel part of a vertical edge for deblocking

The k:th part of a vertical edge in luma the pixels in partition A and B have following correspondance with reconstructed pixel values in luma:
$qj_i = S'_L[\ xS + j, yS + i + k*8]$
$pj_i = S'_L[\ xS - j - 1, yS + i + k*8]$
with j=0..3 and i=0..7
where S'_L , xS and yS are defined in 5.3.5.

The k:th part of a horizontal edge in luma the pixels in partition A and B have following correspondance with reconstructed pixel values in luma:
$qj_i = S'_L[\ xS + j + k*8, yS + i\ ]$
$pj_i = S'_L[\ xS + j + k*8, yS - i - 1\ ]$
with j=0..7 and i=0..3

The k:th part of a vertical edge in chroma the pixels in partition A and B have following correspondance with reconstructed pixel values in chroma:
$qj_i = S'_C[\ xS_C + j, yS_C + i + k*8]$
$pj_i = S'_C[\ xS_C - j - 1, yS_C + i+ k*8\ ]$
with j=0..3 and i=0..7
where S'_C , xS_C and yS_C are defined in 5.3.5.

The k:th part of a horizontal edge in chroma the pixels in partition A and B have following correspondance with reconstructed pixel values in chroma:
$qj_i = S'_C[\ xS_C + j+ k*8, yS_C + i\ ]$
$pj_i = S'_C[\ xS_C + j+ k*8, yS_C - i - 1]$
with j=0..7 and i=0..3

Luma filtering

A part of the luma edge between A and B can be filtered if one of the following conditions is true

- A or B has PredMode==MODE_INTRA
- ***or*** A or B has nonzero transform coefficients
- ***or*** A and B use different reference frames or different number of motion vectors
- ***or*** one motion vector is used to predict partition A and one motion vector is used to predict the partition B and the absolute difference between the horizontal or vertical component of the motion vectors used is greater than or equal to 4 in units of quarter luma frame samples,
- ***or*** two motion vectors and two different reference pictures are used to predict the partition A and two motion vectors for the same two reference pictures are used to predict the partition B and the absolute difference between the horizontal or vertical component of the two motion vectors used in the prediction of the two partitions for the same reference picture is greater than or equal to 4 in units of quarter luma frame samples,
- ***or*** two motion vectors for the same reference picture are used to predict the partition A and two motion vectors for the same reference picture are used to predict the partition B and the absolute difference between the horizontal or vertical component of the corresponding motion vectors used in the prediction of the two partitions is greater than or equal to 4 in units of quarter luma frame samples.
- ***or*** a PBIC value (icScale or icOffset) of partition A used for a reference picture differs from the corresponding value of partition B

- ***and*** $|p2_2 - 2*p1_2 + p0_2| + |q2_2 - 2*q1_2 + q0_2| + |p2_5 - 2*p1_5 + p0_5| + |q2_5 - 2*q1_5 + q0_5| < \beta$

The filtering is also performed on the 8 pixel edge if the conditions described above are fulfilled for any pair of the smaller partitions or transform units on the different sides of the edge. For example, if A consists of two 4x4 partitions (A_1 and A_2) then the above process except the last item is done for both partition pairs A_1 - B and A_2 – B , and the last item is applied for the pair A-B. If pairs A_1 - B ***or*** pair A_2 - B need to be filtered then the whole 8 pixel edge between A and B is filtered.

A part of the luma edge between A and B is ***not*** filtered if one of the following conditions is true.

- A and B belong to the same transform unit partition and the same prediction unit partition
- ***or*** the edge is a prediction unit border ***and*** both A and B have planar_flag ==1.

If the edge between A and B is filtered, one of two types of filtering (weak or strong filtering) is performed. The choice between the strong and the weak filtering is done separately for each line depending on the following conditions. For each line i = 0,7, the strong filtering is performed if all the following conditions are true, otherwise, weak filtering is performed.

- $d < (\beta >> 2)$
- ***and*** $(|p3_i - p0_i| + |q0_i - q3_i|) < (\beta >> 3)$
- ***and*** $|p0_i - q0_i| < ((5*t_C + 1) >> 1)$,

where t_C and β depends on qp their relations are shown in the table below. The t_C is calculated as t_C (qp+4) when A ***or*** B has PredMode==MODE_INTRA.

Relation between qp, tc and β

qp	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
β	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	7
qp	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
tc	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	4
β	8	9	10	11	12	13	14	15	16	17	18	20	22	24	26	28	30	32
qp	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
tc	4	4	5	5	6	6	7	8	9	9	10	10	11	11	12	12	13	13
β	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62	64		
qp	54	55																
tc	14	14																
β																		

Weak filtering

Weak filtering mode filtering is performed based on the above conditions,

$\Delta = Clip(-t_C, t_C, (13*(q0_i - p0_i) + 4*(q1_i - p1_i) - 5*(q2_i - p2_i) + 16) >> 5))$ i = 0,7

$p0_i = Clip_{0\text{-}255}(p0_i + \Delta)$ i = 0,7

$q0_i = Clip_{0\text{-}255}(q0_i - \Delta)$ i = 0,7

$p1_i = Clip_{0\text{-}255}(p1_i + \Delta/2)$ i = 0,7

$q1_i = Clip_{0\text{-}255}(q1_i - \Delta/2)$ i = 0,7

Strong filtering

Strong filtering mode is performed with the following set of operations.

$p0_0 = Clip_{0\text{-}255}((p2_i + 2*p1_i + 2*p0_i + 2*q0_i + q1_i + 4) >> 3)$; i = 0,7

$q0_0 = Clip_{0\text{-}255}((p1_i + 2*p0_i + 2*q0_i + 2*q1_i + q2_i + 4) >> 3)$; i = 0,7

$p1_0 = Clip_{0\text{-}255}((p2_i + p1_i + p0_i + q0_i + 2) >> 2)$; i = 0,7

$q1_0 = Clip_{0\text{-}255}((p0_i + q0_i + q1_i + q2_i + 2) >> 2)$; i = 0,7

$p2_0 = Clip_{0\text{-}255}((2*p3_i + 3*p2_i + p1_i + p0_i + q0_i + 4) >> 3)$; i = 0,7

$q2_0 = Clip_{0\text{-}255}((p0_i + q0_i + q1_i + 3*q2_i + 2*q3_i + 4) >> 3)$; i = 0,7

Chroma filtering

The chroma edge between A and B can be filtered if the following condition is true.

- A or B has PredMode==MODE_INTRA

The filtering is not applied if both following conditions are true.

- The edge is a prediction unit edge

and both A and B have planar_flag==1.

If filtering is performed based on the above condition,

$\Delta = Clip(-t_C, t_C, ((((q0_i - p0_i) << 2) + p1_i - q1_i + 4) >> 3))$ $\quad i = 0,7$

$p0_i = Clip_{0\text{-}255}(p0_i + \Delta)$ $\quad i = 0,7$

$q0_i = Clip_{0\text{-}255}(q0_i - \Delta)$ $\quad i = 0,7$

t_C depend on qp and their relations are shown in the table above. For chroma filtering t_C is calculated as t_C (qp+4).

Planar mode filtering

There is a special filtering mode which is applied to the prediction unit edge when both prediction units are encoded in intra planar mode. Luma and chroma edges between A and B are filtered with *planar filtering* if all of the following conditions are true, otherwise normal filtering described in Luma and Chroma filtering are performed.

- Edge between A and B is a prediction unit edge
- Both A ***and*** B have planar_flag==1
- The prediction units for both A and B have the same size.

Let variable blkSize equals to width (or height) of prediction unit, blkSizeHalf= blkSize>>1 and blkSizeQuarter = blkSize>>2 and bitShift = log_2(blkSize) - 1. If luma edges are filtered, let S = S'_L , xPos = xS, yPos = yS, otherwise let S = S'_C , xPos = xS_C, yPos = yS_C

The horizontal planar mode filtering is performed with the following set of operations.

```
for(k=0;k<blkSize;k++) {
    a1 = S [xPos - blkSizeQuarter][yPos+k];
    a2 = S [xPos + blkSizeQuarter][yPos+k];
    for(l=1;l<blkSizeHalf;l++)
        S[xPos-blkSizeQuarter+l] [yPos+k] = (a1*(blkSizeHalf-l) + a2*l + blkSizeQuarter) >> bitShift;
}
```

The vertical planar mode filtering for luma component is performed with the following set of operations.

```
for(k=0; k<blkSize; k++) {
    a1 = S[xPos +k] [yPos-blkSizeQuarter];
    a2 = S[xPos +k] [yPos+blkSizeQuarter];
    for(l=1; l<blkSizeHalf; l++)
        S[xPos +k][ yPos -blkSizeQuarter+l] = (a1*(blkSizeHalf-l) + a2*l + blkSizeQuarter) >> bitShift;
}
```

6.5.2 Adaptive loop filter process

This process is invoked when adaptive_loop_filter_flag is equal to 1.

This process is performed on a coding unit basis after the completion of the slice construction process prior to adaptive loop filter process for the entire decoded slice, with all coding units in a slice processed in order of coding unit scan order.

~~The following filter~~Filter coefficients c_L for luma samples and c_C for chroma samples are derived by invoking the process specified in subclause 6.5.2.1.

- ~~Filter coefficients c_{L_hor} for horizontal filtering of the samples of the reconstructed signal of the luma component~~
- ~~Filter coefficients c_{L_ver} for vertical filtering of the luma samples of the reconstructed signal of the luma component~~

- ~~Filter coefficients $c_{L_pred_hor}$ for horizontal filtering of the samples of the prediction signal of the luma component~~
- ~~Filter coefficients $c_{L_pred_ver}$ for vertical filtering of the samples of the prediction signal of the luma component~~
- ~~Filter coefficients $c_{L_qpe_hor}$ for horizontal filtering of the samples of the quantized prediction error signal of the luma component~~
- ~~Filter coefficients $c_{L_qpe_ver}$ for vertical filtering of the samples of the quantized prediction error signal of the luma component~~
- ~~Filter coefficients c_{C_hor} for horizontal filtering of the samples of the reconstructed signal of the chroma components~~
- ~~Filter coefficients c_{C_ver} for vertical filtering of the samples of the reconstructed signal of the chroma components~~
- ~~Filter coefficients $c_{C_pred_hor}$ for horizontal filtering of the samples of the prediction signal of the chroma components~~
- ~~Filter coefficients $c_{C_pred_ver}$ for vertical filtering of the samples of the prediction signal of the chroma components~~
- ~~Filter coefficients $c_{C_qpe_hor}$ for horizontal filtering of the samples of the quantized prediction error signal of the chroma components~~
- ~~Filter coefficients $c_{C_qpe_ver}$ for vertical filtering of the samples of the quantized prediction error signal of the chroma components~~

When interpreting the luma samples of the coding unit as to be filtered, depending on alf_cu_control_flag and alf_flag, the following applies.

- If alf_cu_control_flag is equal to 0, the luma samples are filtered,
- Otherwise (alf_cu_control_flag is equal to 1), if alf_flag is equal to 1, the luma samples are filtered.
- Otherwise, the luma samples are not filtered.

When interpreting the chroma samples of the coding unit as to be filtered, depending on alf_chroma_idc, the following applies.

- If alf_chroma_idc is equal to 1, the Cr samples are filtered.
- Otherwise, if alf_chroma_idc is equal to 2, the Cb samples are filtered.
- Otherwise, if alf_chroma_idc is equal to 3, both chroma samples are filtered.
- Otherwise (alf_chroma_idc is equal to 0), both chroma samples are not filtered.

For the luma samples of the coding unit interpreted as to be filtered, the following ordered steps are specified:

1. The position of the upper-left sample of the largest coding unit with address lcuAddr is derived by invoking the inverse largest coding unit scanning process in subclause 4.2.1 with lcuAddr as the input and the output being assigned to (xL, yL).
2. The relative position of the upper-left sample of the current coding unit with index cuIdx is derived by invoking the inverse coding unit scanning process in subclause 4.2.3 with cuIdx as the input and the output being assigned to (xP, yP).
3. The filtering process for luma samples specified in subclause 6.5.2.2 is invoked with the position (xL+xP,yL+yP), the samples of the reconstructed signal (after deblocking) of the luma component $S'_L[\ x, y\ ]$, the samples of the prediction signal of the luma component $S_{L_pred}[\ x, y\ ]$, and the samples of the quantized prediction error signal of the luma component $S_{L_qpe}[\ x, y\ ]$, with x = 0..$PicWidthInSamples_L$, y = 0..$PicHeightInSamples_L$ as input and the output being assigned to pf[x, y], with x, y = 0..CuSize-1 as output.
4. The reconstructed samples in the current slice are updated as follows.

$$S''_L[\ xL + xP + x, yL + yP + y\ ] = pf[\ x, y\ ], \text{ with } x, y = 0..CuSize - 1$$

For the chroma samples of the coding unit interpreted as to be filtered, the following ordered steps are specified (with C being Cb or Cr):

1. The position of the upper-left sample of the largest codint unit with address lcuAddr is derived by invoking the inverse largest coding unit scanning process in subclause 4.2.1 with lcuAddr as the input and the output being assigned to (xL, yL).

$xL_C = xL$ / SubWidthC

$yL_C = yL$ / SubHeightC

2. The relative position of the upper-left sample of the current codint unit with index cuIdx is derived by invoking the inverse coding unit scanning process in subclause 4.2.3 with cuIdx as the input and the output being assigned to (xP, yP).

$xP_C = xP$ / SubWidthC

$yP_C = yP$ / SubHeightC

3. The filtering process for luma samples specified in subclause 6.5.2.3 is invoked with the position ($xL_C + xP_C$, $yL_C + yP_C$), the samples of the reconstructed signal (after deblocking) of the chroma component S'_C[x, y], the samples of the prediction signal of the chroma component S_{C_pred}[x, y], and the samples of the quantized prediction error signal of the chroma component S_{C_qpe}[x, y], with x = 0..$PicWidthInSamples_C$, y = 0..$PicHeightInSamples_C$ as input and the output being assigned to pf[x, y], with x, y = 0..CuSizeC-1 as output.

4. The reconstructed samples in the current slice are updated as follows.

$S''_C[\ xL_C + xP_C + x,\ yL_C + yP_C + y\ ]$ = pf[x, y], with x, y = 0..CuSizeC – 1

6.5.2.1 Derivation process for filter coefficients

Outputs of this process are the filter coefficients c_L for the luma samples and the filter coefficients c_C for the chroma samples.

- ~~Filter coefficients c_{L_hor} for horizontal filtering of the samples of the reconstructed signal of the luma component~~
- ~~Filter coefficients c_{L_ver} for vertical filtering of the luma samples of the reconstructed signal of the luma component~~
- ~~Filter coefficients $c_{L_pred_hor}$ for horizontal filtering of the samples of the prediction signal of the luma component~~
- ~~Filter coefficients $c_{L_pred_ver}$ for vertical filtering of the samples of the prediction signal of the luma component~~
- ~~Filter coefficients $c_{L_qpe_hor}$ for horizontal filtering of the samples of the quantized prediction error signal of the luma component~~
- ~~Filter coefficients $c_{L_qpe_ver}$ for vertical filtering of the samples of the quantized prediction error signal of the luma component~~
- ~~Filter coefficients c_{C_hor} for horizontal filtering of the samples of the reconstructed signal of the chroma components~~
- ~~Filter coefficients c_{C_ver} for vertical filtering of the samples of the reconstructed signal of the chroma components~~
- ~~Filter coefficients $c_{C_pred_hor}$ for horizontal filtering of the samples of the prediction signal of the chroma components~~
- ~~Filter coefficients $c_{C_pred_ver}$ for vertical filtering of the samples of the prediction signal of the chroma components~~
- ~~Filter coefficients $c_{C_qpe_hor}$ for horizontal filtering of the samples of the quantized prediction error signal of the chroma components~~
- ~~Filter coefficients $c_{C_qpe_ver}$ for vertical filtering of the samples of the quantized prediction error signal of the chroma components~~

~~Derivation of the coefficients of the horizontal filters:~~

~~c_{L_hor}~~ [i][j] with i=0...AlfNumFilters-1, j = 0..~~AlfLengthLuma_hor~~AlfNumCoeffLuma – 1 shall be derived as follows.

The first luma filter c_{L_hor}[0] 's coefficients with elements c_{L_hor}[0][j], j = 0..~~AlfLengthLuma_hor~~AlfNumCoeffLuma – 1 shall be derived as,

c_{L_hor}[0][j] = alf_coeff_luma~~_hor~~[0][j]

~~for j = 0..(AlfLengthLuma_hor – 1)/2-1~~

~~c_{L_hor}[0][j] = (1<<alf_quant_step_size_luma_hor) + alf_coeff_luma_hor[0][j]~~

~~for j = (AlfLengthLuma_hor – 1)/2~~

~~c_{L_hor}[0][j] = alf_coeff_luma_hor[0][j]~~

~~for j = (AlfLengthLuma_hor – 1)/2+1.. AlfLengthLuma_hor~~AlfNumCoeffLuma – 1.

The other luma filter c_{L_hor}[i]'s coefficients with elements c_{L_hor}[i][j], i=1...AlfNumFilters-1, j = 0.. ~~AlfLengthLuma_hor~~AlfNumCoeffLuma – 1 shall be derived as follows.

If Alf_PredMethod is equal to 1,

c_{L_hor}[i][j] = alf_coeff_luma~~_hor~~[i][j] + c_{L_hor}[i-1][j] for j = 0.. ~~AlfLengthLuma_hor~~AlfNumCoeffLuma – 1

otherwise,

c_{L_hor}[i][j] = alf_coeff_luma~~_hor~~[i][j] for j = 0.. ~~AlfLengthLuma_hor~~AlfNumCoeffLuma – 1

~~$c_{L_pred_hor}$ [i][j] with i=0...AlfNumFilters-1, j = 0..AlfLengthLuma_pred_hor – 1 shall be derived as follows.~~

~~The first luma filter $c_{L_pred_hor}$[0] 's coefficients with elements $c_{L_pred_hor}$[0][j], j = 0.. AlfLengthLuma_pred_hor – 1 shall be derived as,~~

~~$c_{L_pred_hor}$ [0][j] = alf_coeff_luma_pred_hor[0][j] for j = 0.. (AlfLengthLuma_pred_hor – 1)/2-1~~

~~$c_{L_pred_hor}$ [0][j] = alf_coeff_luma_pred_hor[0][j] - alf_coeff_luma_hor[0][AlfLengthLuma_hor – 1)/2]~~

~~for j = (AlfLengthLuma_pred_hor – 1)/2~~

~~$c_{L_pred_hor}$ [0][j] = alf_coeff_luma_pred_hor[0][j]~~

~~for j = (AlfLengthLuma_pred_hor – 1)/2+1.. AlfLengthLuma_pred_hor – 1~~

~~The other luma filter $c_{L_pred_hor}$[i]'s coefficients with elements $c_{L_pred_hor}$[i][j], i=1...AlfNumFilters-1, j = 0.. AlfLengthLuma_pred_hor – 1 shall be derived as follows.~~

~~If Alf_PredMethod is equal to 1,~~

~~$c_{L_pred_hor}$[i][j] = alf_coeff_luma_pred_hor[i][j] + $c_{L_pred_hor}$[i-1][j]~~

~~for j = 0.. AlfLengthLuma_pred_hor – 1~~

~~otherwise,~~

~~$c_{L_pred_hor}$[i][j] = alf_coeff_luma_pred_hor[i][j] for j = 0.. AlfLengthLuma_pred_hor – 1~~

~~The first luma filter $c_{L_qpe_hor}$[0] 's coefficients with elements $c_{L_qpe_hor}$[0][j], j = 0.. AlfLengthLuma_qpe_hor – 1 shall be derived as,~~

~~$c_{L_qpe_hor}$[0][j] = alf_coeff_luma_qpe_hor[0][j] for j = 0.. (AlfLengthLuma_qpe_hor – 1)/2-1~~

~~$c_{L_qpe_hor}$[0][j] =~~

~~$c_{L_pred_hor}$ [0][(AlfLengthLuma_pred_hor – 1)/2] + alf_coeff_luma_qpe_hor[0][j]~~

~~for j = (AlfLengthLuma_qpe_hor – 1)/2~~

~~$c_{L_qpe_hor}$[0][j] = alf_coeff_luma_qpe_hor[0][j]~~

~~for j = (AlfLengthLuma_qpe_hor – 1)/2+1.. AlfLengthLuma_qpe_hor – 1~~

~~The other luma filter $c_{L_qpe_hor}$[i]'s coefficients with elements $c_{L_qpe_hor}$[i][j], i=1...AlfNumFilters-1, j = 0.. AlfLengthLuma_qpe_hor – 1 shall be derived as follows.~~

~~If Alf_PredMethod is equal to 1,~~

~~$c_{L_qpe_hor}$[i][j] = alf_coeff_luma_qpe_hor[i][j] + $c_{L_qpe_hor}$[i-1][j]~~

~~for j = 0.. AlfLengthLuma_qpe_hor – 1~~

~~otherwise,~~

~~$c_{L_qpe_hor}$[i][j] = alf_coeff_luma_qpe_hor[i][j] for j = 0.. AlfLengthLuma_qpe_hor – 1~~

The chroma filter coefficients c_{C_hor} with elements $c_{C_hor}[i]$, i = 0..~~AlfLengthChroma_hor~~AlfNumCoeffChroma – 1 ~~are~~is derived as follows.

$c_{C_hor}[i]$ = alf_coeff_chroma~~_hor~~[i] for i = 0..~~AlfLengthChroma_hor~~AlfNumCoeffChroma – 2

$c_{C_hor}[i]$ = ~~(1<<alf_quant_step_size_chroma_hor)~~255 – sum – alf_coeff_chroma~~_hor~~[i], otherwise

where

sum = ∑(m_i * alf_coeff_chroma~~_hor~~[i]) with i = 0..~~AlfLengthChroma_hor~~AlfNumCoeffChroma - 2

m_i = 2 with i = 0⋯ ~~AlfLengthChroma_hor~~AlfNumCoeffChroma – 3

= 1 with i = ~~AlfLengthChroma_hor~~AlfNumCoeffChroma – 2

~~The chroma filter coefficients $c_{C_pred_hor}$ with elements $c_{C_pred_hor}[i]$, i = 0.. AlfLengthChroma_pred_hor – 1 are derived as follows.~~

~~$c_{C_pred_hor}[i]$ = alf_coeff_chroma_pred_hor [i]~~

~~for i = 0.. (AlfLengthChroma_pred_hor – 1)/2-1~~

~~$c_{C_pred_hor}[i]$ = alf_coeff_chroma_pred_hor[i] - alf_coeff_chroma_hor[(AlfLengthChroma_hor – 1)/2]~~

~~for i = (AlfLengthChroma_pred_hor – 1)/2~~

~~$c_{C_pred_hor}[i]$ = alf_coeff_chroma_pred_hor[i]~~

~~for i = (AlfLengthChroma_pred_hor – 1)/2+1.. AlfLengthChroma_pred_hor – 1~~

~~The chroma filter coefficients $c_{C_qpe_hor}$ with elements $c_{C_qpe_hor}[i]$, i = 0.. AlfLengthChroma_qpe_hor – 1 are derived as follows.~~

~~$c_{C_qpe_hor}[i]$ = alf_coeff_chroma_qpe_hor [i]~~

~~for i = 0.. (AlfLengthChroma_qpe_hor – 1)/2-1~~

~~$c_{C_qpe_hor}[i]$ = alf_coeff_chroma_qpe_hor[i] + $c_{C_pred_hor}$[(AlfLengthChroma_pred_hor – 1)/2]~~

~~for i = (AlfLengthChroma_qpe_hor – 1)/2~~

~~$c_{C_qpe_hor}[i]$ = alf_coeff_chroma_qpe_hor[i]~~

~~for i = (AlfLengthChroma_qpe_hor – 1)/2+1.. AlfLengthChroma_qpe_hor – 1~~

~~The derivation of the coefficients of the vertical filters follows the one of the vertical filters as specified above. The subscript "hor" need to be replaced by the subscript "ver".~~

6.5.2.2 Filter process for luma samples

Inputs to this process are:

- Position (ox, oy)
- Samples of the reconstructed signal of the luma component S'_L[x, y] ,

  with x = 0..$PicWidthInSamples_L$, y = 0..$PicHeightInSamples_L$
- Samples of the prediction signal of the luma component S_{L_pred}[x, y] ,

  with x = 0..$PicWidthInSamples_L$, y = 0..$PicHeightInSamples_L$
- Samples of the quantized prediction error signal of the luma component S_{L_qpe}[x, y],

  with x = 0..$PicWidthInSamples_L$, y = 0..$PicHeightInSamples_L$

Output of this process are:

– Filtered samples pf[x, y], with x, y = 0..CuSize-1 as output

In a first step, the vertical filter process is performed as follows:

pf~~_tmp~~[x, y] =[~~∑ (~~∑ ∑ (c$_{L_ver}$[i,j] * S'$_L$[x+ox~~, y+oy~~+i-((AlfLengthLuma~~_ver~~-1)>>1~~)].)~~

~~+∑ (c$_{L_pred_ver}$[j] * S$_{L_pred}$ [x+ox,~~), y+~~-~~oy+j-((AlfLengthLuma~~_pred_ver~~-1)>>1)] ~~))~~]

~~+∑ (c$_{L_qpe_ver}$[k] * S$_{L_qpe}$ [x+ox, y+ oy+k-((AlfLengthLuma_qpe_ver-1)>>1)].)]~~

>> Alf_filter_precision_Luma~~_ver~~

with i = 0.. AlfLengthLuma~~_ver~~-1, j = 0.. AlfLengthLuma~~_pred_ver -1, k = 0.. AlfLengthLuma_qpe_ver~~ -1.

~~In a second step, the horizontal filter process is performed as follows:~~

~~pf[x, y] =[∑ (c$_{L_hor}$[i] * pf_tmp')~~

~~+∑ (c$_{L_pred_hor}$[j] * S$_{L_pred}$ [x+ox +j-((AlfLengthLuma_pred_hor-1)>>1), y+ oy].)~~

~~+∑ (c$_{L_qpe_hor}$[k] * S$_{L_qpe}$ [x+ox +k-((AlfLengthLuma_qpe_hor-1)>>1), y+ oy].)]~~

~~>> Alf_filter_precision_Luma_hor~~

~~with i = 0.. AlfLengthLuma_hor -1, j = 0.. AlfLengthLuma_pred_hor -1, k = 0.. AlfLengthLuma_qpe_hor -1.~~

~~The variable pf_tmp' is derived as follows:~~

- ~~If pf_tmp [x +j-((AlfLengthLuma_hor-1)>>1), y] is the result of the vertical filter process, then pf_tmp' is set to pf_tmp[x +j-((AlfLengthLuma_hor-1)>>1), y]~~
- ~~Otherwise (if pf_tmp[x +j-((AlfLengthLuma_hor-1)>>1), y] is not the result of the vertical filter process), then pf_tmp' is set to S'$_L$ [x +xo+j-((AlfLengthLuma_hor-1)>>1), y+yo]~~

6.5.2.3 Filter process for chroma samples

Inputs to this process are:

- Position (ox, oy)
- Samples of the reconstructed signal of the chroma component S'$_C$[x, y] ,

  with x = 0..PicWidthInSamples$_C$, y = 0..PicHeightInSamples$_C$
- Samples of the prediction signal of the chroma component S$_{C_pred}$[x, y] ,

  with x = 0..PicWidthInSamples$_C$, y = 0..PicHeightInSamples$_C$
- Samples of the quantized prediction error signal of the chroma component S$_{C_qpe}$[x, y],

  with x = 0..PicWidthInSamples$_C$, y = 0..PicHeightInSamples$_C$

Output of this process is:

- Filtered samples pf[x, y], with x, y = 0..CuSizeC-1 as output

In a first step, the vertical filter process is performed as follows:

pf_tmp[x, y] =[~~∑ (~~∑ ∑ (c$_{C_ver}$[i,j] * S'$_C$[x+ox~~, y+oy~~+i-((AlfLengthChroma~~_ver~~-1)>>1~~)].)~~

~~+∑ (c$_{C_pred_ver}$[j] * S$_{C_pred}$ [x+ox,~~), y+~~-~~oy+j-((AlfLengthChroma~~_pred_ver~~-1)>>1)] ~~))~~]

~~+∑ (c$_{C_qpe_ver}$[k] * S$_{C_qpe}$ [x+ox, y+ oy+k-((AlfLengthChroma_qpe_ver-1)>>1)].)]~~

>> Alf_filter_precision_Chroma~~_ver~~

with i = 0.. AlfLengthChroma~~_ver~~-1, j = 0.. AlfLengthChroma~~_pred_ver -1, k = 0.. AlfLengthChroma_qpe_ver~~ -1.

~~In a second step, the horizontal filter process is performed as follows:~~

~~pf[x, y] =[∑ (c$_{C_hor}$[i] * pf_tmp')~~

~~+∑ (c$_{C_pred_hor}$[j] * S$_{C_pred}$ [x+ox+j-((AlfLengthChroma_pred_hor-1)>>1), y+oy].)~~

~~+∑ ($e_{C_qpe_hor}$[k] * S_{C_qpe} [x+ ox+k -((AlfLengthChroma_qpe_hor-1)>>1), y+oy].)]~~

~~>> Alf_filter_precision_Chroma_hor~~

~~with i = 0.. AlfLengthChroma_hor -1, j = 0.. AlfLengthChroma_pred_hor-1, k = 0.. AlfLengthChroma_qpe_hor-1.~~

~~The variable pf_tmp' is derived as follows:~~

- ~~If pf_tmp [x +j -((AlfLengthChroma_hor-1)>>1), y] is the result of the vertical filter process, then pf_tmp' is set to pf_tmp[x +j -((AlfLengthChroma_hor-1)>>1), y]~~
- ~~Otherwise (if pf_tmp[x +j-((AlfLengthChroma _hor-1)>>1), y] is not the result of the vertical filter process), then pf_tmp' is set to S'$_C$ [x +xo+j -((AlfLengthChroma _hor-1)>>1), y+yo]~~

_[Ed.: the indices may need some refinement]

[Ed.: clipping to be added]

[3-Input-ALF is still missing]

7 Parsing Process

Inputs to this process are bits from the RBSP.
Outputs of this process are syntax element values.

7.1 VLC Parsing process for side information

[Ed.: first draft from A119]
Following ordered steps are specified:

1. The context variable, ctxID is derived (subclause 7.1.1).
2. The modeIdx variable is decoded and the variables related to prediction unit transform unit are computed (subclause 7.1.2).
3. For each one of the transform unit within the coding unit, the coded block pattern information is computed (subclause 7.1.3).

7.1.1 Derivation of ctxID for prediction unit mode

Input to this process is the slice tpye specified by slice_type. Output of this process is the context identification variable ctxID.

The ctxID variable is given as:

ctxID = 0 for slice_type == I_SLICE
ctxID = 1 for slice_type == P_SLICE
ctxID = 2 for slice_type == B_SLICE

7.1.2 VLC Parsing process for prediction unit mode

This process is invoked when entropy_coding_mode_flag is equal to 0.

The syntax element **mode_table_idx** is decoded using subclause 7.1.2.1. The variable modeIdx is decoded using subclause 7.1.2.2.

The modeIdx variable is used to decode the values of the following:

- pred_mode
- intra_split_flag
- inter_partitioning_idc
- ref_idx_l0
- inter_pred_idc
- split_transform_unit_flag

– PU_dQP_idx

[Ed.: more details to be further edited]

7.1.2.1 Parsing process for the mode_table_idx

Input to this process is context variable array modeTableVLC_idx.
The **mode_table_idx** syntax element is decoded from the bitstream using VLC table given by modeTableVLCTable[modeTableVLC_idx[ctxID]]. The values of VLC table array modeTableVLCTable is given in the below table

VLC table numbers for mode table

ctxID	modeTableVLCTable
0	0,0,5,5,5,5,2
1	5,5,5,5,5,5,2
2	5,5,5,5,5,5,2

After the **mode_table_idx** syntax element is decoded, modeTableVLC_idx [ctxID] is updated according to the following:

If (modeTableVLCTable [ctxID] < **mode_table_idx**)
    modeTableVLC_idx [ctxID] = modeTableVLCTable [ctxID] + 1
else if (modeTableVLCTable [ctxID] > **mode_table_idx**)
    modeTableVLC_idx [ctxID] = modeTableVLCTable [ctxID] – 1
if(modeTableVLCTable [ctxID] < 0)
    modeTableVLCTable [ctxID] = 0
else if(modeTableVLCTable [ctxID] > 6)
    modeTableVLCTable [ctxID] = 6

7.1.2.2 Derivation of modeIdx

The inputs to this process are context identification information ctxID and the two dimensional context list array modeTableIdxCtxList.
The modeIdx variable is derived from the **mode_table_idx** syntax element as follows:

modeTableIdx = modeTableIdxCtxList [idx][mode_table_idx]

The entries of the modeTableIdxCtxList are updated as follows:

modeTableIdx2 = max(0, mode_table_idx-1);
modeIdx2 = modeTableIdxCtxList [idx][modeTableIdx2]
modeTableIdxCtxList [ctxID][modeTableIdx2] = modeTableIdx
modeTableIdxCtxList [ctxID][mode _table_idx] = modeTableIdx2

The initial values of the modeTableIdxCtxList are defined as

[Ed.: size of context list array to be further defined]

modeTableIdxCtxList [idx][n] = n; n=0..127, idx = 0..2

7.1.3 VLC Parsing process for coded block pattern

[Ed: Initial draft]

This process is invoked when entropy_coding_mode_flag is equal to 0.

The syntax element **cbp_table_idx** is decoded using sublcause XX. The variable cbpIdx is decoded using subclause XX.

7.1.3.1 Parsing process for the cbp_table_idx

Input to this process is variable array cbpTableVLC_idx.

The **cbp_table_idx** syntax element is decoded from the bitstream using VLC table given by cbpTableVLCTable[cbpTableVLC_idx[ctxID]]. The values of the VLC table array cbpTableVLCTable is given in the below table.

VLC table numbers for cbp

ctxID	modeTableVLCTable
0	10,10,10,2,2,2,2,7
1	10,10, 6,6,6,6,7,7
2	10,10, 6,6,6,6,7,7

After the **cbp_table_idx** syntax element is decoded, cbpTableVLC_idx[ctxID] is updated according to the following:

If (cbpTableVLCTable [ctxID] < **cbp_table_idx**)
    cbpTableVLCTable _idx [ctxID] = cbpTableVLCTable [ctxID] + 1
else if (cbpTableVLCTable [ctxID] > **cbp_table_idx**)
    cbpTableVLCTable _idx [ctxID] = cbpTableVLCTable [ctxID] – 1
if(cbpTableVLCTable [ctxID] < 0)
    cbpTableVLCTable [ctxID] = 0
else if(cbpTableVLCTable [ctxID] > 7)
    cbpTableVLCTable [ctxID] = 7

7.1.3.1 Derivation of modeIdx

The inputs to this process are context identification information ctxID and the two dimensional context list array cbpTableIdxCtxList.
The cbpIdx variable is derived from the **cbp_table_idx** syntax element as follows:

cbpTableIdx = cbpTableIdxCtxList [idx][cbp_table_idx]

The entries of the cbpTableIdxCtxList are updated as follows:

cbpTableIdx2 = max(0, cbp_table_idx-1);
cbpIdx2 = modeTableIdxCtxList [idx][modeTableIdx2]
cbpTableIdxCtxList [ctxID][modeTableIdx2] = cbpTableIdx
cbpTableIdxCtxList [ctxID][mode _table_idx] = cbpTableIdx2

The initial values of the cbpTableIdxCtxList are defined as

```
cbpTableIdxCtxList[0] =
{14,7,3,11,1,0,4,2,9,12,6,10,13,5,8,30,31,62,15,46,28,34,22,35,41,33,39,17,18,
20,36,47,16,26,44,23,25,43,29,27,19,32,42,38,56,45,50,52,24,57,60,49,61,54,51,
58,55,40,59,37,21,48,53}

cbpTableIdxCtxList[1] =
{14,11,2,12,4,6,10,9,13,1,3,5,8,0,7,15,31,46,30,62,38,42,44,20,45,36,18,22,34,
27,41,25,43,28,29,26,47,40,21,19,52,16,60,37,61,32,23,58,17,24,54,39,33,35,59,
57,50,53,56,51,49,48,55}
```

```
cbpTableIdxCtxList[2] =
{14,11,2,12,4,6,10,9,13,1,3,5,8,0,7,15,31,46,30,62,38,42,44,20,45,36,18,22,34,
27,41,25,43,28,29,26,47,40,21,19,52,16,60,37,61,32,23,58,17,24,54,39,33,35,59,
57,50,53,56,51,49,48,55}
```

7.2 VLC Parsing process for transform coefficients

[Ed.: more editorial work needed in this section]
This process is invoked when entropy_coding_mode_flag is equal to 0.

Inputs to this process are bits from slice data, the index of the current transform unit tuIdx, the size of the current transform unit tuSize.

Output of this process is the list coeffLevel containing transform coefficient levels of the transform block with index tuIdx.

Let W,H equal to width and height of the transform unit indicated with tuIdx. The variable numTotalCoeff specifies the total coefficients in the transform unit block and it is set to WxH. The variable numCodedCoeffs specifies the number of coded coefficients and it is equal to 16 if tuSize equals to TU_4x4, otherwise the variable numCodedCoeffs is set to 64.

Let codedCoeffLevel is a one dimensional array with size equal to numCodedCoeffs.

Following ordered steps are specified:

1. All transform coefficient levels, with indices from 0 to numTotalCoeff - 1, in the list coeffLevel are set equal to 0.
2. All transform coefficient levels, with indices from 0 to numCodedCoeffs - 1, in the list codedCoeffLevel are set equal to 0.
3. The context variable, ctxID is derived (subclause 7.2.1).
4. The position and the level of the last non-zero transform coefficient in zigzag scan order are derived and the list codedCoeffLevel is updated (subclause 7.2.2).
5. The combined run and level information are derived in run mode coding and the list codedCoeffLevel is updated (see subclause in 7.2.3).
6. The remaining coefficient levels are calculated and the list codedCoeffLevel is updated (see subclause in 0)
7. The coeffLevel array is updated using codedCoeffLevel array as follows (see subclause in 7.2.1).

If tuSize > TU_ 16x16 and PredMode == MODE_INTRA, a single 8x8 coefficient block uvCoeffLevel is decoded for both of the chroma components. The decoded uvCoeffLevel are deinterleaved to uCoeffLevel and vCoeffLevel as follows:

```
for(i = 0; i<32;i++) {
        uCoeffLevel[i] = uvCoeffLevel[i*2];
        vCoeffLevel[i] = uvCoeffLevel[i*2+1];
}

for(i = 32; i<64;i++) {
        uCoeffLevel[i] = 0;
        vCoeffLevel[i] = 0;
 }
```

7.2.1 Derivation of ctxID for transform coefficients

Input to this process are prediction unit specified by the prediction unit index puIdx, transform unit size specified by tuSize. Output of this process is the context identification variable ctxID.

The ctxID variable is given where for tuSize = TU_4x4:
ctxID = 0 for tuSize == TU_4x4 and PredMode == MODE_INTRA

ctxID = 1 for tuSize == TU_4x4 and PredMode == MODE_INTER and slice_type == P_SLICE
ctxID = 2 for tuSize == TU_4x4 and PredMode == MODE_INTER and slice_type == B_SLICE

for other values of tuSize:
ctxID = 0 for U and UV blocks
ctxID = 1 for V blocks
ctxID = 2 for tuSize == TU_8x8 and PredMode == MODE_INTRA
ctxID = 3 for tuSize == TU_8x8 and PredMode == MODE_INTER and slice_type == P_SLICE
ctxID = 4 for tuSize == TU_8x8 and PredMode == MODE_INTER and slice_type == B_SLICE
ctxID = 5 for tuSize == TU_16x16 and PredMode == MODE_INTRA
ctxID = 6 for tuSize == TU_16x16 and PredMode == MODE_INTER and slice_type == P_SLICE
ctxID = 7 for tuSize == TU_16x16 and PredMode == MODE_INTER and slice_type == B_SLICE
ctxID = 8 for tuSize > TU_16x16 and PredMode == MODE_INTRA
ctxID = 9 for tuSize > TU_16x16 and PredMode == MODE_INTER and slice_type == P_SLICE
ctxID = 10 for tuSize> TU_16x16 and PredMode == MODE_INTER and slice_type == B_SLICE

7.2.2 Parsing process for the level and position of last non-zero coefficient

Inputs to this process are bits from slice data.
Output of this process is the updated list codedCoeffLevel containing transform coefficient levels of the last non-zero coefficient in zig-zag scan within the transform block, and the variable lastPos, indicating the position of the last non-zero coefficient.

The syntax element **last_pos_table_idx** is decoded using subclause 7.2.2.1. The variable lastPosIdx is derived using subclause 7.2.2.2. The variables lastPos and levelMagnitudeGreaterThanOneFlag are calculated as follows:

– If the variable tuSize is equal to TU_4x4, the variable lastPos is set to lastPosIdx%16 and the variable levelMagnitudeGreaterThanOneFlag is set to lastPosIdx>>4
– Otherwise, the variable lastPos is set to lastPosIdx%64 and the variable levelMagnitudeGreaterThanOneFlag is set to lastPosIdx>>6

The magnitude of the last non-zero coefficient, lastPosMag is calculated as follows:

– If levelMagnitudeGreaterThanOneFlag is set to 1, the last_pos_level syntax element is decoded from the bitstream and lastPosMag = last_pos_level
– If levelMagnitudeGreaterThanOneFlag is set to 0, lastPosMag variable is set to 1
– last_pos_sign syntax element is decoded from the bitstream. If the value of last_pos_sign is equal to 1, lastPosMag is set to -1* lastPosMag

The codedCoeffLevel array is updated as codedCoeffLevel[lastPos] = lastPosMag. The currPos variable is updated as currPos = lastPos

7.2.2.1 Parsing process for the position of last non-zero coefficient

Input to this process is context variable array lastPosVLC_idx.
The **last_pos_table_idx** syntax element is decoded from the bitstream using VLC table given by lpVLCTable[lastPosVLC_idx[ctxID]]. The values of VLC table array lpVLCTable is given in the below table

VLC table numbers for last non-zero coefficient

ctxID	lpVLCTable
0	[10,10,10,10, 2,2,2,7,9,9,9,9,9,4,4,4,4]
1	[10,10,10,10,10,2,9,9,9,9,9,9,9,4,4,4,4]
2	[2, 2, 2, 2, 2,7,7,7,7,7,7,7,7,7,4,4,4]
3	[2, 2, 2, 2, 7,7,7,7,7,7,7,7,7,7,7,7,4]
4	[2, 2, 2, 2, 7,7,7,7,7,7,7,7,7,7,7,7,4]
5	[10, 1, 2, 2, 2,2,7,7,7,7,9,9,9,4,4,4,4]
6	[10,10, 2, 2, 7,7,7,7,7,7,7,7,4,4,4,4,4]
7	[10,10, 2, 2, 7,7,7,7,7,7,7,7,4,4,4,4,4]
8	[2, 2, 2, 2, 7,7,7,7,7,7,7,7,7,7,7,7,4]
9	[2, 2, 2, 2, 7,7,7,7,7,7,7,7,7,7,7,7,4]

After the **last_pos_table_idx** syntax element is decoded, lastPosVLC_idx[ctxID] is updated according to the following:

If (lpVLCTable [ctxID] < **last_pos_table_idx**)

```
            lastPosVLC_idx[ ctxID ] = lpVLCTable [ ctxID ] + 1
    else if (lpVLCTable [ ctxID ] > last_pos_table_idx)
            lastPosVLC_idx[ ctxID ] = lpVLCTable [ctxID] – 1
    if( lpVLCTable[ ctxID ] < 0)
             lpVLCTable[ ctxID ] = 0
    else if( lpVLCTable[ ctxID ] > 16 )
            lpVLCTable[ ctxID ] = 16
```

7.2.2.2 Derivation of lastPosIdx

The input to this process are context identification information ctxID and the two dimensional context list array lastPosCtxList.
The lastPosIdx variable is derived from the **last_pos_table_idx** syntax element as follows:

```
lastPosIdx = lastPosCtxList[idx][ last_pos _table_idx]
```

The entries of the lastPosCtxList are updated as follows:

```
lastPosTableIdx2  =  max(0, last_pos _table_idx-1);
lastPosIdx2 = lastPosCtxList [idx][ lastPosTableIdx2]
lastPosCtxList[ctxID][ lastPosTableIdx2] = lastPosIdx
lastPosCtxList [ctxID][ last_pos _table_idx] = lastPosIdx2
```

Initial values of the lastPosCtxList are defined as:

for tuSize = TU_4x4:
    lastPosCtxList[idx][n] = n; n=0,31
for other tuSize values:

For idx = 5,6,7: lastPosCtxList[idx][n] = n; n=0,127

For idx = 0,1:
lastPosCtxList =
{0,2,1,4,7,3,6,11,13,9,18,22,19,17,8,10,20,24,28,27,21,26,38,42,37,31,23,12,16,33,30,44,45,49,46,36,57,50,55,47,39,43,35,75,66,56,60,58,54,70,63,29,77,87,69,48,51,64,71,65,80,85,76,94,5,15,14,34,52,25,41,72,74,53,59,81,82,68,40,62,95,127,89,84,67,73,126,79,125,93,90,61,32,124,123,83,86,122,88,78,121,92,91,120,119,118,117,116,115,114,113,112,111,110,109,108,107,106,105,104,103,102,101,100,99,98,97,96}

For idx=2:
lastPosCtxList =
{0,1,2,9,14,3,5,17,15,6,8,16,23,18,4,11,24,27,25,20,10,26,50,40,34,31,21,12,13,28,33,32,35,43,57,46,76,65,55,45,38,36,22,39,47,49,53,60,75,95,70,61,59,51,64,63,68,101,111,69,67,74,99,97,7,30,19,42,66,41,54,83,72,29,37,71,81,86,52,58,79,85,84,77,48,62,104,92,89,80,78,56,44,88,93,87,94,96,105,82,127,114,110,109,100,91,73,90,106,103,126,118,125,124,113,117,107,98,112,116,115,123,122,102,108,121,120,119}

For idx=3,4,8,9:
lastPosCtxList =
{0,3,1,4,6,5,12,16,7,2,10,14,15,17,8,13,22,21,18,20,19,37,35,30,26,24,23,11,9,27,29,28,31,38,45,57,71,50,41,36,34,32,25,33,39,40,44,55,79,90,62,46,49,42,54,53,63,87,94,60,64,74,86,95,43,58,52,51,65,66,67,76,59,47,56,61,73,81,69,72,88,84,78,75,70,77,98,85,83,82,92,68,48,91,97,93,100,102,106,112,127,110,107,104,96,101,80,89,103,109,117,113,126,125,124,116,105,99,108,115,114,123,122,121,111,120,119,118}

7.2.3 Parsing process for run mode decoding

Inputs to this process are bits from slice data, current coefficient position currPos, number of coded coefficients numCodedCoeff.
Output of this process is the updated list codedCoeffLevel.

While the variable runMode is equal to 1 and variable pos < numCodedCoeff, the codedCoeffLevel array is updated as follows:

– Variable vlcIdx is determined as follows:

```
if( (numCodedCoeff - 1) - i > 27 )
    vlcIdx = 3
else
    vlcIdx = runsVLCTable[ 63 - (maxNumCoeff-1)]
```

where runsVLCTable is a one dimensional array with values given as:

runsVLCTable = {8,0,0,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,7,7,7...}

– The syntax element is_level_one_run is decoded and variables run and isLevelOne are calculated using the below table.

Mapping from k, level, and run into codeNumber

		isLevelOne_run
currPos	isLevelOne	run = 0,1...
0	1	1,0
	0	2
1	1	2,1,0
	0	4,3
2	1	1,3,2,0
	0	4,6,5
3	1	2,1,3,4,0
	0	6,5,7,8
4	1	1,5,3,2,4,0
	0	6,10,8,7,9
5	1	1,2,6,5,3,4,0
	0	7,8,11,10,9,12
6	1	2,1,3,5,4,7,6,0
	0	9,8,10,12,11,13,14
7	1	1,5,4,2,3,6,8,7,0
	0	9,13,12,10,11,14,15,16
8	1	1,3,8,7,5,2,4,9,6,0
	0	10,12,16,15,14,11,13,18,17
9	1	1,2,5,10,9,7,3,4,11,6,0
	0	8,12,15,17,18,16,13,14,20,19
10	1	2,1,3,4,7,8,5,6,9,11,10,0
	0	13,12,14,15,17,18,16,19,20,22,21
11	1	1,4,3,2,5,7,6,8,10,11,12,9,0
	0	13,17,15,14,16,19,18,20,22,23,24,21
12	1	1,2,6,7,5,3,4,8,9,13,11,12,10,0
	0	14,16,18,21,19,15,17,20,22,26,23,24,25
13	1	1,2,4,8,9,7,6,3,5,12,14,13,11,10,0
	0	15,16,19,21,23,22,20,17,18,26,28,27,24,25
14	1	1,2,3,5,9,10,8,7,4,6,13,16,15,14,12,0
	0	11,17,18,20,22,24,23,26,19,21,27,30,29,25,28
15	1	1,2,3,4,5,8,10,7,6,9,11,13,18,15,16,12,0
	0	14,17,19,20,21,23,25,24,22,26,27,29,31,28,32,30
16	1	1,3,4,2,5,6,10,9,7,8,11,12,13,16,17,18,14,0
	0	15,19,22,20,21,24,26,25,23,27,32,29,28,30,33,34,31
17	1	1,2,4,7,6,3,5,8,9,10,11,18,15,13,14,17,19,12,0
	0	16,21,24,25,26,20,22,27,23,28,30,35,31,29,34,33,36,32
18	1	1,2,5,7,9,10,6,3,4,8,11,17,20,18,15,13,14,19,12,0
	0	16,21,24,25,29,28,27,23,22,26,31,36,34,37,33,32,30,38,35
19	1	1,3,5,7,9,14,12,8,6,2,4,15,18,21,20,16,10,11,22,13,0

	0	17,23,25,27,29,31,33,30,26,19,24,35,37,39,40,36,32,28,38,34
20	1	0,2,4,7,10,9,16,15,14,8,3,5,17,19,22,20,18,11,13,21,12
	0	6,23,25,29,27,31,30,34,38,32,24,26,36,82,39,37,83,28,33,84,35
21	1	2,1,3,4,5,8,9,11,13,10,6,7,12,14,21,24,23,18,15,19,22,16,0
	0	20,17,25,26,27,28,31,34,35,30,29,32,33,36,37,43,44,38,39,40,41,42
22	1	1,5,4,2,3,6,8,7,11,12,9,10,13,14,15,18,22,21,17,19,20,25,16,0
	0	26,28,27,23,24,29,31,32,38,34,30,33,35,37,36,43,44,40,41,85,42,86,39
23	1	1,4,8,6,5,2,3,7,9,12,10,11,13,16,15,14,18,20,21,19,23,22,25,17,0
	0	24,28,33,29,31,26,27,34,35,36,32,30,37,40,38,42,45,47,39,43,46,41,87,44
24	1	0,3,7,12,10,6,5,2,4,8,9,11,13,17,24,21,16,15,19,20,22,25,26,23,18,1
	0	14,28,34,36,33,35,31,27,30,29,32,37,40,46,38,44,39,47,48,49,45,41,42,50,43
25	1	0,2,5,9,13,11,10,8,6,3,4,7,12,15,19,26,23,20,16,14,18,22,27,24,25,21,1
	0	17,28,31,36,35,39,37,34,33,30,29,32,38,43,86,87,45,46,44,41,40,88,48,89,47,42
26	1	1,2,4,7,11,15,12,9,10,8,5,3,6,14,16,23,24,25,21,20,13,19,27,28,26,22,17,0
	0	18,29,31,33,39,36,35,38,41,40,34,30,32,43,48,53,90,49,42,50,37,44,51,47,52,45,46
27	1	0,2,3,5,7,11,15,14,12,10,13,8,4,9,17,19,22,24,26,21,23,16,20,29,30,28,25,18,1
	0	6,27,31,33,34,38,39,37,40,42,41,35,32,36,44,45,87,48,88,49,46,89,43,47,90,50,51,91
>=28	1	0,1,2,3,4,5,7,9,10,13,8,12,11,6,14,15,18,20,24,31,32,30,28,26,22,25,33,37,38,17,19,21,23,29,36,34,45,40, 43,42,44,39,35,46,48,53,54,51,61,62,67,66,68,65,73,74,71,69,86,82,90,88,98,94
	0	16,27,41,47,50,57,58,60,59,55,52,63,56,49,64,70,72,77,76,80,84,81,79,78,75,83,85,87,91,95,92,104,89,93,96,99,100,102,101,97,105,103,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126

- The magnitude of non-zero coefficient is given as level and is calculated as follows:
  - If the variable isLevelOne is equal to 1, the level_sign is decoded from the bitstream. If the value of level_sign is equal to 1, the variable level is set to -1. If the value of level_sign is equal to 0, the variable level is set to 1.
  - Otherwise, if the variable isLevelOne is equal to 0, the level_magnitude_minus2 and level_sign syntax elements are decoded from the bitstream. If the value of level_sign is equal to 0, the value of level is set to level_magnitude_minus2 + 2. Otherwise, the value of level is set to -1*(level_magnitude_minus2+2)
- The codedCoeffLevel array is updated as codedCoeffLevel [currPos] = level
- The variable currPos is updated as currPos = currPos - run
- The variable runMode is calculated as follows:

```
runMode = 1
if (tuSize == TU_4x4 || PredMode == MODE_INTRA)
    runMode == (isLevelOne)
else if(currPos < 14 || sumBigCoef > 2)
    runMode = 0
```

where the variable sumBigCoef is calculated as:

```
for(i = currPos; i < 64; i++) {
    sumBigCoef += ( Abs(codedCoeffLevel [ i ] ) > 1 ? Abs(codedCoeffLevel [ i ] ) : 0 );
```

7.2.4 Parsing process for level mode decoding

Inputs to this process are bits from slice data, a maximum number of non-zero transform coefficient levels, maxCoeff, current coefficient position currPos.

Output of this process is the updated list codedCoeffLevel.

The variable vlcMagIdx is initialized to 0.

While the variable currPos is larger than -1, the codedCoeffLevel array is updated as follows:

- The level_magnitude syntax element is decoded from the bitstream
- The codedCoeffLevel [currPos] is updated as follows:
  - If the level_magnitude is not equal to 0, the level_sign is decoded from the bitstream. If the value of level_sign is equal to 1, the variable codedCoeffLevel [currPos] is set to -1*level_magnitude. If the value of level_sign is equal to 0, the variable codedCoeffLevel [0] is set to level_magnitude.
- The variable currPos is updated as currPos - 1.
- The variable vlcMagIdx is updated as follows:

```
if (level_magnitude > vlc_level_table [ vlcMagIdx ] )
    vlcMagIdx++;
```

The one dimensional vlc_level_table array is given as

VLC Table Index for Decoding Level

x	vlc_level_table[x]
0	4
1	6
2	14
3	28
4	∞

The codedCoeffLevel [0] is updates as follows:

– If the PredMode == MODE_INTRA and none of the prediction samples are available for prediction (subclause 6.2.1.2) the level_magnitude syntax element is decoded from the bitstream using the VLC table given with the variable dcVLC_id. The variable dcVLC_id is equal to 3 if the decoded block is a luminance block, otherwise dcVLC_id is equal to 1.
– If the level_magnitude is not equal to 0, the level_sign is decoded from the bitstream. If the value of level_sign is equal to 1, the variable codedCoeffLevel [0] is set to -1*level_magnitude. If the value of level_sign is equal to 0, the variable codedCoeffLevel [0] is set to level_magnitude.

7.2.1 Update process for coeffLevel array

Inputs to this process are one dimensional coded coefficient levels array codedCoeffLevel, one dimensional transform coefficient levels array coeffLevel, number of total coefficients numTotalCoeff and number of total coded coefficients, numCodedCoeffs.

If the tuSize is larger than TU_8x8 the following ordered steps are specified

1. The inverse scanning process for the transform coefficients as described specified in subclause 6.4.3 is invoked with codedCoeffLevel as the input and the two-dimensional array ccl as the output.
2. The inverse scanning process for the transform coefficients as described specified in subclause 6.4.3 is invoked with coeffLevel as the input and the two-dimensional array cl as the output.
3. The two dimensional array cl is updated as follows
   cl[i, j] = ccl[i,j], with i = 0..7, j = 0..7
4. The forward scanning process for the transform coefficients as described in subclause 7.2.2 is invoked with cl as input and coeffLevel as output

Otherwise, if tuSize is smaller than or equal to TU_4x4 the coefficient level array is updated as follows

coeffLevel[i] = codedCoeffLevel[i] , with i = 0.. numCodedCoeffs-1

7.2.2 Forward scanning process for transform coefficients

Input to this process is a list of N values listTrCoeffs.

Output of this process is a variable c containing a one dimensional array of N values.

The variable c_{ij} which is located in the (j, i) position in the array c is derived as follows.

c_{ij} = BlocktoZigzagPosition(listTrCoeffs, N, idx).

7.3 VLC Tables

n = 0	n = 1	n = 2	n = 3	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9	n = 10
1	1x	1xx	1xxx	1xxxx	1x	1xx	1xxx	1	100	1
01	01x	01xx	01xxx	01xxxx	01x	01xx	01xxx	01	1010	01x
001	001x	001xx	001xxx	001xxxx	001x	001xx	001xxx	00	1011	001xx

0001	0001x	0001xx	0001xxx	0001xxxx	..	..	..		11xxx	0001xxx
00001	00001x	00001xx	00001xxx	00001xxxx					01xxxx	..
0000001	0000001x	0000001xx	0000001xxx	0000001xxxx					001xxxx	
00000001	00000001x	00000001xx	00000001xxx	00000001xxxx					..	
000000001x	000000001xx	000000001xxx	000000001xxxx	000000001xxxxx						
0000000001xx	0000000001xxx	0000000001xxxx	0000000001xxxxx	0000000001xxxxxx						
..	..	..	..	..						

The tables list the codes for code number 0,1,2,3..etc. However, for a more compact description most of the entries in the table ends with a number of "x" -es. As an example 1xx in VLC table 2 represents 4 codes with code numbers 0-3:

100

101

110

111

Generally the code numbers increase in the same way as the natural number representation of the codes increase. The symbol ".." in the tables indicate that the tables continue in the obvious way.

7.4 Parsing process for Exp-Golomb codes of order k

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ge(v).
Inputs to this process are bits from the RBSP.
Outputs of this process are syntax element values.
Syntax elements coded as ge(v) are Exp-Golomb-coded of order k. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, counting k bits, and sign bit if available. This process is specified as follows:

```
q = -1, nr = 0, m = pow(2.0, k)
uiSymbol = 1;
while (uiSymbol)
{
   uiSymbol    = read_bits (uiSymbol)
   q++;
}
for(a = 0; a < k; ++a)
{
   uiSymbol    = read_bits (uiSymbol)
   if(uiSymbol) nr += 1 << a
}
nr += q * m
if (nr != 0){
   uiSymbol    = read_bits (uiSymbol)
   codeNum = (uiSymbol)? nr: -nr;
}
```

7.5 PIPE parsing process

[Ed. Need more details.]

The parsing process specified in this subclause is used when entropy_coding_mode_flag is equal to 1.

The PIPE entropy coding consists of three stages:

– binarization,

– context modelling,

– binary entropy coding.

The stages of binarization and context modelling are the same as in H.264/AVC and are specified in clause 9 of ITU-T Rec. H.264 | ISO/IEC 14496-10. [Ed. Need to specify binarization and context determination for all syntax elements]

The decoding of the binary decisions (bins) proceeds as follows:

– If pipe_multi_codeword_flag is equal to 1 and balanced_cpus is equal to 1, the multi-partition parsing process as specified in subclause 7.5.1 is invoked.

– Otherwise, if pipe-multi_codeword_flag is equal to 1 and balanced_cpus >1, the multi-partition parsing process with load balancing as specified in subclause 7.5.2 is invoked.

– Otherwise, if pipe_max_delay_in_bits_shift_6 is equal to 0, the interleaved parsing process for binary decisions as specified in subclause ~~7.5.2~~7.5.3 is invoked.

– Otherwise, the interleaved parsing process for binary decisions with a delay constraint as specified in subclause 7.5.4~~7.5.3~~7.5.4 is invoked.

The decoded syntax elements are determined based on the binarization schemes and the decoded binary decisions as specified in clause 9 of ITU-T Rec. H.264 | ISO/IEC 14496-10. [Ed. Need more details.]

The PIPE parsing process for transform coefficient levels is specified in subclause 7.6.

7.5.1 Multi-partition parsing process for binary decisions

The V2V codewords for each probability partition, given by probPartIdx, are coded in separate NAL unit partitions.

The first bit for each partition, with partition index probPartIxd, is given by the variable pipe_partition_start[probPartIdx] as specified in subclause 5.1.12 and the size of each partition is given by the variable pipe_partition_size[probPartIdx] as specified in subclause 5.1.12.

The binary decisions are decoded as follows.

Let binDecisions[probPartIdx] be 12 bin buffers, with probPartIdx 0..11, inclusive, which are empty at the beginning of the parsing process for a slices.

With probPartIdx proceeding over 0..11, inclusive, the following applies.

– The variable bitCounter is set equal to pipe_partition_start[probPartIdx].

– The variable numBitsRead is set equal to 0.

– While numBitsRead is less than pipe_partition_size[probPartIdx], the following applies.

  – The next codeword is read from the bitstream at position bitCounter. The ~~VLC~~ V2V table that is used is specified in subclause 7.7 given the value of probPartIdx.

  – The variable bitCounter is incremented by the number of bits in the read codeword.

  – The codeword is converted into a bin sequence as specified in the V2V table of subclause 7.7 that is given the value of probPartIdx.

  – The bin sequence is inserted at the end of the bin buffer binDecisions[probPartIdx]

A requested binary decision (given by the binarization scheme and the context modelling) with a context (pStateIdx, pMPSval) is decoded as follows:

– The variable probPartIdx is determined as specified in Table 7-1~~Table 7-1~~~~Table 7-1~~ given the value of pStateIdx.

– The first bin pBin is removed from the bit buffer binDecisions[probPartIdx].

– The value of the decoded bin is set equal to pBin ^ pMPSval, where "^" specifies the xor operator.

7.5.2 Multi-partition parsing process with load balancing for binary decision

To make use of multiple bin decoders for parallel entropy decoding, the starting posititions balancedPartStart[blancedPartIdx], balancedPartIdx=0, 1, …, balanced_cpus-1, of individually decodeable partial bitstreams of the encoded bitstream is first determined as follows.

Step 1: Calculate uiNumOverallBits = variable pipe_partition_size[0]+ variable pipe_partition_size[1]+…+ variable pipe_partition_size[11];
Step 2: Set balancedPartStart[0] = 0. The following loop determines the remaining starting positions and the size balancedPartSize[balancedPartIdx] of each partial bitstream.

```
for (balancedPartIdx =0; balancedPartIdx < balanced_cpus-1; balancedPartIdx++)
{
        balancedPartSize[balancedPartIdx] = uiNumOverallBits/balanced_cpus;
        balancedPartSize[balancedPartIdx] += balanced_offsets[balancedPartIdx];
        balancedPartStart[balancedPartIdx+1] = balancedPartStart[balancedPartIdx]+balancedPartSize[balancedPartIdx];
}
balancedPartSize[balancedPartIdx] = unNumOverallBits – balancedPartStart[balancedPartIdx];
```

In parallel to the decoding of the other partial bitstreams, the partial bitstream starting at balancedPartStart[balancedPartIdx], where 0 ≤ balancedPartIdx < balanced_cpus can be decoded follows.

Let partBinDecisions[balancedPartIdx] be 12 bin buffers which are empty at the beginning of the parsing process for a slice. For probPartIdx=0,..., 11, do the following.

- If pipe_partition_start[probPartIdx] >= balancedPartStart[balancedPartIdx]+ balancedPartSize[balancedPartIdx], stop.
- Otherwise, if pipe_partition_start[probPartIdx] < balancedPartStart[balancedPartIdx]- pipe_partition_size[probPartIdx], continue to the next probPartIdx.
- Otherwise, do the following
  - set the variable bitCounter to max(pipe_partition_start[probPartIdx], balancedPartStart[balancedPartIdx]).
  - set the variable numBitsRead to bitCounter- pipe_partition_start[probPartIdx].
  - While numBitsRead is less than pipe_partition_size [probPartIdx], the following applies.
    - The next codeword is read from the bitstream at position bitCounter. The V2V table that is used is specified in subclause 7.7 given the value of probPartIdx.
    - The variable bitCounter is incremented by the number of bits in the read codeword. The codeword is converted into a bin sequence as specified in the V2V table of subclause 7.7 that is given the value of probPartIdx.
    - The bin sequence is inserted at the end of the bin buffer partBinDecisions[balancedPartIdx][probPartIdx].

After all partBinDecisions are decoded, they are concatenated to form binDecisions as follows.
For probPartIdx=0, …, 11, do the following.

- Initialize binDecisions[probPartIdx] as empty.
- For balancedPartIdx=0,…,balanced_cpus, do the following.
  - Insert partBinDecisions[balancedPartIdx][probPartIdx] at the end of binDecisions[probPartIdx].

With binDecisions, a requested binary decision (given by the binarization scheme and the context modelling) with a context (pStateIdx, pMPSval) is decoded as described at the end of subclause 7.5.1 above.

~~7.5.2~~7.5.3 Interleaved parsing process for binary decisions

The V2V codewords for the 12 probability partitions are interleaved in a single NAL unit partition.

Let binBuffer[probPartIdx] be 12 bin buffers, with probPartIdx 0..11, inclusive, which are empty at the beginning of the parsing process for a slices.

A requested binary decision (given by the binarization scheme and the context modelling) with a context (pStateIdx, pMPSval) is decoded as follows:

- The variable probPartIdx is determined as specified in Table 7-1~~Table 7-1Table 7-1~~ given the value of pStateIdx.
- When the binBuffer[probPartIdx] is empty, the following applies.
  - The next codeword is read from the bitstream. The ~~VLC~~ V2V table that is used is specified in subclause 7.7 given the value of probPartIdx.
  - The codeword is converted into a bin sequence as specified in the V2V table of subclause 7.7 that is given the value of probPartIdx.
  - The bin sequence is inserted in the bin buffer binBuffer[probPartIdx].
- The first bin pBin is removed from the bit buffer binBuffer[probPartIdx].
- The value of the decoded bin is set equal to pBin ^ pMPSval, where "^" specifies the xor operator.

~~7.5.3~~7.5.4 Interleaved parsing process for binary decisions with a delay constraint

The V2V codewords for the 12 probability partitions are interleaved in a single NAL unit partition. In addition, the decoding process takes into account a delay constraint. The maximum delay in bits is specified by the syntax element pipe_max_delay_in_bits_shift_6.

Let binBuffer[probPartIdx] be 12 bin buffers, with probPartIdx 0..11, inclusive, which are empty at the beginning of the parsing process for a slices.

Let maxDelayCountSum be a variable that is initially set equal to 0. Let maxDelayCount[probPartIdx] be 12 variables, with probPartIdx 0..11, inclusive, that are initially set equal to 0.

A requested binary decision (given by the binarization scheme and the context modelling) with a context (pStateIdx, pMPSval) is decoded as follows:

- The variable probPartIdx is determined as specified in Table 7-1~~Table 7-1Table 7-1~~ given the value of pStateIdx.
- When maxDelayCountSum − maxDelayCount[probPartIdx] is greater than pipe_max_delay_in_bits_shift_6 * 32, the bin buffer binBuffer[probPartIdx] is emptied (i.e., all bins are removed).
- When the binBuffer[probPartIdx] is empty, the following applies.
  - The next codeword is read from the bitstream. The ~~VLC~~ V2V table that is used is specified in subclause 7.7 given the value of probPartIdx.
  - The codeword is converted into a bin sequence as specified in the V2V table of subclause 7.7 that is given the value of probPartIdx.
  - The bin sequence is inserted in the bin buffer binBuffer[probPartIdx].
  - Let maxNumBits the number of bits of the longest codeword in the codeword tables given by pStateIdx as specified in subclause 7.7.
  - The counter maxDelayCountSum is increased by maxNumBits.
  - The counter maxDelayCount[probPartIdx] is set equal to maxDelayCountSum.
- The first bin pBin is removed from the bit buffer binBuffer[probPartIdx].
- The value of the decoded bin is set equal to pBin ^ pMPSval, where "^" specifies the xor operator.

7.6 PIPE Parsing process for transform coefficients

[Ed. Needs better specification. The current specification is unsuitable for a standard.]

Transform coefficient levels are parsed in two steps like in CABAC of H.264/AVC. In the first step, the significance map specifies the locations of significant coefficient levels are parsed. The second step parsed the absolute coefficient levels (minus one) and the sign for each significant coefficient position.

7.6.1 Parsing process for significant coefficient positions

The parsing process of significance map based on an adaptive scan and context modeling depends on already parsed neighbours. The scanning order for coding the significance map is adapted by switching between two predefined scan patterns as illustrated in Fig. 2. For the first scanning pattern, the diagonal sub-scans are scanned from bottom-left to top-right (left illustration of Fig. 2), and for the second scanning pattern, the diagonal sub-scans are scanned from top-right to bottom-left (right illustration of Fig. 2).

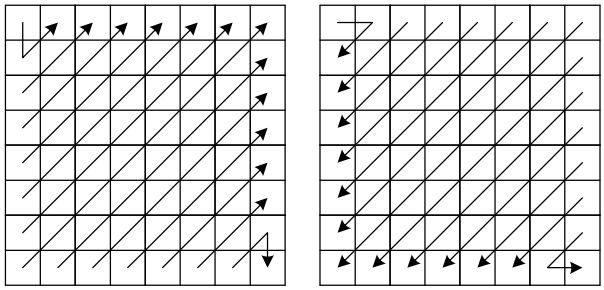


Figure 2: Scanning patterns for significance map

The parsing of the significance map starts with the second scanning pattern. While parsing the syntax elements, the number of significant transform coefficient values is counted by two counters c_1 and c_2. The first counter c_1 counts the number of significant transform coefficients that are located in the bottom-left part of the transform block; i.e., this counter is incremented by one when a significant transform coefficient level is parsed for which the horizontal coordinate x inside the transform block is less than the vertical coordinate y. The second counter c_2 counts the number of significant transform coefficients that are located in the top-right part of the transform block; i.e., this counter is incremented by one when a significant transform coefficient level is parsed for which the horizontal coordinate x inside the transform block is greater than the vertical coordinate y. At the end of each diagonal sub-scan, it is decided whether the first or the second of the predefined scanning patterns is used for the next diagonal sub-scan. This decision is based on the values of the counters c_1 and c_2. When the counter for the bottom-left part of the transform block is greater than the counter for the bottom-left part, the scanning pattern that scans the diagonal sub-scans from bottom-left to top-right is used; otherwise (the counter for the bottom-left part of the transform block is less than or equal to the counter for the bottom-left part), the scanning pattern that scans the diagonal sub-scans from topright to bottom-left is used.

For 4x4 blocks, the context modeling for the syntax elements significant_coeff_flag is done as specified in AVC. For 8x8 blocks, the transform block is decomposed into 16 sub-blocks of 2x2 samples, and each of these sub-blocks is associated with a separate context. The context model selection for larger transform blocks (e.g., for blocks greater than 8x8) is based on the number of already parsed significant transform coefficients in a predefined neighborhood (inside the transform block). Following evaluation is made for this case, where ctxIdx specifies the context model index for the syntax element significant_coeff_flag, and ctxInc is a temporal counter init by zero:

- Scan position lies inside first 2x2 sub-block
  - ctxIdx = (posY << 1) + posX
- Scan position lies on the top border
  - ctxInc += significant_coeff_flag[x-1, y] ? 1 : 0

- ctxInc += significant_coeff_flag[x-2, y] ? 1 : 0
- ctxInc += significant_coeff_flag[x-2,y+1] ? 1 : 0
- if selected scanPattern=0 (fist scan pattern)
  - ctxInc += significant_coeff_flag[x-1,y+1] ? 1 : 0
- ctxIdx = 4 + ((ctxInc + 1) >> 1)

- Scan position lies on the left border
  - ctxInc += significant_coeff_flag[x, y-1] ? 1 : 0
  - ctxInc += significant_coeff_flag[x, y-2] ? 1 : 0
  - ctxInc += significant_coeff_flag[x+1, y-2] ? 1 : 0
  - if selected scanPattern=1
    - ctxInc += significant_coeff_flag[x+1, y-1] ? 1 : 0
  - uiIdx = 7 + ((ctxInc + 1) >> 1)
- Scan position lies inside the transform block
  - ctxInc += significant_coeff_flag[x, y-1] ? 1 : 0
  - ctxInc += significant_coeff_flag[x-1, y] ? 1 : 0
  - ctxInc += significant_coeff_flag[x-1, y-1] ? 1 : 0
  - if x > 1
    - ctxInc += significant_coeff_flag[x-2, y] ? 1 : 0
    - ctxInc += significant_coeff_flag[x-2, y-1] ? 1 : 0
    - if y < tuSize – 1
      - ctxInc += significant_coeff_flag[x-2, y+1] ? 1 : 0
  - if y > 1
    - ctxInc += significant_coeff_flag[x, y-2] ? 1 : 0
    - ctxInc += significant_coeff_flag[x-1, y-2] ? 1 : 0
    - if x < tuSize – 1
      - ctxInc += significant_coeff_flag[x+1, y-2] ? 1 : 0
    - if scanPattern = 1
      - if x < tuSize – 1
        - ctxInc += significant_coeff_flag[x+1, y-1] ? 1 : 0
    - else
      - if y < tuSize – 1
        - ctxInc += significant_coeff_flag[x-1, y+1] ? 1 : 0
  - ctxIdx = 10 + min(4, (ctxInc + 1) >> 1)

For coding of the last_significant_coeff_flag, the context modeling has been designed that depends on a distance measure of the current scan position to the top-left corner of the given transform block. The context model for coding the last_significant_coeff_flag is chosen based on the scan diagonal on which the current scan position lies. The distance measure x + y is mapped on a reduced set of context models. Given x, y and the block size, the ctxIdx is derived by following formula:

- Blocksize is 4x4
  - Same context selection like in AVC CABAC
- Blocksize is larger than 4x4
  - ctxIdx = (x + y) >> (Log2(tuSize) - 3)

7.6.2 Parsing process for the absolute level and sign

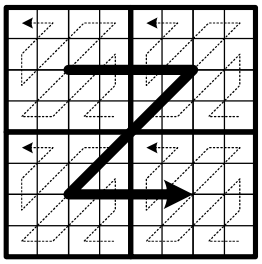


Figure 3: Scan pattern for coefficient levels

For 4x4 blocks, the context modeling process for coefficient levels is done as specified in AVC. For blocks larger than 4x4, the coding process for absolute transform coefficient levels maps each quadratic (or rectangular) block onto an ordered set (vector) of 4x4 sub-blocks by using a forward zig-zag scan; while the transform coefficient levels inside a sub-block are processed in a reverse zig-zag scan as illustrated in Fig. 3. The handling of context model selection for 4x4 blocks is the same like in AVC CABAC. In AVC CABAC, five context models are used for the syntax element coeff_abs_greater_one and another five context models for the syntax element coeff_abs_level_m2. The syntax element coeff_level_sign is bypassed. Following, the context selection of AVC CABAC for a 4x4 is summarized:

- coeff_abs_greater_one
  - if first scan position
    - ctxIdx = 1
  - if previous coeff_abs_greater_one = 0
    - if ctxIdx > 0
      - ctxIdx = min(ctxIdx++, 4)
  - else
    - ctxIdx = 0
- coeff_abs_level_m2
  - ctxIdx = 0
  - if coeff_abs_greater_one = 1
    - ctxIdx++

The number context models is extended resulting in 30 context models for coeff_abs_greater_one and another 30 context models for coeff_abs_level_m2. An offset is selected for each sub-blocks, while the context model selection inside a sub-block is the same like 4x4 blocks in AVC CABAC resulting in a limitation of context selection to [offset, offset+5]. The choice of the offset for both syntax elements for a sub-block depends on the number of absolute coefficient levels greater than one in the previous sub-block, where empty sub-blocks are skipped. For the first sub-block, the the offset is 20. For 4x4 blocks, the offset is zero. For all other sub-blocks, following offset mapping is used, where the index of the array is the number of coefficient levels greater than one in the previous sub-block (where empty sub-blocks are skipped):

- offset = [5, 5, 5, 5, 10, 10, 10, 10, 15, 15, 15, 15, 20, 20, 20, 20, 25]

7.7 PIPE tables

Table 7-1~~Table 7-1Table 7-1~~ specifies that mapping of a context state (given by a variable pStateIdx) to a probability partition index probPartIdx.

Table 7-2~~Table 7-2Table 7-2~~ to Table 7-13~~Table 7-13Table 7-13~~ specify the V2V codes for the probability partition indices 0 to 11, inclusive. The tables show a mapping from a bin sequence with a variable number of bins to variable-length codewords and vice versa.

Table 7-1 Mapping of the variable pStateIdx to the variable probPartIdx

probPartIdx	pStateIdx
0	0..1
1	2
2	3..4
3	5..6
4	7..9
5	10..12
6	13..16
7	17..20
8	21..28
9	29..39

10	40..55
11	56..63

Table 7-2 V2V mapping for probPartIdx = 0

bin sequence	codeword
0	0
1	1

Table 7-3 V2V mapping for probPartIdx = 1

bin sequence	codeword
10	10
01	11
001	010
110	011
000 11	001 01
000 000	001 10
111 00	001 11
000 011	000 100
000 101	000 101
111 11	000 110
000 010 0	000 111
000 001 0	001 000
000 100 0	001 001
000 010 1	000 000 1
000 100 1	000 001 0
000 001 1	000 001 1
111 011	000 010 0
111 101	000 010 1
111 010 0	000 011 0
111 100 0	000 011 1
111 010 1	000 000 00
111 100 1	000 000 01

Table 7-4 V2V mapping for probPartIdx = 2

bin sequence	codeword
10	10
01	11
001	011
000 00	010 0
110 0	010 1
000 11	000 11
111 1	001 00
000 100	001 01
000 010	001 10
111 00	001 11
000 101	000 010
110 11	000 011
111 01	000 100
110 100	000 101
000 011 1	000 000 1
000 011 00	000 001 0
110 101 0	000 001 1
000 011 01	000 000 00
110 101 1	000 000 01

Table 7-5 V2V mapping for probPartIdx = 3

bin sequence	codeword
10	10

01	11
001	010
000 0	011
111	001 0
110 0	001 1
000 11	000 10
000 100	000 11
110 11	000 010
110 100	000 011
000 101 1	000 001 0
000 101 00	000 001 1
110 101 1	000 000 10
110 101 00	000 000 11
000 101 011	000 000 010
000 101 010 0	000 000 011
110 101 011	000 000 000 1
110 101 010 0	000 000 001 0
000 101 010 10	000 000 001 1
110 101 010 1	000 000 000 00
000 101 010 11	000 000 000 01

Table 7-6 V2V mapping for probPartIdx = 4

bin sequence	codeword
000 00	110
11	111
000 10	011 0
000 01	011 1
100 00	100 0
010 00	100 1
001 00	101 0
101 0	101 1
000 11	000 10
011 00	000 11
001 10	001 00
001 01	001 01
010 01	001 10
010 10	001 11
100 01	010 00
100 10	010 01
011 1	010 10
101 1	010 11
011 01	000 000
001 11	000 001
010 11	000 010
100 11	000 011

Table 7-7 V2V mapping for probPartIdx = 5

bin sequence	codeword
00	1
010 00	001 1
100 00	010 0
011	010 1
101	011 0
110	011 1
010 10	000 01
010 01	000 10
100 01	000 10
100 10	001 00

111	001 01
010 11	000 000
100 11	000 001

Table 7-8 V2V mapping for probPartIdx = 6

bin sequence	codeword
1	11
010	100
000 000 0	101
001 000	010 0
000 010	101 1
000 100	011 0
000 001	011 1
011 0	000 11
000 000 10	001 00
001 10	001 01
001 01	001 10
000 11	001 11
001 001 0	000 010
000 101 0	000 011
000 011 0	000 100
011 1	001 101
001 11	000 001 0
000 000 110	000 001 1
001 001 1	000 000 00
000 101 1	000 000 01
000 011 1	000 000 10
000 000 111	000 000 11

Table 7-9 V2V mapping for probPartIdx = 7

bin sequence	codeword
000	1
010	001
100	010
001	011
110 0	000 10
011 0	000 11
101 000 0	000 011
111 0	000 001 0
011 1	000 001 1
110 1	000 010 0
101 1	000 010 1
101 001	000 000 01
101 000 1	000 000 10
101 010 0	000 000 11
111 1	000 000 001
101 011	000 000 000 0
101 010 1	000 000 000 1

Table 7-10 V2V mapping for probPartIdx = 8

bin sequence	codeword
000 0	1
10	010
010	011
000 100	001 0
001 000	001 1
000 11	000 011
001 01	000 100

001 10	000 101
011 00	000 110
110 00	000 111
000 101 00	000 010 0
001 001 00	000 010 1
111	000 000 10
011 1	000 000 11
110 1	000 001 00
001 11	000 001 01
011 01	000 001 10
110 01	000 001 11
000 101 1	000 000 000
001 001 1	000 000 001
000 101 01	000 000 010
001 001 01	000 000 011

Table 7-11 V2V mapping for probPartIdx = 9

bin sequence	codeword
000 000 00	1
100	001 0
010	001 1
001 0	010 0
000 1	010 1
000 01	011 0
000 001	011 1
000 000 100 00	000 10
000 000 010 000	000 11
11	000 010 0
011	000 010 1
101	000 011 0
001 1	000 011 1
000 000 11	000 000 00
000 000 101	000 000 01
000 000 011	000 000 10
000 000 100 1	000 000 11
000 000 010 1	000 001 00
000 000 100 01	000 001 01
000 000 010 01	000 001 10
000 000 010 001	000 001 11

Table 7-12 V2V mapping for probPartIdx = 10

bin sequence	codeword
000 000 000 000 000 00	1
1	000 01
01	000 10
001	000 11
000 1	001 00
000 01	001 01
000 001	001 10
000 000 1	001 11
000 000 01	010 00
000 000 001	010 01
000 000 000 1	010 10
000 000 000 01	010 11
000 000 000 001	011 00
000 000 000 000 1	011 01
000 000 000 000 01	011 10
000 000 000 000 001	011 11

000 000 000 000 000 1	000 000
000 000 000 000 000 01	000 001

Table 7-13 V2V mapping for probPartIdx = 11

bin sequence	codeword
000 000 000 000 000 000 000 000 000 000 00	1
000 000 000 000 000 000 000 000 000 000 01	000 000
000 000 000 000 000 000 000 000 000 000 1	000 001
000 000 000 000 000 000 000 000 000 001	000 010
000 000 000 000 000 000 000 000 000 01	000 011
000 000 000 000 000 000 000 000 000 1	000 100
000 000 000 000 000 000 000 000 001	000 101
000 000 000 000 000 000 000 000 01	000 110
000 000 000 000 000 000 000 000 1	000 111
000 000 000 000 000 000 000 001	001 000
000 000 000 000 000 000 000 01	001 001
000 000 000 000 000 000 000 1	001 010
000 000 000 000 000 000 001	001 011
000 000 000 000 000 000 01	001 100
000 000 000 000 000 000 1	001 101
000 000 000 000 000 001	001 110
000 000 000 000 000 01	001 111
000 000 000 000 000 1	010 000
000 000 000 000 001	010 001
000 000 000 000 01	010 010
000 000 000 000 1	010 011
000 000 000 001	010 100
000 000 000 01	010 101
000 000 000 1	010 110
000 000 001	010 111
000 000 01	011 000
000 000 1	011 001
000 001	011 010
000 01	011 011
000 1	011 100
001	011 101
01	011 110
1	011 111

[Ed. In our understanding, clauses 8 to 14 should be removed, when the decoding process description stabalizes]

8 Unit definition

8.1 Coding Tree Block (CTB)

The coding unit (CTB) is defined as a basic unit which has a square shape. Although it has a similar role to the macroblock and sub-macroblock in AVC, the main difference lies in the fact that CTB can have various sizes, with no distinction corresponding to its size. All processing except frame-based loop filtering is performed on a CTB basis, including intra/inter prediction, transform, quantization and entropy coding. Two special terms are defined: the largest coding unit (LCTB) and the smallest coding unit (SCTB). For convenient implementation, LCTB size and SCTB size are limited to values which are a power of 2 and which are greater than or equal to 8.

It is assumed that a picture consists of non-overlapped LCTBs. Since the CTB is restricted to be a square shape, the CTB structure within a LCTB can be expressed in a recursive tree representation adapted to the picture. That is, CTB is characterized by LCTB size and the hierarchical depth in the LCTB that the CTB belongs to.

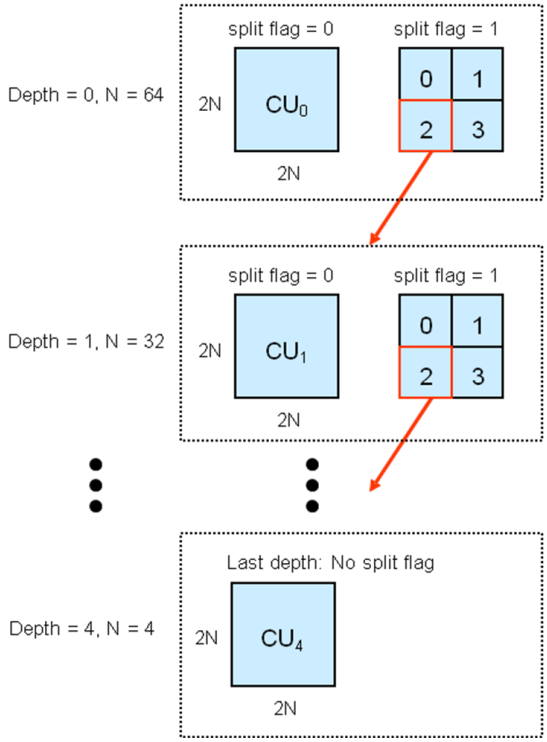


Fig. 1: Illustration of recursive CTB structure (LCTB size = 128, maximum hierarchical depth = 5)

Figure 1 shows an example where LCTB size is 128 and the maximum hierarchical depth is 5. The recursive structure is represented by a series of split flags. For CTB_d, which has depth d and size 2Nx2N, the coding of CTB is performed in the current depth when split flag is set to zero. When the split flag is set to 1, CTB_d is split into 4 independent CTB_{d+1} which have depth (d+1) and size NxN. In this case, CTB_{d+1} is called a sub-CTB of CTB_d similar to a sub-macroblock in AVC. Unless the depth of sub-CTB (d+1) is equal to the maximum allowed depth (4 in this case), each CTB_{d+1} is processed in a recursive manner. If the depth of sub-CTB (d+1) is equal to the maximum allowed depth, further splitting is not allowed. Note that a CTB can be further split into PUs.

The sizes of LCTB and SCTB are specified in the Sequence Parameter Set (SPS). The embedded information in the SPS is LCTB size (s) and the maximum hierarchical depth (h) in a LCTB. For example, if s = 128 and h = 5, then 5 kinds of CTB sizes are possible: 128x128 (LCTB), 64x64, 32x32, 16x16 and 8x8 (SCTB). If s = 16 and h = 2, then 16x16 (LCTB) and 8x8 (SCTB) are possible; this is a similar block structure to AVC. Therefore, if the LCTB size and maximum hierarchical depth are given, this defines the possible CTB sizes which are allowed.

This kind of arbitrary unit representation provides several major benefits. The first benefit comes from the support of CTB sizes greater than the conventional 16x16 macroblock. When the region of interest is homogeneous, a large CTB can represent the region in a smaller number of symbols than is possible in the case of several small blocks.

Furthermore, supporting arbitrary LCTB sizes enables the codec to be readily optimized for various content, applications and devices. Compared to the use of fixed size macroblock, support of various LCTB sizes is one of the major properties. It is especially useful for low resolution video services, which is still commonly used in the market. By choosing LCTB size and maximum hierarchical depth appropriately, the hierarchical block structure can be optimized in a better way for the targeted application. In the later stage of the standardization, the range of LCTB sizes could be specified in the Profiles and Levels section to match the requirements more specifically.

Finally, by eliminating the distinction between macroblock and sub-macroblock and using only one unit type, CTB, the multi-level hierarchical structure can be specified in a very simple and elegant way: LCTB size, maximum hierarchical depth and a series of split flags. Together with the proposed size-independent syntax representation, it's sufficient to specify syntax items of one general size for the remaining coding tools. This kind of consistency can greatly simplify the specification effort as well as the actual parsing process.

The maximum depth for CTB hierarchy is arbitrary and potentially larger than the maximum depth allowed in the AVC block hierarchy. Therefore, the text includes a size-independent syntax representation which specifies all syntax elements in a consistent way independent of the CTB size. By contrast, in AVC block level syntax elements

such as transform_8x8_mode_flag, coded_block_flag and intra_pred_modes are coded differently depending on whether the block size is 16, 8 or 4. The splitting process for the CTB can be specified recursively and all other syntax elements for the leaf CTB are defined in the same way independent on the CTB size. This kind of representation is very useful in terms of reduced parsing complexity and improved clarity if a large hierarchical depth is allowed.

For the purpose of mode decision, transmission of the data associated with each block etc. all CTBs are traversed in raster scan order (left-to-right, top-down), and within each CTB, the subblocks are traversed in depth-first order.

8.2 Prediction unit (PU)

Once the splitting process is done, prediction methods are specified for every CTB which is not further split i.e. the leaf nodes of the CTB hierarchical tree.

Coupled with CTB, a basic unit for the prediction mode is introduced: the prediction unit (PU). It should be noted that the PU is defined only for the last-depth CTB, i.e., the leaf nodes of the CTB-related quadtree and its size is limited to that of the CTB.

Similar to conventional standards, we define two different terms to specify the prediction method: the prediction type and the PU splitting. The prediction type is one of the values among intra or inter, which roughly describe the nature of the prediction method. After that, possible PU splittings are defined according to the prediction type.

The PU for intra has 2 different possible splittings: 2Nx2N (i.e. no split) and NxN (quarter split). The PU for inter has 8 different possible splittings: 4 symmetric splittings (2Nx2N, 2NxN, Nx2N, NxN) and 4 asymmetric splittings (2NxnU, 2NxnD, nLx2N and nRx2N). The number N is derived from the size of the CTB which the PU belongs to. For example, if the size of CTB is 128x128, then both 128x128 and 64x64 PUs for intra are possible. For inter prediction, 128x128, 128x64, 64x128, 64x64, 128x32 (for 2NxnU and 2NxnD), 128x96 (for 2NxnU and 2NxnD), 32x128 (for nRx2N and nLx2N) and 96x128 (for nRx2N and nLx2N) are possible.

Figure 1a shows an example where a 64x64 block is asymmetrically partitioned into 64x16, 64x48, 16x64, or 48x16 blocks, instead of two 64x32 or 32x64 blocks. Note that the motion partitions shown in the figure correspond to 2NxnU, 2NxnD, nLx2N, and nRx2N, respectively.

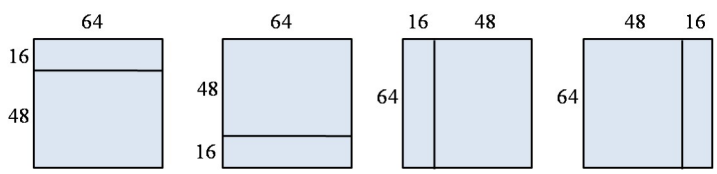


Figure 1a: Example of asymmetric motion partitions for 64x64 block

In addition, we have geometrical shapes for partitioning of inter PUs that are described as follows.

For each CTB, one type of PU splittings is defined by an arbitrary straight line across the CTB that divides the CTB into two regions. The boundary of the PU partition is defined by an angle subtended by a line perpendicular to the boundary with the X axis and the distance of the partition line from the origin . The equation of the line defining the partition boundary is specified as

$$y = \frac{-1}{\tan\theta}x + \frac{\rho}{\sin\theta} = mx + c.$$

Two 32 bit lookup tables are used, one to store the slope, $m = -1/\tan\theta$, and the other to store the Y-intercept, $c = \rho/\sin\theta$. The region to which each pixel belongs is calculated on the fly.

At each level CTB, 32 different values of are defined (from 0 to in steps of). The number of values that ρ can take depends on the CTB size. For example, for CTB size of 16×16, takes 8 possible values (from 0 to 7 in steps of 1). For block sizes of 32×32 and 64×64, takes 16 and 32 possible values, respectively.

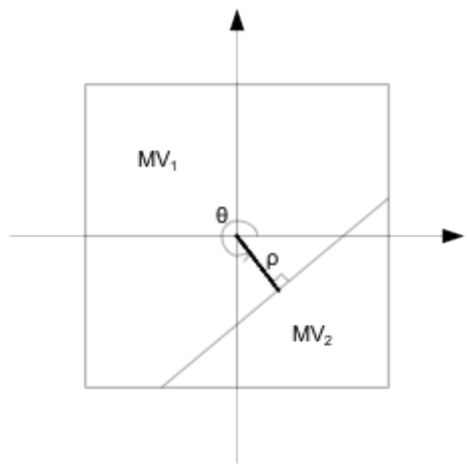


Figure 2: Parameters defining a geometry motion partition

For a CTB that is predicted in a manner described above, overlapped block motion compensation (OBMC) is applied. Let the two regions created by a geometry partition be denoted by region 1 and region 2. Let the corresponding motion vectors be denoted by MV_1 and MV_2, respectively. A pixel from region 1 (2) is defined to be a boundary pixel if any of its four connected neighbors (left, top, right, and bottom) belongs to region 2 (1). Fig. 2 shows an example where light blue pixels belong to the boundary of region 1 and white pixels belong to the boundary of region2. If a pixel is not a boundary pixel, normal motion compensation is performed using the appropriate motion vector. If a pixel is a boundary pixel, the motion compensation is performed using a weighted sum of the motion predictions from the two motion vectors, MV_1 and MV_2. The weights are $2/3$ for the region containing the boundary pixel and $1/3$ f for the other region.

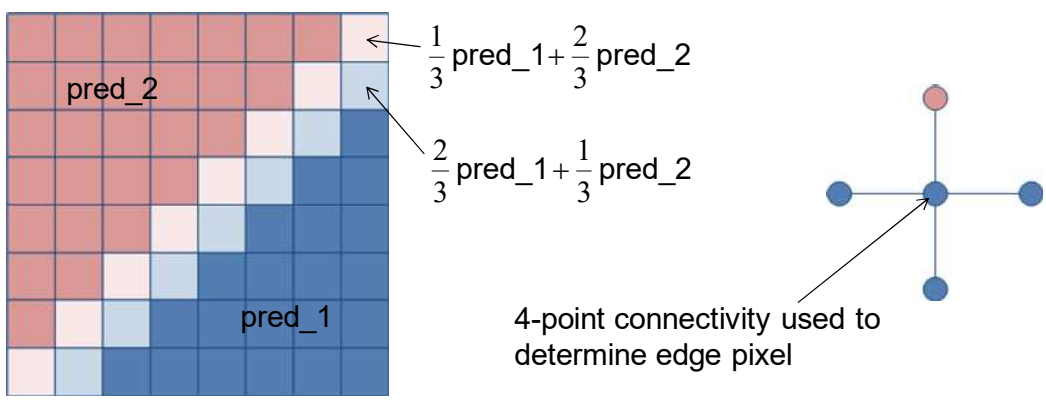


Figure 3: Overlapped motion compensation for geometry partitions

All information related to prediction is signaled on a PU basis, for instance, the direction of intra prediction for intra or the motion vector difference and reference index for inter. Motion vector prediction and motion compensation are also performed on a PU basis.

For each inter predicted blocks, individual motion parameters are transmitted. In order to achieve an improved coding efficiency, the block merging process enables to merge neighbouring blocks into so-called regions. By doing so, the motion parameters do not need to be transmitted for each block of the region individually, but instead the parameters are transmitted only for once for the whole region.

For each inter prediction block, the set of all prediction blocks that are coded before that block in processing order is called the "set of causal blocks". The set of blocks that is admissible for merging with a particular block is called the "set of available blocks" and is always a subset of the set of causal blocks. If a particular block is encoded and its set of available blocks is not empty, it is signaled whether this block is to be merged with one block out of this set and if so, with which of them. Otherwise, merging cannot be used for this block.

The set of available blocks is formed as follows. Starting from the top-left sample position of the current block, its left neighbouring sample position and its top neighbouring sample position is derived. The set of available blocks can have only up to two elements, namely those blocks out of the set of causal blocks that contain one of the two sample positions. Thus the set of available blocks can only have the two direct neighbouring blocks of the top-left sample position of the current block as its elements. Note that only inter predicted blocks can be members of the set of available blocks.

If the set of available blocks is not empty, one flag called merge_flag is signaled, specifying whether the current block is merged with any of the available blocks. Otherwise, or if the merge_flag is equal to 0 (for "false"), this block is not merged with one of its causal blocks and all parameters are transmitted ordinarily. If the merge_flag is equal to 1 (for "true"), the following applies. If the set of available blocks contains one and only one block, this block is used for merging. Otherwise the set of available blocks contains exactly two blocks. If the motion parameters of these two blocks are identical, these motion parameters are used for the current block. Otherwise (the two blocks have different motion parameters), a flag called merge_left_flag is signaled. If merge_left_flag is equal to 1 (for "true"), the block containing the left neighbouring sample position of the top-left sample position of the current block is selected out of the set of available blocks. If merge_left_flag is equal to 0 (for "false"), the other (i.e., top neighbouring) block out of the set of available blocks is selected. The motion parameters of the selected block are used for the current block.

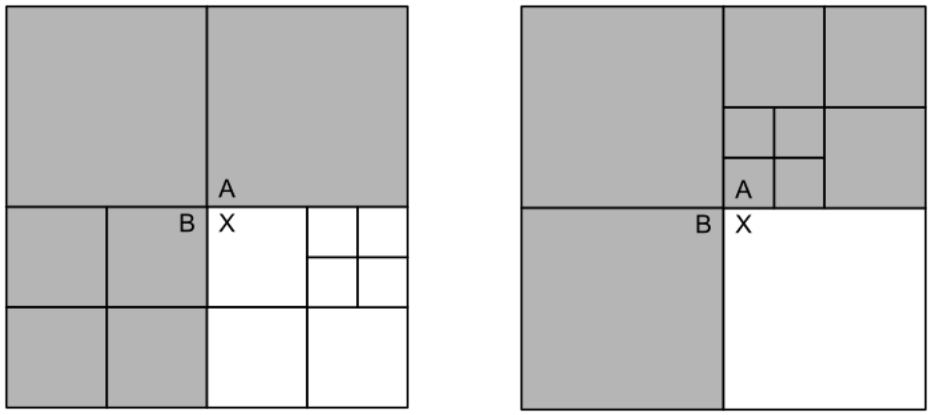


Fig. 3: Two examples illustrating the set of causal blocks (grey) and the set of available blocks (A, B).

Fig. 3 shows two examples for a quadtree-based division of a picture into prediction blocks. On the left side of Fig. 4, the top two blocks of the biggest size are so-called macroblocks, i.e., they are prediction blocks of the maximum possible size. The other blocks in this left figure are obtained as a subdivision of their corresponding macroblock. The current block is marked with an "X". All the grey-shaded blocks are encoded before the current block, so they form the set of causal blocks. As explicated in the description of the derivation of the set of available blocks, only the blocks containing the direct (i.e. top or left) neighbouring samples of the top-left sample position of the current block can be members of the set of available blocks. Thus the current block can be merged with either block "A" or block "B". If merge_flag is equal to 0 (for "false"), the current block "X" is not merged with any of the two blocks. If blocks "A" and "B" have identical motion parameters, no distinction needs to be made, since merging with any of the two blocks will lead to the same result. So, in this case, the merge_left_flag is not transmitted. Otherwise, if blocks "A" and "B" have different prediction parameters, merge_left_flag equal to 1 (for "true") will merge blocks "X" and "B", whereas merge_left_flag equal to 0 (for "false") will merge blocks "X" and "A".

In Fig. 4 (right) another example is shown. Here the current block "X" and the left neighbour block "B" are macroblocks, i.e., they have the maximum allowed block size. The size of the top neighbour block "A" is one quarter of the macroblock size. The blocks which are element of the set of causal blocks are grey-shaded. Note that the current block "X" can only be merged with the two blocks "A" or "B", not with any of the other top neighbouring blocks.

8.3 Transform unit (TU)

In addition to the CTB and PU definitions, the transform unit (TU) for transform and quantization is defined separately. It should be noted that the size of the TU may be as large as the size of the CTB. Like the partitioning used for prediction, the partitioning of a single prediction error block into transform blocks is represented by quad

trees. The greatest applicable transform size and the maximum depth of such quad trees (giving the minimum applicable size) are variable and are signaled in the bit stream. Transform blocks with an edge length of any positive power of two can be employed. Transform block sizes are constrained to the range 4x4 to 64x64 (for chroma, also 2x2 transforms may be used).

9 Motion representation

9.1 Motion vector prediction for rectangular partitions

A new method to form motion vector prediction is introduced. Unlike in AVC, this method takes the temporal distance of the reference frames into account for motion vector prediction. The neighboring blocks that are used for the prediction of the vertical components are illustrated in Figure 2.

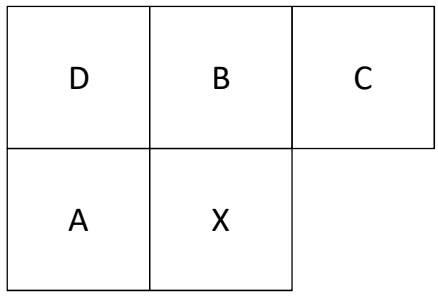


Figure 2: Neighboring blocks used for motion vector prediction

But the motion vectors of neighboring blocks A, B, and C may point to different reference frames in the buffer. Therefore, the motion vectors are normalized based on their temporal distance from the current frame, before taking the median. This is illustrated in Figure 3 for an IPPP configuration. In the figure, blue colored frames are the reference frames stored in the buffer. Let A_r, B_r, C_r, and X_r refer to the reference frames corresponding to blocks A, B, C, and X, respectively. Let their temporal distances from the current frame be denoted by $d(A_r), d(B_r), d(C_r)$, and $d(X_r)$, respectively. Then, each of the neighboring motion vectors, $MV^Y, Y \in \{A, B, C\}$ is scaled as follows:

$$\text{scaled_}MV^Y = MV^Y \frac{d(X_r)}{d(Y_r)}.$$

This scheme is applied to both P and B frames.

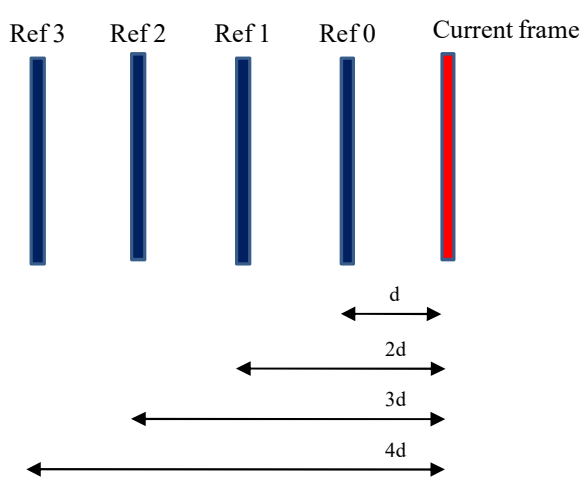


Figure 3: Reference frames and their temporal distances from the current frame

Depending on the settings of interleaved_motion_predictor_flag and mv_competition_flag, the derivation process of motion vector predictor is determined in one of the following ways as shown in 6.3.1.4:

interleaved_motion_predictor_flag	mv_competition_flag	normative section
0	1	6.3.1.4.1
0	0	6.3.1.4.6
1	1	6.3.1.4.7
1	0	6.3.1.4.6 and 6.3.1.4.9

9.2 Motion vector prediction for geometric block partitions

The process of forming motion vector prediction for a geometric block partition is shown Figure 2. Motion vectors of neighboring blocks are used to form the motion vector prediction. Depending on the partition, a subset of blocks from {A, B, C, E, F} are used to form motion vector prediction. The choice of block for motion vector prediction depends upon whether they lie on the same side or opposite side of the geometry partition boundary.

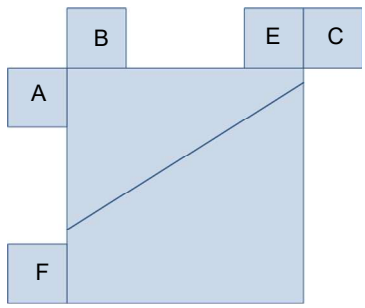


Figure2: Motion vector prediction for a block having geometry partition

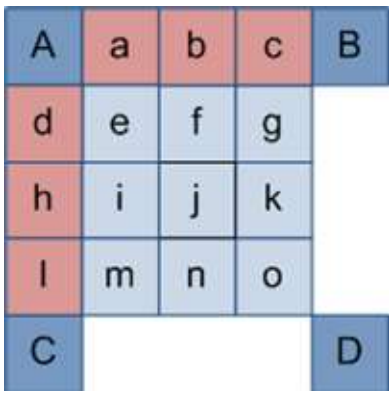
Let blkA, blkB, blkC, blkD, blkE and blkF be the prediction blocks that contain the luma samples at locations (x0 − 1, y0), (x0, y0 − 1), (x0 + currW, y0 − 1), (x0 − 1, y0 − 1), (x0 + currW − 1, y0 − 1), and (x0 − 1, y0 + currW − 1), respectively. For each partition, if blkX (X being replaced A, B, C, D, E and F) is sharing a boundary with the partition, blkX is used to derive the prediction motion vector.

The prediction motion vector mvPred is derived as specified by the folowing ordered steps:

1. If blkA, blkB and blkC are all connected to the current partition, median filter is used to derive mvPred.
2. Otherwise, if blkA is connected to the current partition, mvPred = mvA
3. Otherwise, if blkB is connected to the current partition, mvPred = mvB
4. Otherwise, if blkE is connected to the current partition, mvPred = mvE
5. Otherwise, if blkF is connected to the current partition, mvPred = mvF
6. Otherwise, if blkC is connected to the current partition, mvPred = mvC
7. Otherwise mvPred = (0,0)

9.3 Interpolation Methods

This text uses single pass switched interpolation filters with offsets (single pass SIFO). The 1/4th pixel motion positions are shown in the figure.



9.3.1 Single pass switched Interpolation Filters with offsets (single pass SIFO)

In this text, 7 different filter sets are defined. Each filter set consists of 15 filters, one for each fractional pixel position.

- Filter set 0: This filter set uses high precision filtering with the filters shown in the following table with the exception of position 'g', where a non-separable filter is used.

Table 9-1: 6-tap separable filters

1/4	{ 8, -32, 224, 72, -24, 8 } (8 additions, 4 shifts)
1/2	{ 8, -40, 160, 160, -40, 8 } (6 additions, 3 shifts)
3/4	{ 8, -24, 72, 224, -32, 8 } (8 additions, 4 shifts)

The filter coefficients are scaled by 256 and the intermediate data is kept in higher precision. For position 'g', the following filter is used (followed by right shift by 7 bits):

0	5	5	0
5	22	22	5
5	22	22	5
0	5	5	0

- Filter set 1: This filter set uses 12 tap filters for both horizontal and vertical filtering. The filter coefficients are scaled by 256. A higher bit depth of filter coefficients leads to more accurate interpolation; for comparison, AVC uses 5 bits for filter coefficients. Also, the intermediate values are maintained at higher bit precision.

Table 9-2: 12-tap separable filters

¼	{-1, 5, -12, 20,-40, 229, 76, -32, 16, -8, 4,-1 } (18 additions, 6 shifts)
½	{-1, 8, -16, 24,-48, 161, 161, -48, 24, -16, 8,-1 } (15 additions, 4 shifts)
¾	{-1, 4, -8, 16, -32, 76, 229, -40, 20, -12, 5,-1 } (18 additions, 6 shifts)

- Filter set 2: This set of interpolation filter is referred to as the Directional Interpolation Filter (DIF) and is used for all 15 quarter-pixel positions. For each of the three horizontal positions and the three vertical positions which are aligned with full pixel positions, a single 6-tap filter is used. For the 9 innermost quarter-pixel positions two 6-tap filters at +45 degree and -45 degree angles are used.

The filter coefficients for DIF are
[3 -15 111 37 -10 2]/128 for ¼ displacement (and mirrored for ¾ displacement)
[3 -17 78 78 -17 3]/128 for ½ displacement

- Filter set 3: The third set of interpolation filters consists of Separable Filters (SF) where interpolated samples are calculated by first applying a 6-tap filter horizontally and then vertically.

  The filter coefficients for SF are
  [3 -15 111 37 -10 2]/128 for ¼ displacement (and mirrored for ¾ displacement)
  [3 -17 78 78 -17 3]/128 for ½ displacement

- Filter set 4: Consists of a combination of a 2-tap pre-filter and a 4-tap interpolation filter.
  - The pre-filter is a 1D IIR filter in horizontal and vertical directions. The coefficients are given below:
    - Left-to-right filtering: [1] / [1, c]
    - Right-to-left filtering: [c] / [1, c]

    where, the constant c = -11276. The pre-filter produces the same number of samples as the input to it. The filtering can be performed on the picture region to be interpolated.

  - The interpolation filter is implemented as a sequence of 1D horizontal and vertical filtering step. The FIR filters consist of 4-taps. The filter coefficients are the same for both horizontal and vertical directions. The filters used for different sub-pel positions are given below:
    - Full-pel: [6242, 20284, 6242, 0] / 32768
    - 1/4 pel : [2888, 19078, 10520, 280] / 32768
    - 1/2 pel : [1073, 15311, 15311, 1073] / 32768
    - 3/4 pel : [280, 10520, 19078, 2888] / 32768

- Filter set 5 and 6: For each set, positions *a*, *b*, and *c* use a six-tap horizontal filters. Positions *d*, *h*, and *l* use six-tap vertical filters. For the remaining fractional pixel positions, 4×4 non-separable filters are used. Each of the non-separable filters has horizontal, vertical or diagonal symmetry.

9.3.2 Choice of filter set and offsets

The choice of the filter and offset is signalled to the decoder for each slice. For each of the 15 fractional pixel positions, the choice of the filter is signalled using 3 bits. Before encoding a frame, the encoder selects a filter for each fractional pixel position based on statistics gathered from previously encoded frames. In this text, the filter that minimizes prediction error for the previously encoded frames is selected. For each fractional pixel position, the minimization is performed only on blocks whose motion vector points to that fractional pixel location. The choice of filter remains the same irrespective of the reference frame in which the motion search is being performed.

For reference frame 0 from each list, offsets are sent to the decoder for each of the 15 fractional pixel positions as well as the full pixel position For other reference frames only one frame offset is sent. The offsets are encoded using exponential Golomb codes.

9.3.3 Partition based Illumination Compensation (PBIC)

Illumination-compensation parameters are signaled at each partition using scale and offset parameters.

For single-list prediction, one transformed scale t0 and one offset o are transmitted. Scale s is derived from t0 as s = (64+t0).

For bi-prediction, two transformed scales t0 and t1, and one offset o are transmitted. Scales s0 and s1 are derived from t0 and t1 as s0 = (64 + t0)/2+t1 and s1 = (64+t0) – s0.

The scales are applied to all components, whereas the offsets are applied only to the luma component. In addition, for chroma components, the value '128' (chroma value corresponding to a gray pixel without color) is removed before the application of scale and this value '128' is added after the application of scale.

In the case of luma component, illumination compensation is applied to motion-compensated samples as follows:

- Single-list Prediction

- w = (s*v)/64+o
- w is illumination-compensated sample
- v is motion-compensated sample

- Bi-prediction
  - w = (w0+w1)/2
  - w0 = (2*s0*v0/64)+o
  - w1 = (2*s1*v1/64)+o
  - w is illumination-compensated sample
  - w0 and w1 are illumination-compensated samples for each list
  - v0 and v1 are motion-compensated samples for each list

Note that the implementation is done such that w0 and w1 are obtained first to be inline with the TMuC implementation of motion compensation for bi-prediction. The samples w0 and w1 are later averaged to obtain w.

In the case of chroma components, illumination compensation is applied to motion-compensated samples as follows (the terms v, v0, v1, w, w0, w1 have the same meaning as above, but represent chroma samples instead of luma samples):

- Single-list Prediction
  - w = (s*(v-128))/64+128
- Bi-prediction
  - w = (w0+w1)/2
  - w0 = (2*s0*(v0-128)/64)+128
  - w1 = (2*s1*(v1-128)/64)+128

Note that the above equations (for both luma and chroma) assume that the motion-compensated values are represented using an unsigned 8-bit notation. When IBDI is used, appropriate scaling modifications are used to do the operations at higher bit-depth.

The applicable PBIC parameters ({t0, o} or {t0, t1, o}) are coded differentially in combination with motion vector differences. A list of predictors is constructed using a default predictor (all parameters being 0) and predictors using PBIC parameters of neighboring PUs. A predictor index is signaled to indicate the predictor to be used. A zero-tree is used to indicate which of the PBIC parameter differences and motion vector differences are zero. For single-list prediction, the zero-tree deals with 4 parameters (mvdx, mvdy, t0 and o). For bi-prediction, the zero-tree deals with 7 parameters (mvd0x, mvd0y, mvd1x, mvd1y, t0, t1 and o). The non-zero PBIC parameter differences are coded similar to non-zero motion vector differences.

In the deblocking filter, edges between blocks with different PBIC parameters are treated similar to the edges between blocks with MV differences.~~Intensity compensation parameters are signaled at each partition using scale and offset parameters.~~

~~In the single list prediction case, one scale s and one offset o are transmitted. For bi-prediction, two transformed scales t0 and t1, and one offset o are transmitted. Scales s0 and s1 are derived from t0 and t1 as s0 = (64 + t0)/2+t1 and s1 = (64+t0) – s0.~~

~~The scales are applied to all components, whereas the offsets are applied only to the luma component (in the following the offset o is considered to be 0 for chroma components).~~

~~In the single list prediction case, a predicted, illumination compensated, sample w is set to ((64+s)*v)/64+o, where v is a motion compensated sample. For bi-prediction, a predicted sample w is obtained by the weighted average (s0*v0+s1*v1)/64+o where v0 and v1 are the motion compensated samples from each list.~~

~~The parameters s, t0, t1 and o are coded differentially in combination with motion vector differences.~~

~~This feature is enabled by a flag on slice level.~~

9.4 Adaptive Motion Vector Resolution

For each region in a motion partition, the motion accuracy can be adaptively chosen to be 1/4th pixel or 1/8th pixel. We will refer to this as adaptive motion vector resolution. The choice of motion vector resolution is signaled to the decoder. For each motion vector, a motion vector resolution flag is encoded. If the flag is zero, the motion vector

precision is 1/4th otherwise 1/8th. If the flag is 1 and motion vector is nonzero, refinement information for this motion vector is sent which specifies 1/8th pel precision.

The encoder always maintains the motion vector (MV) and MVD information at 1/8th pixel resolution. Then, the MV prediction for the current block is formed with 1/8th pixel accuracy. If the current block has only 1/4th pixel motion accuracy, the MV prediction is converted to 1/4th pixel accuracy. On the other hand if the current block has 1/8th pixel motion accuracy, the MVD is formed directly by subtracting the MV prediction from the motion vector for the current block. Once MVD is formed, if the current block has 1/4th pixel accuracy, for all the neighboring blocks used for determining the MVD contexts, the MVDs are converted to 1/4th pixel accuracy. Similar procedure is followed for 1/8th pixel accuracy.

10 Intra-frame prediction

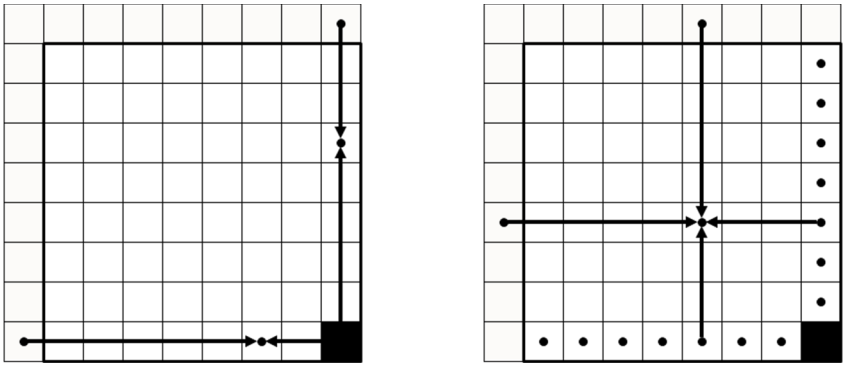
For blocks of size 64x64 : 33 Directions (ADI+Planar)
For blocks of size 32x32 : 33 Directions (ADI+Planar)
For blocks of size 16x16 : 33 Directions (ADI+Planar)
For blocks of size 8x8 : 33 directions (Angular+Planar)
For blocks of size 4x4: 9 directions (AVC)

10.1 Adaptive reference sample smoothing

In addition, a smoothing operation can be applied to the reference samples as a pre-processing step before calculating the prediction. This smoothing operation corresponds to applying an FIR-filter (1,2,1)>>2 with low-pass characteristics to the samples along the two reference rays. Whether this smoothing operation is used is determined by the encoder and signaled in the bit stream.

10.2 Planar prediction

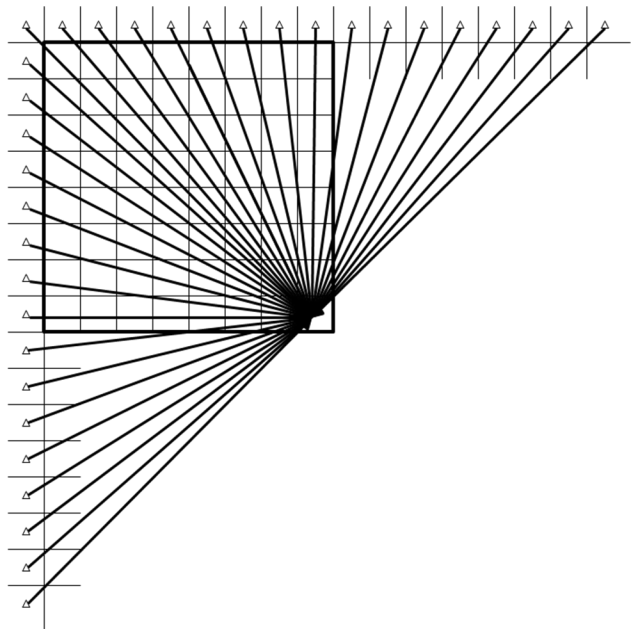
The planar prediction is designed to be able to reconstruct smooth image segments in a visually pleasing way. It provides maximal continuity of the image plane at the macroblock borders and is able to follow gradual changes of the pixel values by signalling a planar gradient for each macroblock coded in this mode. When a macroblock is coded in planar mode its bottom-right sample is signalled in the bitstream, the rightmost and bottom samples of the macroblock are linearly interpolated, and the middle samples are bilinearly interpolated from the border samples. When planar mode is signalled, the same algorithm is applied to luminance and both chrominance components separately with individual signalling of the bottom-right samples (16x16 based operation for luminance and 8x8 based for chrominance). The process is illustrated in the picture below.



Planar prediction of an 8x8 (chrominance) block. Bottom-right sample is signalled in the bitstream, rightmost and bottom samples are interpolated linearly, and the middle samples are interpolated bi-linearly.

10.3 Angular prediction

In order to be able to accurately represent directional structures the Intra 8x8 coding mode provides a possibility to predict the blocks at any direction shown below.



Available prediction directions in the angular mode of Intra 8x8 macroblocks.

In the angular mode the prediction direction is given by the displacement of the bottom row of the block and the reference row above the block in the case of vertical prediction or displacement of the rightmost column of the block and reference column left from the block in the case of the horizontal prediction. The displacement is signalled at 1 pixel accuracy. When projection of the predicted pixel falls inbetween reference samples, the predicted value for the pixel is linearly interpolated from the reference samples (at 1/8th pel accuracy). The process is illustrated in the picture below for the sixth row of the block when building vertical prediction at +1 pixel displacement.

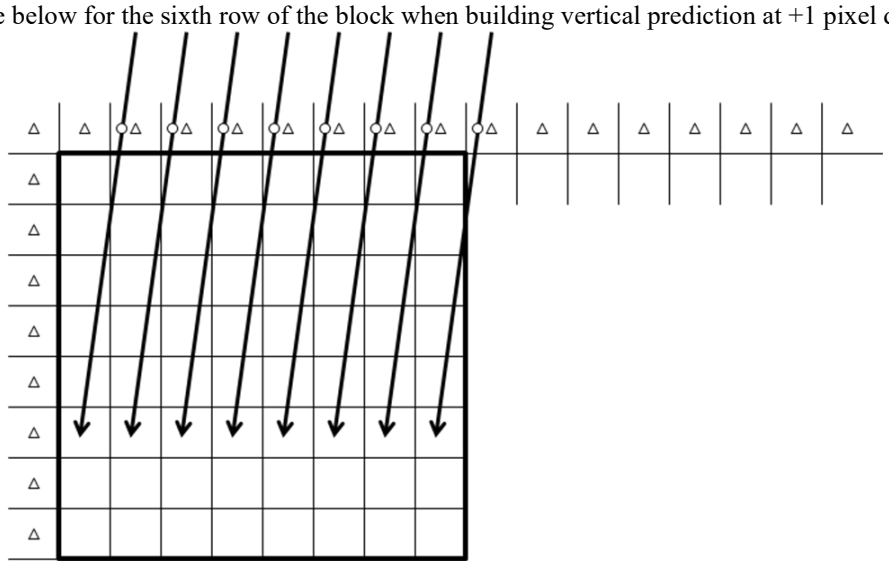
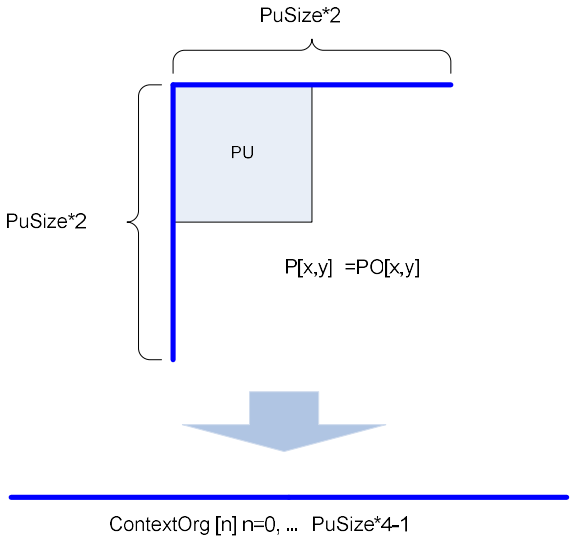


Figure 4 An example of angular prediction when operating the sixth row of the block with +1 pixel displacement. Projections of the pixels now fall at the 6/8 sub-pixel location (circles) in the reference row of pixels.

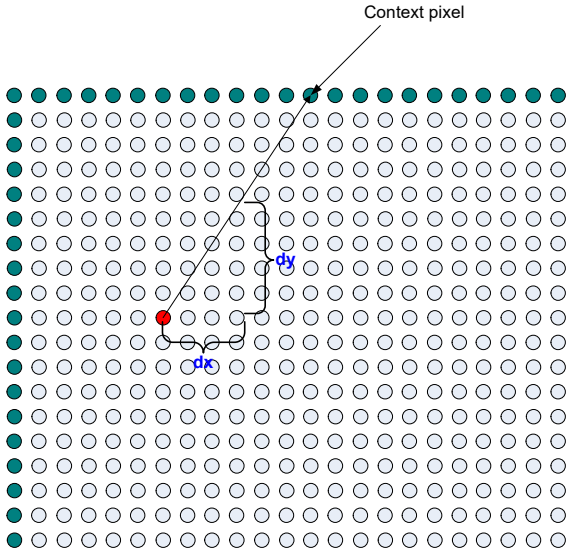
10.4 Arbitrary Directional Intra (ADI)

Arbitrary Directional Intra (ADI) generates prediction pixels by directional extrapolation or calculation using the nearest boundary pixels of the already decoded area. But in ADI, even boundary pixels from the left down region may be used as context pixels for prediction as depicted in below



Example of context pixels for ADI

Whilst the 9 prediction modes are defined separately as Vertical, Horizontal, DC, Diagonal Down-Left, Diagonal Down-Right, Vertical-Right, Horizontal-Down, Vertical-Left, and Horizontal-Up in AVC, most of prediction modes in ADI are defined by integer pair information (dx, dy). The (dx, dy) pair represents the direction which each mode uses for context pixel extrapolation



prediction methods defined by (dx, dy)

10.5 Edge based prediction

If the edge based prediction mode is switched on, the DC prediction mode can be replaced by an adaptive directional prediction mode depending on the presence of an edge in the neighboring reconstructed pixels.
The edge-based prediction mode consists of two steps:

- edge detection based on the Sobel operators to determine if an edge is present and if an edge is present, its direction
- extrapolation of the reference pixels along the direction of the detected edge

If an edge intersecting with the block to be predicted is detected, the reference pixels are extrapolated along the direction of the detected edge as showed in Figure 5.

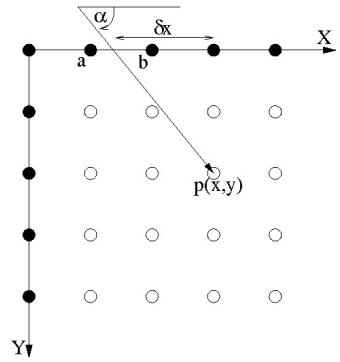


Figure 5: Computation of the edge based prediction for intra coding

The prediction of the pixel p(x,y) is:

$$w_a = \delta x - floor(\delta x)$$
$$w_b = ceil(\delta x) - \delta x$$
$$p = w_a \times a + w_b \times b$$

10.6 Combined Intra Prediction (CIP)

The CIP is a low-complexity tool for providing improved prediction, especially in large blocks where a directional linear prediction may not work well, even with the large range of angles available to ADI. CIP predictions comprise a weighted combination of an ADI prediction together with a pixel-by-pixel mean prediction. It provides pixel-by-pixel adaptation but is a simpler tool than, for example, local template matching approaches.

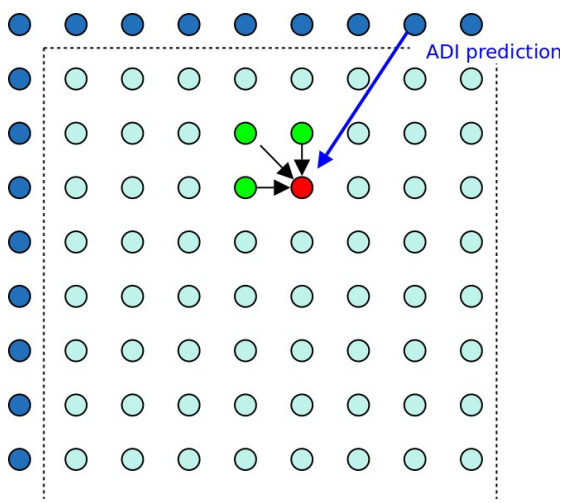


Figure 4-12 Example of Combined Intra Prediction utilizing a local mean and ADI prediction

The local mean is constructed as the average of pixel values to the left, top-left and above the current picture. In the decoder and within the local decoder in the encoder, these values are the reconstructed values after inverse quantization and transform. In the encoder, the reconstructed values are not available because quantization and transform of the prediction residue have not yet been performed, so original values are used This local mean prediction is therefore different between forward prediction (encoder only) and reconstruction (encoder and decoder).

The combination factor is a 5-bit number between 0 and 31/32 which determines the proportion of local mean in the prediction. Since this number is less than 1, the prediction/reconstruction difference noise is damped, and pixel-by-pixel prediction adaptation is possible without noise blow-up. The ADI component provides an overall prediction direction and the local mean prediction provides local adaptation.

For each CTB, a single CIP flag is sent for all PUs. If set, the flag indicates that CIP prediction is used, otherwise ADI prediction is used. The use of CIP is signaled in the SPS.

11 Spatial transforms

11.1 Large transform (16x16, 32x32, 64x64)

For smooth data, a large transform has several advantages such as better energy compaction and reduced quantization error. In HD sequences, most image patterns in a macroblock represent a small part of objects or backgrounds which can be described as homogeneous texture patterns with little variation. Therefore, the coding efficiency of high resolution video can be improved by the use of large transforms as well as large block sizes. By contrast, AVC supports only 4x4 and 8x8 transform sizes. Larger transform sizes can also be chosen (or set) for each coding unit (blocks or partitions). Transform sizes larger than the prediction unit can also be supported.

3 additional sizes of transform are included: 16x16, 32x32 and 64x64. In a large transform design, it is very important to minimise complexity. The transforms in this text are based on Chen's fast DCT algorithm [10]. Chen's algorithm is not the fastest one, but it has reduced implementation complexity due to the regular butterfly structure. Moreover, it is readily extensible to larger transform sizes.

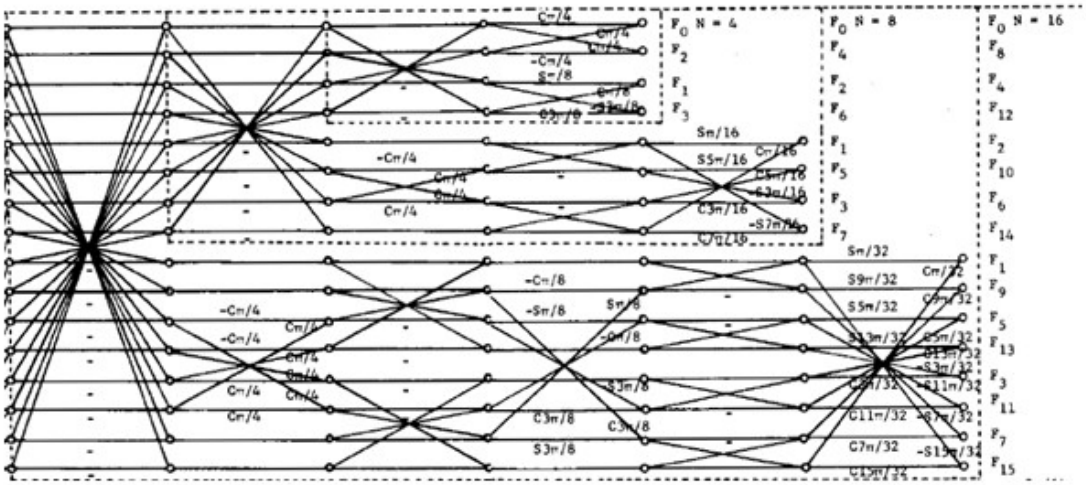


Figure 5-1 Signal flow graph of Chen's fast 16-point DCT transform

Figure 5-1 shows the signal flow graph of Chen's fast factorization of a 16 point DCT transform. In this figure, multiplication constants are represented by sinusoidal functions of specific angles, requiring floating point operations. To solve this problem, we scale and approximate the factors by fixed precision using pre-defined values, which can be calculated by cost effective shift operations. The approximated constants are shown in Table 0., with pre-defined precision value 64 in this case. Here, the ak's are approximated values of cos(k*pi/32) for k =1,2,…,15.

Table 0 Approximated constants for 16 point transform

a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	a14	a15
63/64	62/64	61/64	59/64	56/64	53/64	49/64	45/64	40/64	35/64	30/64	24/64	18/64	12/64	6/64

When the constant is approximated by the dyadic rational, the transform is no longer truly orthogonal. However, this minor non-orthogonality does not result in any perceptible negative effects on the compression performance, whilst the complexity can be significantly reduced.

11.2 Rotational transform (ROT)

DCT is the most widely used transform in block based video and image codecs. The input to the transform is the residual, i.e. the difference between the prediction and the original signal. But it is well-known that the DCT basis functions are not optimal for some types of residual signal. For example, the residual signal having strong diagonal components cannot be represented efficiently with the DCT basis vectors.

Typically, directional transform schemes are exploited to solve this problem since spatial domain rotation makes it dificult to maintain the original square shape of the block and generally requires many floating point operations. However, it's hard to develop directional transform cores, especially since many transforms of different sizes are

needed. This text uses mode dependent directional transforms (MDDT) for block sizes of 4×4 and 8×8, and uses ROT for the higher block sizes. ROT can be applied as a 2nd transform after the DCT operation for block sizes of 16×16 or higher.

The main idea of ROT is to change the coordination system of the transform basis, instead of direct rotation of the input source. For this purpose, the following matrices are defined as

$$R_{vertical}(\alpha_1,\alpha_2,\alpha_3) =$$
$$= \begin{pmatrix} \cos\alpha_1\cos\alpha_3 - \sin\alpha_1\cos\alpha_2\sin\alpha_3 & -\sin\alpha_1\cos\alpha_3 - \cos\alpha_1\cos\alpha_2\sin\alpha_3 & \sin\alpha_2\sin\alpha_3 & 0 \\ \cos\alpha_1\sin\alpha_3 + \sin\alpha_1\cos\alpha_2\cos\alpha_3 & -\sin\alpha_1\sin\alpha_3 + \cos\alpha_1\cos\alpha_2\cos\alpha_3 & -\sin\alpha_2\cos\alpha_3 & 0 \\ \sin\alpha_1\sin\alpha_2 & \cos\alpha_1\sin\alpha_2 & \cos\alpha_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_{horizontall}(\alpha_4,\alpha_5,\alpha_6) =$$
$$= \begin{pmatrix} \cos\alpha_4\cos\alpha_6 - \sin\alpha_4\cos\alpha_5\sin\alpha_6 & -\sin\alpha_4\cos\alpha_6 - \cos\alpha_4\cos\alpha_5\sin\alpha_6 & \sin\alpha_5\sin\alpha_6 & 0 \\ \cos\alpha_4\sin\alpha_6 + \sin\alpha_4\cos\alpha_5\cos\alpha_6 & -\sin\alpha_4\sin\alpha_6 + \cos\alpha_4\cos\alpha_5\cos\alpha_6 & -\sin\alpha_5\cos\alpha_6 & 0 \\ \sin\alpha_4\sin\alpha_5 & \cos\alpha_4\sin\alpha_5 & \cos\alpha_5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $R_{vertical}$ and $R_{horizontal}$ represent the rotation matrices for horizontal and vertical directions with the rotation angles of (α1, α2, α3) and (α4, α5, α6), respectively. In this text, only four out of all possible rotation angles are quantized and used, in order to minimize the encoder-side complexity. The chosen transform is signalled to the decoder using the chosen intra prediction direction as the context.

Furthermore, since ROT is a second transform, not all of the DCT coefficients are processed. For the TUs larger than 8×8, only 8x8 low-frequency areas are rotated by multiplying by the ROT matrix since most coefficients are already compacted into these low-frequency areas.

11.3 Mode dependent directional transforms for intra-prediction residuals

In this text, nine directional intra prediction modes are used for block size of 4×4 and 33 directional intra prediction modes are used for block sizes of 8×8 (INTRA4x4 and INTRA8x8). For each directional prediction mode, the prediction residual exhibits a different directionality. In addition, the magnitude of the error increases for predicted pixels farther away from the boundary pixels used for prediction. To take advantage of this, mode dependent directional transforms (MDDT) are used to encode 4×4 and 8×8 intra prediction residuals. The type of MDDT is coupled with the selected intra prediction direction, so is not explicitly signaled. We briefly describe design and implementation of the MDDT.

For the 8×8 block size, the 33 intra prediction directions are clustered into nine separate directions. Thus, for both 4×4 and 8×8 blocks, nine separate MDDT transforms are designed. The Karhunen-Loève transform (KLT) derived from the statistics of the intra prediction residuals for that mode, would be the optimal transform choice from a rate-distortion perspective. However, KLT is a non-separable transform. For a residual block of size $N \times N$, the size of the KLT matrix is $N^2 \times N^2$. Thus, KLT is prohibitively expensive in term of storage and computational requirement. A separable directional transform is used, which can be described as

$$Y = C_i X R_i ,$$

where *X, Y, C*, and *R* are all of size $N \times N$ and *i* is the prediction mode. Singular Value Decomposition (SVD) is applied to the training set of residuals first in the row direction and then, in the column direction. A fixed-point approximation of the transform matrices to reduce computational cost is used.

Adaptive coefficient scanning

After applying separable directional transform, the 2-D transform coefficient matrix is converted to 1-D. In AVC, zigzag scanning order is used for this purpose. In the case of MDDT, even after separable directional transforms are applied, the resulting 2-D transform coefficient matrix still carries some directionality. For example, consider

vertical prediction (mode 0). After prediction, transform and quantization, the nonzero coefficients tend to exist along the horizontal direction. By using coefficient scanning oriented in the horizontal direction instead of the zigzag scanning, the non-zero coefficients in the 2D matrix can be further compacted toward the beginning of the 1-D vector. This in turn improves entropy coding efficiency. Quantized transform coefficients of different prediction modes carry different statistics. Therefore, for each mode, adaptive coefficient scanning is used. This is accomplished as follows:

6. At the beginning of each video slice, initialize the coefficient scanning order for each prediction mode;
7. When a block is entropy-coded, for each non-zero coefficient coded, increment the count at the corresponding position by one;
8. After each macroblock is coded, update the coefficient scanning order according to the statistics collected;
9. Normalize the collected statistics if needed;
10. Use the updated order for coding of future blocks. Go back to 2 until slice is finished.

11.4 Transform Selection

For the luma component of the inter residual signal, two switchable transforms can be adaptively selected for each block size. Therefore, the selected transform can better fit the residue data. The selection is between the DCT and a KLT using a rate-distortion criterion. For transform sizes smaller than 16x16, no explicit flag is sent to indicate which transform is used for each block. Instead, the flag is incorporated within the transform coefficients themselves to reduce overhead. At the decoder, the transform coefficients are first recovered and the parity of their sum is computed. If the parity is even, the inverse DCT is applied. If the parity is odd, the inverse KLT is applied. When the transform size is 16x16 and above, an explicit flag is used to indicate which transform to apply.

For a residual block of size NxN, a separable KLT transform is used, which can be described as:

$$Y = C_{\text{inter}} X C^{T}_{\text{inter}},$$

where X, Y, C_{inter}, are of size NxN.

The KLT basis functions are derived from the statistics of the inter prediction residuals from a representative set of sequences. The offline training is done so that the KLT complements the DCT. The transform is implemented in integer arithmetic and the norms are forced to be the same as those of the DCT. In this way, the quantization tables and entropy coding of the DCT can be reused. Also, the first basis vector of the KLT is forced to be constant, so it is consistent with the first basis vector of the DCT, which improves subjective quality at low bit-rates.

The scanning order of the KLT coefficients is the same as for the DCT coefficients.

11.5 Quantization

The basic principle for quantization and de-quantization of coefficients for large transforms is the same as that used in AVC, i.e. a scalar quantizer with dead-zone.

12 Deblocking filter

Deblocking filter operates across 8x8 block edges for luma as well as for chroma blocks.

$p3_0$ $p2_0$ $p1_0$ $p0_0$	$q0_0$ $q1_0$ $q2_0$ $q3_0$
$p3_1$ $p2_1$ $p1_1$ $p0_1$	$q0_1$ $q1_1$ $q2_1$ $q3_1$
$p3_2$ $p2_2$ $p1_2$ $p0_2$	$q0_2$ $q1_2$ $q2_2$ $q3_2$
A $p3_3$ $p2_3$ $p1_3$ $p0_3$	$q0_3$ $q1_3$ $q2_3$ $q3_3$ **B**
$P3_4$ $p2_4$ $p1_4$ $p0_4$	$q0_4$ $q1_4$ $q2_4$ $q3_4$
$P3_5$ $p2_5$ $p1_5$ $p0_5$	$q0_5$ $q1_5$ $q2_5$ $q3_5$
$P3_6$ $p2_6$ $p1_6$ $p0_6$	$q0_6$ $q1_6$ $q2_6$ $q3_6$
$P3_7$ $p2_7$ $p1_7$ $p0_7$	$q0_7$ $q1_7$ $q2_7$ $q3_7$

Notation of Block Edge for Deblocking

Luma filtering

The luma edge between blocks A and B can be filtered if one of the following conditions is true

- Block A or Block B has mbMode==INTRA
- ***or*** Block A or block B has nonzero transform coefficients
- ***or*** The absolute difference between the horizontal or vertical component of the motion vectors used for Block A and Block B is greater than or equal to 4 in units of quarter luma samples**,**
- ***or*** a PBIC value (icScale or icOffset) of Block A differs from the corresponding value of Block B
- ***and*** $|p2_2 - 2*p1_2 + p0_2| + |q2_2 - 2*q1_2 + q0_2| + |p2_5 - 2*p1_5 + p0_5| + |q2_5 - 2*q1_5 + q0_5| < \beta$

The luma edge between blocks A and B is ***not*** filtered if one of the following conditions is true.

- Block A and Block B belong to the same macroblock ***and*** mb_svt_flag==1
- ***or*** Block A and Block B belong to the same macroblock ***and*** (mbMode==INTER_16x16 or mbMode==INTRA) ***and*** transform 16x16 is used
- ***or*** the edge is an internal edge of a large macroblock LMB32
- ***or*** the edge is an internal edge of a large macroblock LMB64
- ***or*** the edge is a macroblock border ***and*** both Block A and Block B have planar_flag ==1.

If the edge between Block A and Block B is filtered, one of two types of filtering (weak or strong filtering) is performed. The choice between the strong and the weak filtering is done separately for each line depending on the following conditions. For each line i = 0,7, the strong filtering is performed if all the following conditions are true, otherwise, weak filtering is performed.

- $d < (\beta>>2)$
- ***and*** $(|p3_i - p0_i| + |q0_i - q3_i|) < (\beta>>3)$
- ***and*** $|p0_i - q0_i| < ((5*t_C + 1)>>1)$,

where t_C and β depends on qp their relations are shown in Table 1. t_C is increased by 4 when one of Block A or Block B has mbMode==INTRA.

Table 1 Corr_qpc, tc and β

qp	0	1	2	3	4	5	6	7	8	9	10	11	13	13	14	15	16	17
Corr_qpc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t_c	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2
B	0	0	0	0	0	0	6	7	8	9	10	11	12	13	14	15	16	17
qp	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Corr_qpc	0	0	1	1	1	1	2	2	2	3	3	4	4	5	5	6	6	6
tc	2	2	2	3	3	3	3	4	4	4	5	5	6	6	7	8	9	9
B	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52
qp	36	37	38	39	40	41	42	43	44	45								
Corr_qpc	7	7	7	8	8	8												
tc	10	10	11	11	12	12	13	13	14	14								
B	54	56	58	60	62	64												

Weak filtering
Weak filtering mode filtering is performed based on the above conditions,

$\Delta = Clip(-t_C, t_C, (13*(q0_i - p0_i) + 4*(q1_i - p1_i) - 5*(q2_i - p2_i) + 16)>>5))$ $i = 0,7$
$p0_i = Clip_{0\text{-}255}(p0_i + \Delta)$ $i = 0,7$
$q0_i = Clip_{0\text{-}255}(q0_i - \Delta)$ $i = 0,7$
$p1_i = Clip_{0\text{-}255}(p1_i + \Delta/2)$ $i = 0,7$
$q1_i = Clip_{0\text{-}255}(q1_i - \Delta/2)$ $i = 0,7$

Strong filtering
Strong filtering mode is performed with the following set of operations.

$p0_i = Clip_{0\text{-}255}((p2_i + 2*p1_i + 2*p0_i + 2*q0_i + q1_i + 4)>>3);$ $i = 0,7$
$q0_i = Clip_{0\text{-}255}((p1_i + 2*p0_i + 2*q0_i + 2*q1_i + q2_i + 4)>>3);$ $i = 0,7$
$p1_i = Clip_{0\text{-}255}((p2_i + p1_i + p0_i + q0_i + 2)>>2);$ $i = 0,7$
$q1_i = Clip_{0\text{-}255}((p0_i + q0_i + q1_i + q2_i + 2)>>2);$ $i = 0,7$
$p2_i = Clip_{0\text{-}255}((2*p3_i + 3*p2_i + p1_i + p0_i + q0_i + 4)>>3);$ $i = 0,7$
$q2_i = Clip_{0\text{-}255}((p0_i + q0_i + q1_i + 3*q2_i + 2*q3_i + 4)>>3);$ $i = 0,7$

Chroma filtering

The chroma edge between blocks A and B can be filtered if the following condition is true.

- Block A or Block B has mbMode==INTRA

The filtering is not applied if both following conditions are true.

- The edge is a macroblock edge

and both Block A and Block B have planar_flag==1.

If filtering is performed based on the above condition,

$\Delta = Clip(-t_C, t_C, ((((q0_i - p0_i) << 2) + p1_i - q1_i + 4) >> 3))$ $i = 0,7$
$p0_i = Clip_{0\text{-}255}(p0_i + \Delta)$ $i = 0,7$
$q0_i = Clip_{0\text{-}255}(q0_i - \Delta)$ $i = 0,7$

t_C depend on qp and their relations are shown in Table 1. For chroma filtering t_C is increased by 4.

Planar mode filtering

There is a special filtering mode which is applied to the macroblock edge when both macroblocks are encoded in intra planar mode. Luma and chroma edges between blocks A and B are filtered with *planar filtering* if all of the following conditions are true, otherwise normal filtering described in 0 and 0 are performed.

- Edge between Macroblock A and Macroblock B is a macroblock edge
- Both Macroblock A ***and*** Macroblock B have planar_flag==1

Planar luma filtering
The planar mode filtering for luma component is performed with the following set of operations.

$p3_i = (7*p3_i + 3)/7$ $i = 0,15$
$p2_i = (6*p3_i + q3_i + 3)/7$ $i = 0,15$
$p1_i = (5*p3_i + 2*q3_i + 3)/7$ $i = 0,15$
$p0_i = (4*p3_i + 3*q3_i + 3)/7$ $i = 0,15$
$q0_i = (3*p3_i + 4*q3_i + 3)/7$ $i = 0,15$
$q1_i = (2*p3_i + 5*q3_i + 3)/7$ $i = 0,15$
$q2_i = (p3_i + 6*q3_i + 3)/7$ $i = 0,15$
$q3_i = (7*q3_i + 3)/7$ $i = 0,15$

Planar chroma filtering
The planar mode filtering for chroma component is performed with the following set of operations.

$p1_i = (4*p1_i + 2)/4$ i = 0,7
$p0_i = (3*p1_i + q1_i + 2)/4$ i = 0,7
q0i = (2*p1i + 2q1i + 2)/4 i = 0,7
q1i = (p1i + 3*q1i + 2)/4 i = 0,7

13 In-loop filtering

The purpose of the adaptive loop filtering (ALF) process is to further reduce the distortion between the original picture and the reconstructed picture caused by complex lossy coding. Filters minimizing the distortion for both luma and chroma components are calculated using the Wiener filter approach. Based on the prediction signal $\hat{s}$, the quantized prediction error e', and the reconstructed filter signal s'', the following adaptive loop filter operation is performed:

$$s''' = \sum_{i=1}^{M} w_i \cdot s''_i + \sum_{i=1}^{N} w_{M+i} \cdot \hat{s}_i + \sum_{i=1}^{O} w_{M+N+i} \cdot e'_i + c$$

After filters are applied in a frame or coding units, filter coefficients are explicitly sent in the bitstream. This adaptive loop filter increases reconstructed picture quality as well as reference picture quality for the next picture coding. The filtering control basis is usually obtained independently from the prediction structure, which results in a large amount of information being sent to the decoder and its inability to access prediction information in its process.

In this text, Coding Block Tree (CBT) synchronized ALF is presented. By utilizing the coding unit structure, the amount of control information to be sent is reduced. Due to the flexible coding unit structure from large to small block, coding unit boundary provides great flexibility to be used as control basis in the adaptive loop filter process. Since the optimal coding unit boundary obtained from previous coding stages reflects objects' boundaries having similar properties, like prediction and quantization error, a simple control scheme which infers filter control structure from the coding unit structure and the optimal control depth reduces the complexity of the optimal control map while achieving similar performance.

The filter control information is signaled in the slice header in order to avoid re-multiplexing of the information at the CU level coded data.

Once the control units for filtering are determined, a control flag indicating whether filtering is applied is sent to maximize filter performance and reduce complexity of filter application. Depending on the largest block size, the maximum depth, sequence property and coding parameters, the number of coding units can be too many to be used as filter control units itself. Thus the optimal control depth is sent to indicate the maximum depth (i.e., the smallest size) of a coding unit boundary to be used as filter control map.

The text uses M set of non-separable/separable filters. The M sets of non-separable/separable filters is transmitted to the decoder for each frame. Whenever the ALF segmentation map indicates that a block should be filtered, for each pixel, a specific set of filters is chosen based on a measure of local characteristic of an image, called activity measure. The text uses the sum-modified Laplacian measure. The sum-modified Laplacian for pixel is calculated as follows:

$$\mathrm{var}(i,j) = \sum_{k=-K}^{K} \sum_{l=-L}^{L} \left|2R(i+k,j+l) - R(i+k-1,j+l) - R(i+k+1,j+l)\right| + \left|2R(i+k,j+l) - R(i+k,j+l-1) - R(i+k,j+l+1)\right| .$$

14 Entropy Coding

14.1 Low-Complexity Entropy Coding

General

All syntax elements are encoded using either FLC or "UVLC-like" VLC tables. Each syntax element use one out of 10 pre-determined VLC tables that are designed for different probability distributions. All VLC tables are organized in the conventional way with the shortest binary codeword at the beginning.

Encoding of a parameter/event using a VLC table is typically done in three steps:

1) Convert the parameter/event value to a *table index* by using some enumeration scheme.
2) Use the *table index* to generate a *code number* through lookup in a *sorting table.* The purpose of the *sorting table* is to assign *code numbers* according to increasing probability so that parameters/events with the high probability are assigned a code number with a low value.
3) Use the *code number* to generate a *binary codeword* by lookup in the pre-determined VLC table.

The VLC encoding process is illustrated in Figure 5, while the VLC decoding process is illustrated in Figure 6.

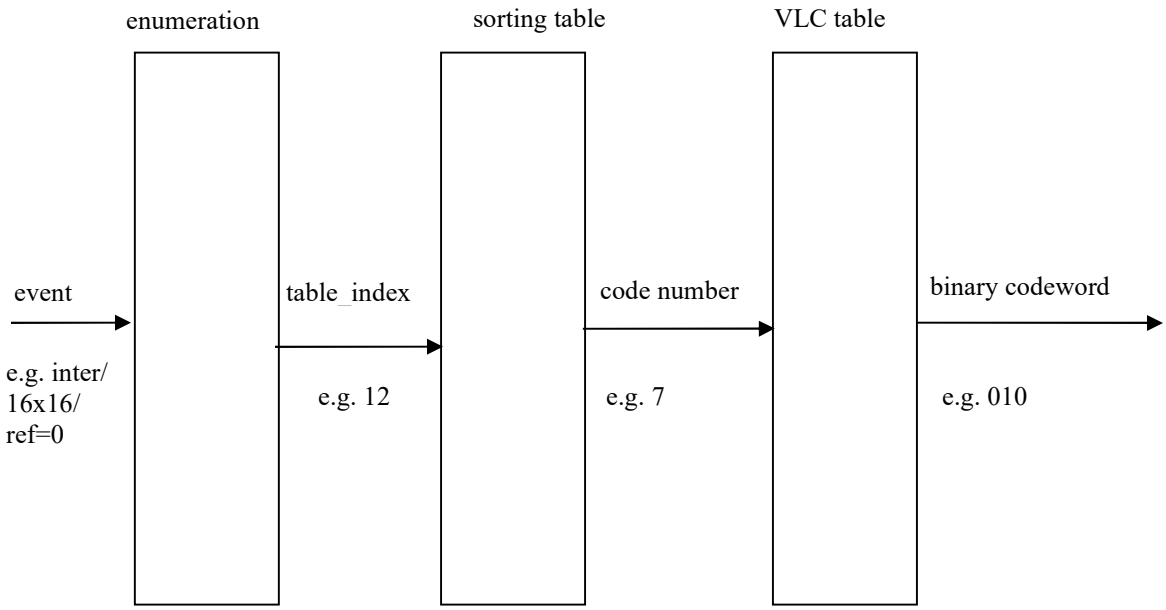


Figure 9 VLC encoding process

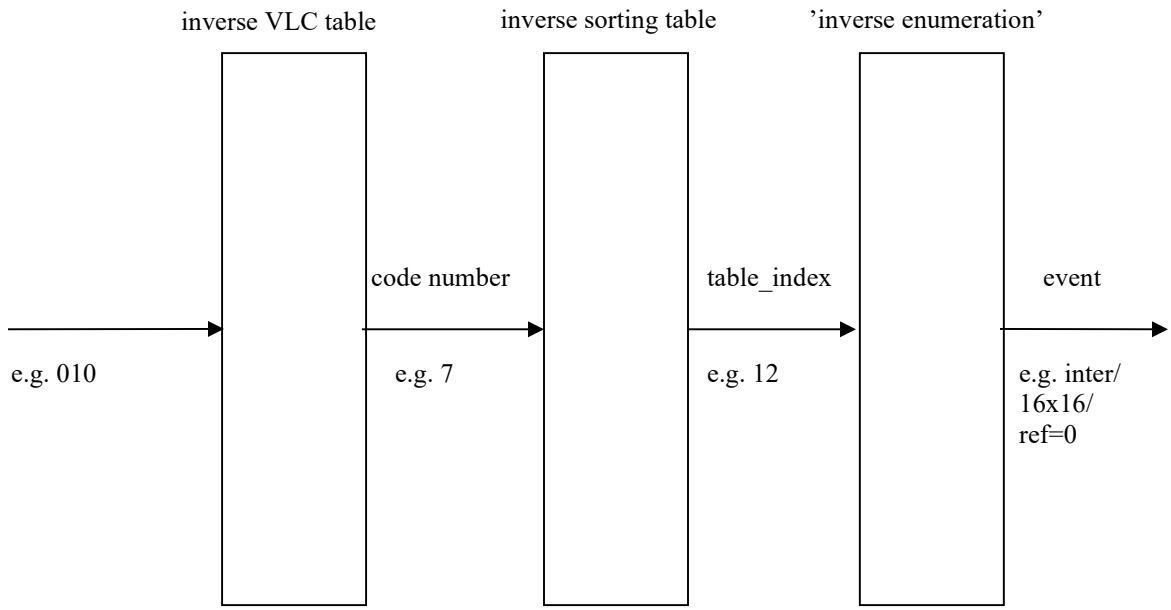


Figure 10 VLC decoding process

Adaptivity

Two different forms of adaptivity are applied to VLC encoding:

1) For some parameters/events, the particular VLC table to be used depends on the value of previously encoded parameters/events. A context variable is maintained that indicates expected probability distribution of the events. This variable is updated as the events are encoded and it is used to look-up VLC table number.

2) For some parameters/events, the *sorting table* is made adaptive. This process is most easily described in terms of *decoder* operations as follows: For each binary codeword that is decoded into a *code number*, determine the corresponding *table index* by lookup into an *inverse sorting table*. Next, swap that entry in the inverse *sorting table* with the entry immediately above. This mechanism ensures that a *table index* value that occurs frequently eventually propagates towards the top of the inverse sorting *table*, corresponding to the most likely value and the shortest *binary codeword*. On the encoder side, both a sorting table and the corresponding inverse sorting table need to be updated simultaneously. Updating the inverse sorting table on the decoder side is illustrated in Figure 7.

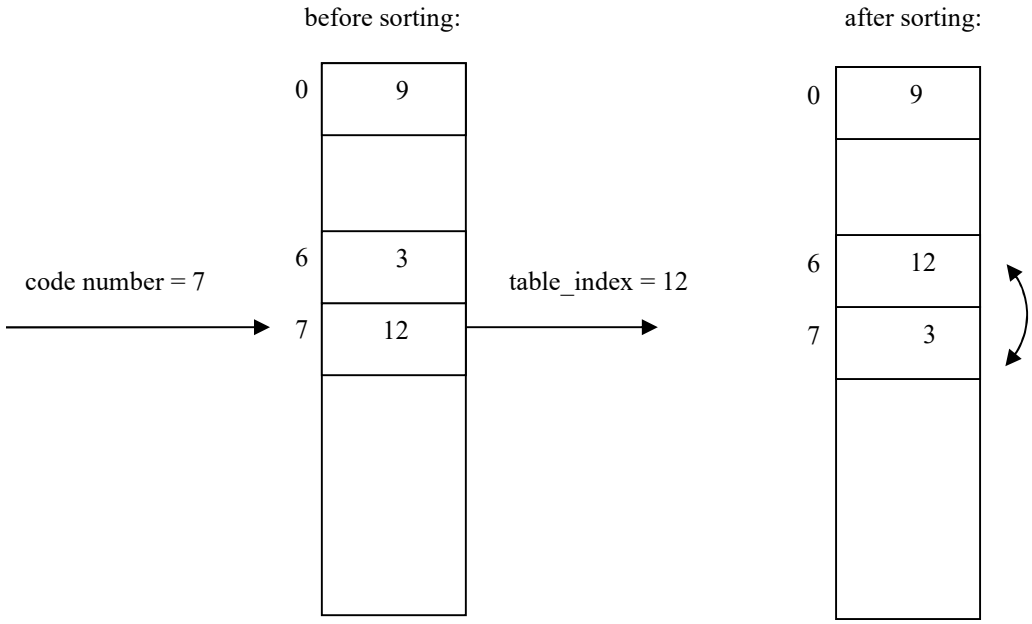


Figure 11 Update of inverse sorting table - example

Transform coefficients

Entropy coding of transform coefficients is performed by an improved version of CAVLC as follows:

1) Organize the quantized transform coefficients, each represented by a level and sign in a 1D array using the conventional zig-zag scan.

2) Coefficients are encoded backwards along the 1D array from the most high frequency coefficient towards the DC coefficients.

3) Each event (to be described below) is encoded using an "UVLC-like" table. Different VLC tables are used for different events.

4) First, the last significant coefficient (last_coeff) is coded in two steps:

i. Code the position (in backward scan order) of last_coeff (last_pos) and a flag (levelFlag = level>1) as a combined event.

ii. If level>1 code the value of level as a single event

iii. Code the sign as one bit

5) If level==1, continue encoding non-zero levels in run-mode.

6) If level>1, switch to level-mode if any of the following is true:

i. Cumulative sum of all level of coefficients coded so far >1 (excluding the level of last_coeff), is bigger than a threshold

ii. The position of the current level is bigger than the threshold position

7) In *run-mode*, encode each non-zero coefficient using the preceding run and levelFlag as a combined event. Then encode the sign and level (if levelFlag==1).

8) In level-mode, encode each coefficient (including zero coefficients) as level and sign.

For the combined last_coeff/levelFlag event, an *adaptive sorting table* is used.

Side information

The following parameters are coded as a combined event for each macroblock:

- macroblock mode
- motion vector partition
- transform size
- macroblock level CBP information

- reference frame index for P frames
- dqp flag (indicating whether or not dqp is signalled for that MB)

For this combined event, an *adaptive sorting table* is used.

The following side information parameters are encoded separately as single events:

- block level CBP
- motion vector differences
- reference frame index for B frames
- intra prediction modes

Block level CBP is encoded using an *adaptive sorting table*.

14.2 High Coding Efficiency Entropy Coding

For entropy coding, a variation of CABAC (as found in AVC) is employed. The binarization and context modeling are basically the same as in CABAC of AVC, except from a few modifications and additions as further explained below. However, the actual coding of the bins is based on a novel concept that has been introduced in order to support parallelized implementations of entropy encoding and decoding with load balancing.

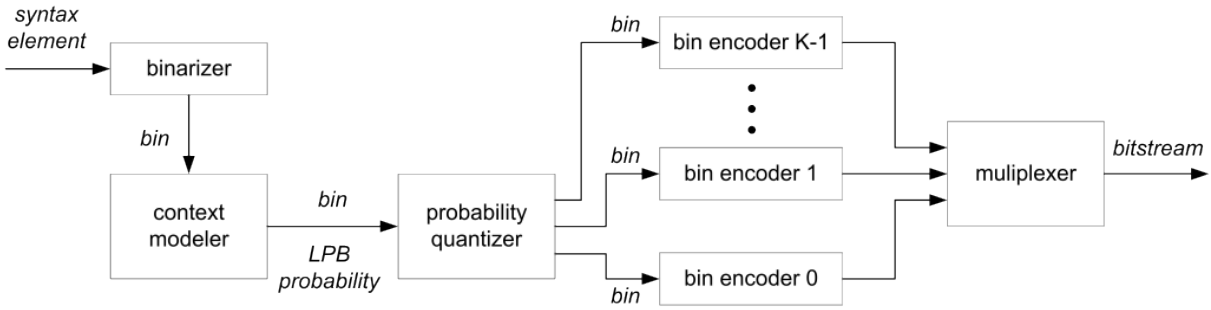


Figure 10: Parallel V2V entropy coding

14.2.1 Novel entropy coding concept

In Fig. 9, the basic entropy coding concept is illustrated. If a syntax element does not already represent a binary syntax element, it is first binarized, i.e., it is mapped onto a sequence of binary decisions (bins). For each bin a context is selected. A context represents a (binary) probability model for a class of bins; it is characterized by the probability and the value of the less probable bin (LPB). As in AVC, the LPB probability is represented by one out of 64 states. At the beginning of the encoding/decoding of a slice, the probability models are initialized using fixed values (as in CABAC of AVC). And after encoding/decoding a bin with a particular model, the model (i.e., the corresponding probability and value of the LPB) is updated. The probability model update, i.e., the probability estimation process is the same as in CABAC of AVC. The association of a bin with a context model is also similar as in CABAC of AVC. It depends on the syntax element, the bin number, and, for some bins, the values of neighboring syntax elements.

The above described binarization and association of the bins with context models is basically the same as in CABAC of AVC. The main difference is the next step. Instead of arithmetically coding the bins with the associated probability models using a single arithmetic coding engine as in CABAC of AVC, the estimated LPB probabilities are quantized, i.e., they are mapped onto a small number of LPB probability intervals and for each of these probability intervals a separate bin encoder/decoder is operated. In one implementation, we used 12 probability intervals and thus 12 different bin encoders/decoders. Each bin encoder/decoder operates at a fixed LPB probability, which can be considered as the representative probability for an LPB interval. The selection of a bin encoder/decoder is implemented via a look-up table that associates a bin encoder with each of the 64 state indices for the LPB probability. Hence, the 64 states that are used for estimating the LPB probability of a context model are mapped onto the selected (e.g. 12 in one of the proposed implementation) probability classes, for each of which a separate bin encoder/decoder is operated. The partial bitstreams that are generated by the bin encoders can be written to different partitions of a NAL unit (with the partitioning of the NAL unit being indicated in the slice header) or they can be interleaved into a single bitstream.

The bin encoders and decoders represent simple variable length bin to variable length (V2V) codes, by which a variable number of bins is mapped onto variable length codewords and vice versa.

In the case where parallel processing at the decoder is desired, a load balancing scheme is provided to distribute the computational load onto multiple bin decoders evenly.

14.2.2 V2V codes

For the bin encoders (cf. 9), V2V codes are employed. It should be again noted that each bin encoder operates at a fixed LPB probability. And for a fixed LPB probability, we can simply construct a code that maps a variable number of bins onto a variable length codeword and vice versa. As a particular example, the following table shows an example for such a mapping. In the example the bins "0" represent the more probable bin values and the bins "1" represent the less probable bin values. The example code has been designed for a LPB probability of 0.15 and it yields a redundancy of only 0.25 % relative to the entropy limit for this probability. Generally, it is possible to get closer to the entropy limit when the table size is increased.

Table 1: Example for a mapping between a variable number of bins and variable length codewords

Sequence of bins (bin order is from left to right)	codewords (bits order is from left to right)
0000	1
01	001
10	010
001	011
0001 00	0001
11	0000 1
0001 1	0000 00
0001 01	0000 01

At the end of a slice or due to a flush event (e.g. buffer full), it is possible that a bin sequence does not represent any bin sequence that is associated with a codeword, that is the bin sequence sits at an internal node of the V2V tree. In order to terminate the codeword, the shortest codeword for a bin sequence that contains the current bin sequence as prefix is written. At the decoder side, the additionally decoded bins are discarded. As an example, we could use the mapping in the above table and have a bin sequence "00" at the end of a slice. Then we would write the codeword "1" (corresponding to the bin sequence "0000"). At the decoder side, the codeword "1" would be translated into the bin sequence "0000", but the decoder would discard (ignore) the last two decoded bins.

The partial bitstreams can be written to different partitions of the slice data NAL unit (with transmitting the partitioning information in the slice header). It is also possible to combine the partial bitstreams into one single bitstream. Let L_k be the encoded length of the kth partial bitstream, where $0 <= k < N$, where N=StateCount. We define a prefix code C the following way:

If $n < 128$, then $C(n) = n << 1$;
Else if $n < 16512$, then $C(n) = ((n - 128) << 2) | 1$;
Else if $n < 2113664$, then $C(n) = ((n - 16512) << 3) | 3$;
Else $C(n) = ((n - 2113664) << 3) | 7$;

where "<<" is a right shift, and "|" is a bit-wise OR.

The output of the parallel framework is the concatenation of the following:

1. [Header] $C(L_k)$ for $0 <= k < N$,
2. [Payload] The encoded sequence for bitstream k ($0 <= k < N$).

The decoder first decodes the L_k values from the header, and uses these values to find the starting point of the encoded sequence k within the payload. After this, the N bitstreams can be decoded independently of each other from the payload, hence parallel decoding is feasible.

It is also possible to interleave the codewords from different partial bitstreams. This is in particular beneficial for small slice data NAL units, for which parallelized bin decoding is not required. In order to realize such an interleaving, a first-in-first-out codeword buffer for variable-length codewords can be introduced at the encoder side. When a particular bin encoder starts the encoding of a new bin sequence (i.e., the bin encoder receives the first bin

after having written the previous codeword), it reserves the next variable-length codeword entry in the codeword buffer. When the received bins represent a bin sequence that corresponds to a codeword, this codeword is written to the previously reserved codeword entry, that is, output in a first-in-first-out order. Examples for the codeword buffer status are illustrated in Fig. 8. At the decoder side, the codewords can be directly read from the bitstream in decoding order.

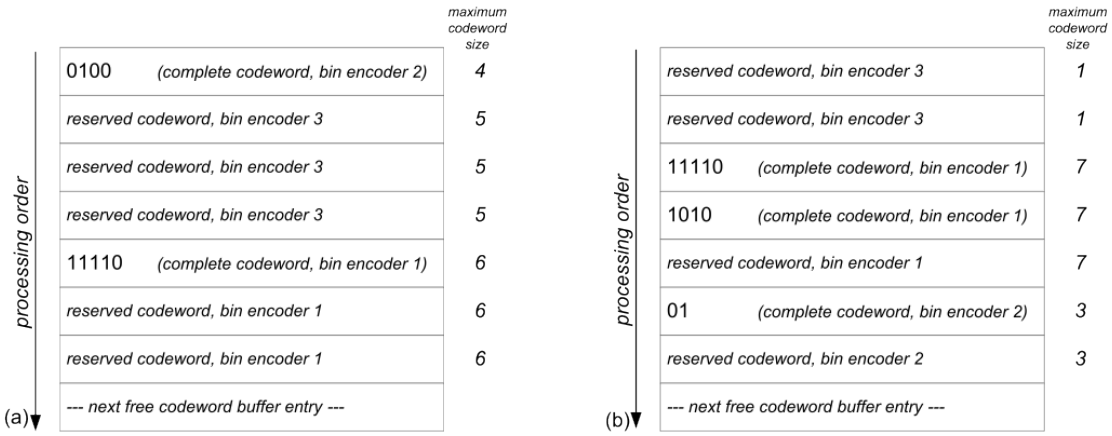


Figure 11: Interleaving codewords by using a first-in-first-out codeword buffer

It is also possible to combine the codeword interleaving with a low-delay buffer control. That means, we can suitably restrict the buffer delay that is introduced by the codeword interleaving. With such a low-delay control, a particular codeword is terminated when the maximum number of bits (this maximum number of bits represents the maximum codeword length for a particular bin encoder and is given by the corresponding mapping table) that are associated with the reserved or completed buffer entries following the buffer entry for the particular codeword is greater than a particular threshold. At the decoder side, the extra decoded bins must be discarded; this can be done by using the same measure of the maximum number of bits as at the encoder side.

~~14.2.3Encoding process for load balancing in parallel V2V (Informative)~~

~~[Ed.: Would this section be better to be moved to an Informative Annex?]~~

~~This section describes the load balancing of bins for an arbitrary (possibly unknown) number of bin decoders available in the parallel framework described in Figure 9 above. In this section all length values are expressed in bits, unless stated otherwise.~~

~~The parallel V2V encoding process creates the encoded bit streams S'_k of length M'_k for each of the input bin symbol sequence S_k, k=0,…,N, with the given entropy coder, where N=mergeStateCount.~~

~~The output of the load balancing extension to the parallel framework is the concatenation of the following:~~

- ~~[First part of the header] $C(M'_k)$ for k = 0,…,N,~~
- ~~[Second part of the header,] auxiliary offset information~~
- ~~[Payload] S'_k (k=0,…,63)~~

~~The byte-align prefix code C() is the same as defined in Section 5.1.16.1.~~

~~The format and content of the second part of the header depends on the V2V codes used to encode the S_k sequences, and a distributor function. This part is a self-delimiting sequence.~~

~~Let D(a, b) be the set of all $(v_1,\ldots,v_b)$ vectors, for which v_k is a positive integer (k=1,…,b), and $v_1 + v_2 + \ldots + v_b = a$. Let $S' = S'_0 + \ldots + S'_N$. Then, using the notations above, the distributor function is a function of the form F : {0,1}^K -> D(K, b). The distributor partitions the concatenated encoded sequences into b units, such that decoding each unit would take approximately the same time.~~

~~In this form, the distribution function gives the lengths of the decodable segments. The starting positions of these segments are given by the associated function F'(x), where for F(x)= $(v_1,\ldots,v_b)$, F'(x)=$(0, v_1, v_1+v_2, v_1+v_2+v_3, \ldots, v_1+\ldots+v_b)$. Obviously, the functions F and F' mutually determine each other.~~

~~Example: the uniform distributor assumes that the decoding complexity of any S'_k depends only on M'_k, that is, the length of S'_k. Then, for the concatenated encoded bit stream x of length S', F(x) = (S'/b, S'/b, …, S'/b) (the division may be rounded up or down to meet the final sum $M'_0+\ldots+M'_N$). Example: the weighted distributor assumes that the decoding complexity of any S'_k is uniform within S'_k, but may vary according to k. Let w_k be this complexity for S'_k, and $W=w_0*M'_0 + \ldots + w_{63}*M'_{63}$. Then the distributor assigns the values F'(x) = $(v_1,\ldots,v_b)$ such that for any k < b: the decoding complexity of the segment $x[v_k,\ldots,v_{k+1}-1]$ is W/b.~~

~~The format of the second part of the header is as follow:~~

~~Consider a V2V bin encoder. For all k=1,…,b-1, let B_k be the number of bits, which we have to read ahead from F'(x)[k] to arrive at the start of the next codeword. That is, B_k=0 if a codeword starts right at F'(x)[k], B_k=1 if the next codeword starts at F'(x)[k+1], and so on. Then Part 2 of the header is the concatenation of the values $C(B_k)$ for k=1,…,b-1.~~

~~It is possible to achieve load balancing for any specific set of b segments and d bin decoders; however, some sets obviously perform better than others. The best performance among all (b, d) values is reached when b is a multiple of d.~~

~~Example: If d divides b, then the load balancing algorithm simplifies to G_k=b/d for all k=1,…,d.~~

~~The general method works with the case when the encoder has no information about the number d of bin decoders at the side of the decoder. In the case that the encoder has a knowledge of some potential candidates $d_1, d_2, \ldots, d_k$ for the number of bin decoders, we present the following improvement: let b be a number which is a multiple of all of $d_1, \ldots, d_k$ (the least common multiple of the numbers $d_1, \ldots, d_k$ is adequate). In this case, for all the candidates the load distribution algorithm simplifies to the previous example, while for all other bin decoder numbers it resorts to the general algorithm.~~

~~Example: Suppose that the encoder knows that target decoders may be equipped with any one of 1, 2, 3, 4, 6, 8, 10, 12 or 16 bin decoders. The least common multiple of these numbers is 240, so the encoder should set b=240.~~

~~Example: If the encoder sets $b=2^t$, then optimal load balancing is achieved for each of 1, 2, 4, 8, …, 2^t bin decoders.~~

~~14.2.4~~14.2.3 Binarization and context modeling

[Ed. put into parsing process of context adaptive entropy coding]
As already mentioned above, we have reused the binarization and context modeling schemes of AVC CABAC for most syntax elements in this video coding algorithm, especially for those which have already been present in AVC like, e.g., syntax elements related to intra prediction modes or motion parameters. For entropy coding of novel syntax elements like, e.g., flags indicating the quadtree structure, suitable context models have been designed. In addition, for coding of syntax elements related to transform coefficients of larger block sizes, an improved context modeling scheme has been employed.

~~14.2.5~~14.2.4 Improved context modeling scheme for coding of transform coefficients

The novel context modeling scheme applies to the syntax elements significant_coeff_flag, last_significant_coeff_flag, and coeff_abs_level_minus_one of blocks of size 8x8 and larger, and it is complemented by an adaptive switching of the scan during encoding/decoding of the significance map (specifying the locations of non-zero transform coefficient levels).

Coding of the significance map

Coding of the syntax elements significant_coeff_flag and the last_significant_coeff_flag, as constituting the significance map, is improved by an adaptive scan and a new context modeling based on a defined neighborhood of already coded scan positions. This turned out to be beneficial especially for large block sizes.

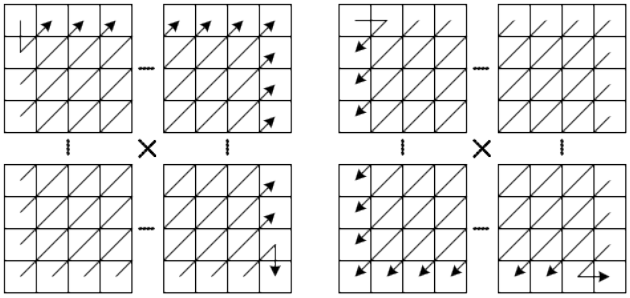


Fig. 9 Illustration of the two scanning patterns for the significance map

Scanning order

The scanning order for coding the significance map is adapted by switching between two predefined scan patterns. For the first scanning pattern, the diagonal sub-scans are scanned from bottomleft to topright (left illustration of Fig. 9), and for the second scanning pattern, the diagonal subscans are scanned from topright to bottomleft (right illustration of Fig. 9). The coding of the significance map starts with the second scanning pattern. While coding the syntax elements, the number of significant transform coefficient values is counted by two counters c_1 and c_2. The first counter c_1 counts the number of significant transform coefficients that are located in the bottom-left part of the transform block; i.e., this counter is incremented by one when a significant transform coefficient level is coded/decoded for which the horizontal coordinate x inside the transform block is less than the vertical coordinate y. The second counter c_2 counts the number of significant transform coefficients that are located in the top-right part of the transform block; i.e., this counter is incremented by one when a significant transform coefficient level is coded/decoded for which the horizontal coordinate x inside the transform block is greater than the vertical coordinate y. At the end of each diagonal sub-scan, it is decided whether the first or the second of the predefined scanning patterns is used for the next diagonal sub-scan. This decision is based on the values of the counters c_1 and c_2. When the counter for the bottom-left part of the transform block is greater than the counter for the bottom-left part, the scanning pattern that scans the diagonal sub-scans from bottom-left to top-right is used; otherwise (the counter for the bottom-left part of the transform block is less than or equal to the counter for the bottom-left part), the scanning pattern that scans the diagonal sub-scans from topright to bottom-left is used.

Context modeling

For 4x4 blocks, the context modeling for the syntax elements significant_coeff_flag is done as specified in AVC. For 8x8 blocks, the transform block is decomposed into 16 sub-blocks of 2x2 samples, and each of these sub-blocks is associated with a separate context. The context model selection for larger transform blocks (e.g., for blocks greater than 8x8) is based on the number of already coded significant transform coefficients in a predefined neighborhood (inside the transform block). For coding of the last_significant_coeff_flag, a context modeling has been designed that depends on a distance measure of the current scan position to the top-left corner of the given transform block. To be more specific, the context model for coding the last_significant_coeff_flag is chosen based on the scan diagonal on which the current scan position lies (i.e., it is chosen based on x + y, where x and y represent the horizontal and vertical location of a scan position inside the transform block, respectively). To avoid overfitting, the distance measure x + y is mapped on a reduced set of context models in a certain way (e.g. by quantizing x + y).

Coding of absolute values of transform coefficient levels

The coding process for absolute transform coefficient levels maps each quadratic (or rectangular) block of size 8x8 and larger onto an ordered set (vector) of 4x4 sub-blocks by using a forward zig-zag scan; while the transform coefficient levels inside a sub-block are processed in a reverse zig-zag scan. Following the handling of 4x4 blocks in AVC CABAC, the context model set for each sub-block consists of two times five context models with five models for both the first bin and all remaining bins (up to and including the 14. bin) of the coeff_abs_level_minus_one syntax element, where the selection of context models is done exactly as in the original CABAC. However, as a novel feature, different sub-blocks may select different sets of context models, where the choice of the context model set for a sub-block depends on certain statistics of one or more already coded sub-blocks.

AAnnex B
Byte stream format

This annex specifies syntax and semantics of a byte stream format specified for use by applications that deliver some or all of the NAL unit stream as an ordered stream of bytes or bits within which the locations of NAL unit boundaries need to be identifiable from patterns in the data, such as ITU-T Rec. H.222.0 | ISO/IEC 13818-1 systems or ITU-T Rec. H.320 systems. For bit-oriented delivery, the bit order for the byte stream format is specified to start with the MSB of the first byte, proceed to the LSB of the first byte, followed by the MSB of the second byte, etc.

The byte stream format consists of a sequence of byte stream NAL unit syntax structures. Each byte stream NAL unit syntax structure contains one start code prefix followed by one nal_unit(NumBytesInNALunit) syntax structure. It may (and under some circumstances, it shall) also contain an additional zero_byte syntax element. It may also contain one or more additional trailing_zero_8bits syntax elements. When it is the first byte stream NAL unit in the bitstream, it may also contain one or more additional leading_zero_8bits syntax elements.

A.1 Byte stream NAL unit syntax and semantics

A.1.1 Byte stream NAL unit syntax

byte_stream_nal_unit(NumBytesInNALunit) {	C	Descriptor
while(next_bits(24) != 0x000001 && next_bits(32) != 0x00000001)		
leading_zero_8bits /* equal to 0x00 */		f(8)
if(next_bits(24) != 0x000001)		
zero_byte /* equal to 0x00 */		f(8)
start_code_prefix_one_3bytes /* equal to 0x000001 */		f(24)
nal_unit(NumBytesInNALunit)		
while(more_data_in_byte_stream( ) && next_bits(24) != 0x000001 && next_bits(32) != 0x00000001)		
trailing_zero_8bits /* equal to 0x00 */		f(8)
}		

A.1.2 Byte stream NAL unit semantics

The order of byte stream NAL units in the byte stream shall follow the decoding order of the NAL units contained in the byte stream NAL units. The content of each byte stream NAL unit is associated with the same access unit as the NAL unit contained in the byte stream NAL unit.

leading_zero_8bits is a byte equal to 0x00.

zero_byte is a single byte equal to 0x00.

When any of the following conditions are fulfilled, the zero_byte syntax element shall be present:

– the nal_unit_type within the nal_unit( ) is equal to 7 (sequence parameter set) or 8 (picture parameter set),

– the byte stream NAL unit syntax structure contains the first NAL unit of an access unit in decoding order.

start_code_prefix_one_3bytes is a fixed-value sequence of 3 bytes equal to 0x000001. This syntax element is called a start code prefix.

trailing_zero_8bits is a byte equal to 0x00.

A.2 Byte stream NAL unit decoding process

Input to this process consists of an ordered stream of bytes consisting of a sequence of byte stream NAL unit syntax structures.

Output of this process consists of a sequence of NAL unit syntax structures.

At the beginning of the decoding process, the decoder initialises its current position in the byte stream to the beginning of the byte stream. It then extracts and discards each leading_zero_8bits syntax element (if present), moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next four bytes in the bitstream form the four-byte sequence 0x00000001.

The decoder then performs the following step-wise process repeatedly to extract and decode each NAL unit syntax structure in the byte stream until the end of the byte stream has been encountered (as determined by unspecified means) and the last NAL unit in the byte stream has been decoded:

1. When the next four bytes in the bitstream form the four-byte sequence 0x00000001, the next byte in the byte stream (which is a zero_byte syntax element) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this discarded byte.

2. The next three-byte sequence in the byte stream (which is a start_code_prefix_one_3bytes) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this three byte sequence.

3. NumBytesInNALunit is set equal to the number of bytes starting with the byte at the current position in the byte stream up to and including the last byte that precedes the location of any of the following conditions:

   – A subsequent byte-aligned three-byte sequence equal to 0x000000,

   – A subsequent byte-aligned three-byte sequence equal to 0x000001,

   – The end of the byte stream, as determined by unspecified means.

4. NumBytesInNALunit bytes are removed from the bitstream and the current position in the byte stream is advanced by NumBytesInNALunit bytes. This sequence of bytes is nal_unit(NumBytesInNALunit) and is decoded using the NAL unit decoding process.

5. When the current position in the byte stream is not at the end of the byte stream (as determined by unspecified means) and the next bytes in the byte stream do not start with a three-byte sequence equal to 0x000001 and the next bytes in the byte stream do not start with a four byte sequence equal to 0x00000001, the decoder extracts and discards each trailing_zero_8bits syntax element, moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next bytes in the byte stream form the four-byte sequence 0x00000001 or the end of the byte stream has been encountered (as determined by unspecified means).

A.3 Decoder byte-alignment recovery (informative)

Many applications provide data to a decoder in a manner that is inherently byte aligned, and thus have no need for the bit-oriented byte alignment detection procedure described in this subclause.

A decoder is said to have byte-alignment with a bitstream when the decoder is able to determine whether or not the positions of data in the bitstream are byte-aligned. When a decoder does not have byte alignment with the encoder's byte stream, the decoder may examine the incoming bitstream for the binary pattern ‘00000000 00000000 00000000 00000001’ (31 consecutive bits equal to 0 followed by a bit equal to 1). The bit immediately following this pattern is the first bit of an aligned byte following a start code prefix. Upon detecting this pattern, the decoder will be byte aligned with the encoder and positioned at the start of a NAL unit in the byte stream.

Once byte aligned with the encoder, the decoder can examine the incoming byte stream for subsequent three-byte sequences 0x000001 and 0x000003.

When the three-byte sequence 0x000001 is detected, this is a start code prefix.

When the three-byte sequence 0x000003 is detected, the third byte (0x03) is an emulation_prevention_three_byte to be discarded as specified in subclause XX.

When an error in the bitstream syntax is detected (e.g., a non-zero value of the forbidden_zero_bit or one of the three byte or four-byte sequences that are prohibited in subclause XX), the decoder may consider the detected condition as an indication that byte alignment may have been lost and may discard all bitstream data until the detection of byte alignment at a later position in the bitstream as described in this subclause.

BAnnex C
Hypothetical reference decoder

[Ed: Text describing the HRD and the bumping process to be taken from MPEG-4 AVC/H.264 as a starting point for this test model under consideration]

This annex specifies the hypothetical reference decoder (HRD) and its use to check bitstream and decoder conformance.

B.1 Operation of coded picture buffer (CPB)

B.2 Operation of the decoded picture buffer (DPB)

B.3 Bitstream conformance

B.4 Decoder conformance