INTERNATIONAL STANDARD

**ISO/IEC 14496-31**

First edition
2017-11

# Information technology — Coding of audio-visual objects —

## Part 31:
## Video coding for browsers

*Technologies de l'information — Codage des objets audiovisuels —*

*Partie 31: Codage vidéo pour navigateurs*

Reference number
ISO/IEC 14496-31:2017(E)

## Contents

# Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information.*

A list of all parts in the ISO/IEC 14496 series can be found on the ISO website.

# Information technology — Coding of audio-visual objects —

# Part 31: Video coding for browsers

## 1 Scope

This document specifies the video coding for browsers (VCB) syntax format and decoding process.

## 2 Normative references

There are no normative references in this document.

## 3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

**3.1**
**4:2:0 colour format**
colour format with two chroma arrays that each have half the height and half the width of the luma array

**3.2**
**AC transform coefficient**
*transform coefficient* (3.55) for which the *frequency index* (3.23) in at least one of the two dimensions is non-zero

**3.3**
**bitstream**
sequence of bits that forms the representation of *coded frames* (3.11)

**3.4**
**bit string**
ordered string with a limited number of bits, in which the left-most bit is the most significant bit (MSB) and the right-most bit is the least significant bit (LSB)

**3.5**
**block**
MxN (M-column by N-row) array of *sample values* (3.22) or MxN array of *transform coefficients* (3.55)

**3.6**
**Boolean decoder**
arithmetic decoder that processes one Boolean element at a time, wherein each Boolean element indicates how to refine the segmentation of a number line and hence specifies the decoded bits

**3.7**
**Boolean decoding**
process of decoding a *syntax element* (3.54) using a *Boolean decoder* (3.6)

**3.8**
**byte**
8-bit *bit string* (3.4)

**3.9**
**byte aligned**
position that is an integer multiple of eight, such that the first bit in the *bitstream* (3.3) has position 0 and the position of other bits is stated relative to this position

**3.10**
**chroma**
sample array or single *sample* (3.51), denoted by the symbols Cb and Cr, representing one of the two colour difference signals related to the primary colours

**3.11**
**coded frame**
set of *syntax elements* (3.54) that represents a *frame* (3.22) in the *bitstream* (3.3)

**3.12**
**component**
array or single *sample* (3.51) from one of the three arrays [*luma* (3.32) and two *chroma* (3.10)] that compose a picture in *4:2:0 colour format* (3.1)

**3.13**
**control partition**
partition containing the frame header and all of the macroblock headers for a *coded frame* (3.11)

**3.14**
**DC transform coefficient**
*transform coefficient* (3.55) for which the *frequency index* (3.23) in both of the two dimensions is zero

**3.15**
**decoded frame**
*frame* (3.22) reconstructed from a *coded frame* (3.11) by a *decoder* (3.16)

**3.16**
**decoder**
embodiment of the *decoding process* (3.17)

**3.17**
**decoding process**
process that derives *decoded frames* (3.15) from *syntax elements* (3.54)

**3.18**
**dequantization**
process in which *transform coefficients* (3.55) are obtained by scaling the *quantized transform coefficients* (3.45)

**3.19**
**encoder**
embodiment of an *encoding process* (3.20)

**3.20**
**encoding process**
process that generates a *bitstream* (3.3)

Note 1 to entry: Other than establishing criteria for conformance of bitstreams produced by an encoder, the encoding process is not specified in this document (although an example encoding process is informatively described in Annex B).

**3.21**
**flag**
binary variable

**3.22**
**frame**
array of luma samples (Y) and two corresponding arrays of chroma samples (Cb and Cr) in *4:2:0 colour format* (3.1)

**3.23**
**frequency index**
one-dimensional or two-dimensional index associated with a *transform coefficient* (3.55) prior to an *inverse transform* (3.29) part of the *decoding process* (3.17)

**3.24**
**inter coding**
*macroblock* (3.33) or *frame* (3.22) coded using *inter prediction* (3.25)

**3.25**
**inter prediction**
process of deriving the *prediction* (3.37) for the current *frame* (3.22) using previously *decoded frames* (3.15)

**3.26**
**intra coding**
*macroblock* (3.33) or *frame* (3.22) coded using *intra prediction* (3.28)

**3.27**
**I frame**
*frame* (3.22) that may be decoded using only *intra prediction* (3.28)

**3.28**
**intra prediction**
process of deriving the *prediction value* (3.42) for the current *sample* (3.51) using previously decoded *sample values* (3.52) in the same *decoded frame* (3.15)

**3.29**
**inverse transform**
part of the *decoding process* (3.17) by which a set of *transform coefficients* (3.55) are converted into spatial-domain values

**3.30**
**key frame**
*frame* (3.22) where the *decoding process* (3.17) is reset, such that a key frame and all following frames are always decodable without access to preceding frames and such that a frame is a key frame if and only if it is an *I frame* (3.27)

**3.31**
**level**
specified set of constraints on the permissible values for the *syntax elements* (3.54) and other characteristics of the *bitstreams* (3.3) produced by conforming *encoders* (3.19) established in combination with defined *profiles* (3.43) for ensuring interoperability of the bitsreams produced by conforming encoders with the *decoding process* (3.17) specified by conforming *decoders* (3.16)

**3.32**
**luma**
sample array or *single sample* (3.51), represented by the symbol Y, that ordinarily represents the brightness signal related to the primary colours

**3.33**
**macroblock**
16x16 luma sample value *block* (3.5) and its two corresponding 8x8 chroma sample value blocks

**3.34**
**motion vector**
two-dimensional vector used for *inter prediction* (3.25) that provides an offset from the coordinates in the *decoded frame* (3.15) to the coordinates in a *reference frame* (3.49)

**3.35**
**parse**
process of extracting a *syntax element* (3.54) from a *bitstream* (3.3)

**3.36**
**partitioning**
process of dividing a set into subsets such that each element in the set belongs to only one subset

**3.37**
**prediction**
prediction process consisting of either *inter* (3.25) or *intra prediction* (3.28)

**3.38**
**prediction frame**
**P frame**
*frame* (3.22) decoded by referencing another frame

**3.40**
**prediction process**
process of estimating the decoded *sample value* (3.52) or data element using a predictor

**3.41**
**prediction residual**
difference between the reconstructed *samples* (3.51) and the corresponding *prediction values* (3.42)

**3.42**
**prediction value**
combination of the previously decoded *sample values* (3.52) or data elements, used in the *prediction process* (3.40) of the next sample value or data element

**3.43**
**profile**
specified subset of syntax, semantics and processes established in combination with defined *levels* (3.31) for ensuring interoperability of the *bitstreams* (3.3) produced by conforming *encoders* (3.19) with the *decoding process* (3.17) specified for conforming *decoders* (3.16)

**3.44**
**quantization parameter**
variable used for scaling the *quantized coefficients* (3.45) in the *decoding process* (3.17)

**3.45**
**quantized coefficient**
*transform coefficient* (3.55) before *dequantization* (3.18)

**3.46**
**random access point**
point, other than the beginning of the *bitstream* (3.3), for starting the *decoding process* (3.17) for the remainder of a bitstream

**3.47**
**raster scan**
mapping of a two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the top row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc. rows of the pattern (going down) each scanned from left to right

**3.48**
**reconstruction**
addition of the decoded *prediction residual* (3.41) and the corresponding *prediction values* (3.42)

**3.49**
**reference frame**
previously *decoded frame* (3.15) used during *inter prediction* (3.25)

**3.50**
**reserved**
*syntax element* (3.54) value which may be used to extend this specification in the future

Note 1 to entry: Such values are not allowed to be present in bitstreams conforming to this document.

**3.51**
**sample**
basic element of the array of the *decoded frame* (3.15)

**3.52**
**sample value**
value of a *sample* (3.51), which is an integer in the range of 0 to 255, inclusive

**3.53**
**sub-block**
4x4 *block* (3.5)

**3.54**
**syntax element**
element of data represented in the *bitstream* (3.3)

**3.55**
**transform coefficient**
scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* (3.23) in an *inverse transform* (3.29) part of the *decoding process* (3.17)

## 4 Abbreviated terms

| | |
|---|---|
| DCT | Discrete cosine transform |
| LSB | Least significant bit |
| MB | Macroblock |
| MSB | Most significant bit |

| | |
|---|---|
| SB | Sub-block |
| VCB | Video coding for browsers |
| WHT | Walsh hadamard transform |

## 5 Conventions

### 5.1 General

The mathematical operators and their precedence rules used in this document are similar to those used in the C programming language. However, the operation of integer division with truncation is specifically defined.

In addition, an array with two elements used to hold a motion vector (such as Mv) can be accessed using either normal array notation (e.g. Mv[ 0 ] and Mv[ 1 ]), or by name (i.e. Mv). The only operations defined when using the name are assignment and equality testing. The assignment of an array is represented using the normal notation A = B and is specified to mean the same as doing both the individual assignments A[ 0 ] = B[ 0 ] and A[ 1 ] = B[ 1 ]. Equality testing of two motion vectors is represented using the notation A == B and is specified to mean the same as (A[ 0 ] == B[ 0 ] && A[ 1 ] == B[ 1 ]).

### 5.2 Arithmetic operators

| | |
|---|---|
| + | Addition |
| − | Subtraction (as a binary operator) or negation (as a unary prefix operator) |
| * | Multiplication |
| / | 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. |
| a % b | Remainder of a divided by b, defined only for a >= 0 and b > 0. |

### 5.3 Logical operators

| | |
|---|---|
| a && b | Logical AND operation between a and b |
| a \|\| b | Logical OR operation between a and b |
| ! | Logical NOT operation |

### 5.4 Relational operators

| | |
|---|---|
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| == | Equal to |
| != | Not equal to |

## 5.5 Bitwise operators

| | |
|---|---|
| & | AND operation |
| \| | OR operation |
| ~ | Negation operation |
| a >> b | Shift a in 2's complement binary integer representation format to the right by b bit positions. This operator is only used with b being a non-negative integer. Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of a prior to the shift operation. |
| a << b | Shift a in 2's complement binary integer representation format to the left by b bit positions. This operator is only used with b being a non-negative integer. Bits shifted into the LSBs as a result of the left shift have a value equal to 0. |

## 5.6 Assignment operators

| | |
|---|---|
| = | Assignment operator |
| ++ | Increment, x++ is equivalent to x = x + 1. When this operator is used for an array index, the variable value is obtained before the increment operation. |
| −− | Decrement, i.e. x−− is equivalent to x = x − 1. When this operator is used for an array index, the variable value is obtained before the decrement operation. |
| += | Addition assignment operator, for example: x += 3 corresponds to x = x + 3, and, x += (−3) is equivalent to x = x + (−3). |
| −= | Subtraction assignment operator, for example: x −= 3 corresponds to x = x − 3, and, x −= (−3) is equivalent to x = x − (−3). |

## 5.7 Range notation

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

## 5.8 Mathematical functions

The following mathematical functions are defined as follows:

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

$$\mathrm{Clip1}(x) = \mathrm{Clip3}(0, 255, x)$$

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

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

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

## 5.9 Order of operation precedence

When order of precedence in an expression is not indicated explicitly by the 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 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 document is the same as used in the C programming language.

**Table 1 — Operation precedence from highest (at top of the table) to lowest (at the bottom of the 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 : z" |
| "x..y" |
| "x = y", "x += y", "x −= y" |

## 5.10 Method of describing bitstream syntax

The bitstream syntax is specified in Clause 6 in a style similar to the C programming language. A syntax element is completely described by its name together with either one or two descriptors for its method of coded representation. Syntax element names consist of descriptive groups of lowercase letters separated by an underscore character. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements.

A syntax element is represented in **bold** type in two places; where it is read from the bitstream (in Clause 6) and where it is defined (in Clause 7). Otherwise, syntax elements are represented in non-bold type throughout.

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 lowercase and uppercase letters without any underscore characters. Variables starting with an uppercase letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an uppercase 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 lowercase letter are only used within the subclause from which they are derived.

The association of values and names is specified in the text. In some cases, “mnemonic” names for syntax elements or variables are used interchangeably with their numerical values. The names are constructed from one or more groups of letters separated by an underscore character. Each group starts with an uppercase letter and may contain more uppercase letters.

Hexadecimal notation, indicated by prefixing the hexadecimal number by “0x”, may be used when the number of bits is an integer multiple of 4. For example, “0x1a” represents a bit-string “0001 1010”.

A value equal to 0 represents a FALSE condition in a test statement. The value TRUE is represented by any value other than zero.

Table 2 lists examples of the syntax specification format. When **syntax_element** appears (with bold face font), it specifies that a syntax element is parsed from a bitstream.

**Table 2 — Examples of syntax specification formats**

| **syntax** | **type** |
|---|---|
| /* A statement can be a syntax element with an associated descriptor or can be an expression used to specify its existence, type and value, as in the following examples */ | |
| **syntax_element** | ue(v) |
| conditioning statement | |
| | |
| /* A group of statements enclosed in brackets is a compound statement and is treated functionally as a single statement. */ | |
| { | |
|     statement | |
|     statement | |
|     … | |
| } | |
| | |
| /* A “while” structure specifies that the statement is to be evaluated repeatedly while the condition remains true. */ | |
| while (condition) | |
|     statement | |
| | |
| /* A “do … while” structure executes the statement once and then tests the condition. It repeatedly evaluates the statement while the condition remains true. */ | |
| do | |
|     statement | |
| while (condition) | |
| | |

**Table 2** *(continued)*

| syntax | type |
|---|---|
| /* An "if ... else" structure tests the condition first. If it is true, the primary statement is evaluated. Otherwise, the alternative statement is evaluated. If the alternative statement is unnecessary to be evaluated, the "else" and corresponding alternative statement can be omitted. */ | |
| if (condition) | |
| primary statement | |
| else | |
| alternative statement | |
| | |
| /* A "for" structure evaluates the initial statement at the beginning, then tests the condition. If it is true, the primary and subsequent statements are evaluated until the condition becomes false. */ | |
| for ( initial statement; condition; subsequent statement ) | |
| primary statement | |

## 5.11 Functions

Functions used for syntax specification are defined in this subclause.

The specification of these functions makes use of a bitstream position indicator. This bitstream position indicator locates the position of the bit that is going to be read next.

**get_position( )**: Return the value of a bitstream position indicator.

**init_bool( b, sz )**: Initialize the decoding process for Boolean decoder b with a size of sz as specified in 9.2.2.

**set_bool( b )**: Change to using Boolean decoder b as specified in 9.2.3.

## 5.12 Descriptors

### 5.12.1 General

The following descriptors specify the parsing process of each syntax element.

### 5.12.2 f(n)

An unsigned n-bit number appearing directly in a bitstream. The bits are read from high to low order. The parsing process specified in 9.1 is invoked and the syntax element is set equal to the return value.

### 5.12.3 B(p)

A single Boolean encoded bit with estimated probability p÷256 of being 0. The syntax element is set equal to the return value of read_bool( p ) (see 9.2.4 for a specification of this process).

NOTE p might be equal to 0, but there is no special treatment of this case.

### 5.12.4 u(n)

An unsigned Boolean encoded n-bit number encoded as n flags (a "literal"). The bits are read from high to low order. The syntax element is set equal to the return value of read_literal( n ) (see 9.2.5 for a specification of this process).

### 5.12.5 F

A single Boolean encoded bit with estimated probability 50% of being 0. The syntax element is set equal to the return value of read_bool( 128 ).

NOTE A flag is the same as B(128) or u(1).

### 5.12.6 P(7)

A Boolean encoded 7-bit specification of an 8-bit estimated probability. Coded as an u(7) number x; the resulting 8-bit estimated probability is x ? x << 1 : 1.

### 5.12.7 P(8)

A Boolean encoded 8-bit estimated probability. This is the same as u(8), but this notation is used to emphasize that an estimated probability is being coded.

### 5.12.8 T

A Boolean tree-encoded value from an alphabet. Such values represent the leaves of a binary tree. The non-leaf nodes of the tree have associated probabilities p and are represented by B(p). A zero represents choosing the left branch below the current node and a one represents choosing the right branch. Each element of this type has an associated table of estimated node probability values defined in this document. Reference is made to those tables when required.

Every leaf value whose tree depth is x is decoded using x B(p) values.

There are many ways that a given alphabet can be so represented. The choice of tree has little impact on data rate but does affect decoder performance.

## 6 Bitstream syntax

### 6.1 General

This clause specifies the bitstream syntax in a tabular form. The meaning of each of the syntax elements is presented in Clause 7.

### 6.2 Frame syntax

A bitstream consists of a sequence of one or more coded frames ordered in decoding order. Methods of framing the coded frames in a container format are outside the scope of this document. Each coded frame in turn is given to the decoding process as a bit string together with a variable, named "sz" in Table 3, that specifies the total number of bytes in the coded frame.

**Table 3 — Frame syntax**

| frame( sz ) { | **Type** |
|---|---|
| uncompressed_header( ) | |
| sz −= ControlPartitionSize + HeaderSize | |
| init_bool( 0, ControlPartitionSize ) | |
| set_bool( 0 ) | |
| compressed_header( ) | |
| for ( i = 0; i < NbrPartitions − 1; i++ ) { | |
| **partition_size_low** | f(8) |
| **partition_size_mid** | f(8) |

**Table 3** *(continued)*

| | |
|---|---|
|     **partition_size_high** | f(8) |
|     sz −= partitionSize[ i ] + 3 | |
|   } | |
|   partitionSize[ NbrPartitions − 1 ] = sz | |
|   for ( b = 0; b < NbrPartitions; b++ ) | |
|     init_bool( b + 1, partitionSize[ b ] ) | |
|   for ( col = 0; col < MbCols; col++ ) { | |
|     for ( i = 0; i < 9; i++ ) | |
|       AboveToken[ col ][ i ] = 0 | |
|     MbRefFrame[ −1 ][ col ] = 0 | |
|     MbSplit[ −1 ][ col ] = 0 | |
|   } | |
|   MbRefFrame[ −1 ][ −1 ] = 0 | |
|   MbSplit[ −1 ][ −1 ] = 0 | |
|   for ( row = 0; row < MbRows; row++ ) { | |
|     for ( i = 0; i < 9; i++ ) | |
|       LeftToken[ i ] = 0 | |
|   MbRefFrame[ row ][ −1 ] = 0 | |
|   MbSplit[ row ][ −1 ] = 0 | |
|   for ( col = 0; col < MbCols; col++ ) { | |
|     set_bool( 0 ) | |
|     mb_header( row, col ) | |
|     set_bool( 1 + (row % NbrPartitions) ) | |
|     prediction_residual_data( row, col ) | |
|     } | |
|   } | |
| } | |

## 6.3 Uncompressed header syntax

| uncompressed_header( ) { | **Type** |
|---|---|
|   **control_partition_size_low** | f(3) |
|   **show_frame** | f(1) |
|   **version_number** | f(3) |
|   **frame_type** | f(1) |
|   **control_partition_size_mid** | f(8) |
|   **control_partition_size_high** | f(8) |
|   if ( KeyFrame ) { | |
|     **sync_byte_0** | f(8) |
|     **sync_byte_1** | f(8) |
|     **sync_byte_2** | f(8) |
|     **frame_width_low** | f(8) |
|     **horizontal_scale** | f(2) |
|     **frame_width_high** | f(6) |
|     **frame_height_low** | f(8) |

**Table 3** *(continued)*

| | |
|---|---|
|   **vertical_scale** | f(2) |
|   **frame_height_high** | f(6) |
|   HeaderSize = 10 | |
|  } else | |
|   HeaderSize = 3 | |
| } | |

## 6.4 Compressed header syntax

### 6.4.1 General

| compressed_header( ) { | Type |
|---|---|
|  if ( KeyFrame ) { | |
|   **colour_space** | u(1) |
|   **clamping_type** | u(1) |
|  } | |
|  **segmentation_enabled** | u(1) |
|  if ( segmentation_enabled ) | |
|   update_segmentation( ) | |
|  **filter_type** | u(1) |
|  **loop_filter_level** | u(6) |
|  **sharpness_level** | u(3) |
|  mb_lf_adjustments( ) | |
|  **log2_nbr_of_dct_partitions** | u(2) |
|  quant_indices( ) | |
|  if ( KeyFrame ) | |
|   **refresh_entropy_probs** | u(1) |
|  else { | |
|   **refresh_ref1_frame** | u(1) |
|   **refresh_ref2_frame** | u(1) |
|   if ( !refresh_ref1_frame ) | |
|    **copy_buffer_to_ref1** | u(2) |
|   if ( !refresh_ref2_frame ) | |
|    **copy_buffer_to_ref2** | u(2) |
|   **sign_bias_ref1** | u(1) |
|   **sign_bias_ref2** | u(1) |
|   **refresh_entropy_probs** | u(1) |
|   **refresh_ref0** | u(1) |
|  } | |
|  token_prob_update( ) | |
|  **mb_has_skip_coeff** | u(1) |
|  if ( mb_has_skip_coeff ) | |
|   **prob_skip_false** | P(8) |
|  if ( !KeyFrame ) { | |

| | |
|---|---|
| **prob_intra** | P(8) |
| **prob_ref0** | P(8) |
| **prob_ref1** | P(8) |
| **intra_16x16_prob_update_flag** | u(1) |
| if ( intra_16x16_prob_update_flag ) | |
| for ( i = 0; i < 4; i++ ) | |
| **intra_16x16_prob**[ i ] | P(8) |
| **intra_chroma_prob_update_flag** | u(1) |
| if ( intra_chroma_prob_update_flag ) | |
| for ( i = 0; i < 3; i++ ) | |
| **intra_chroma_prob**[ i ] | P(8) |
| mv_prob_update( ) | |
| } | |
| } | |

### 6.4.2 Segment based adjustments syntax

| update_segmentation( ) { | **Type** |
|---|---|
| **update_mb_segmentation_map** | u(1) |
| **update_segment_feature_data** | u(1) |
| if ( update_segment_feature_data ) { | |
| **segment_feature_mode** | u(1) |
| for ( i = 0; i < 4; i++ ) { | |
| **quantizer_update** | u(1) |
| if ( quantizer_update ) { | |
| **quantizer_update_value** | u(7) |
| **quantizer_update_sign** | u(1) |
| } | |
| } | |
| for ( i = 0; i < 4; i++ ) { | |
| **loop_filter_update** | u(1) |
| if ( loop_filter_update ) { | |
| **lf_update_value** | u(6) |
| **lf_update_sign** | u(1) |
| } | |
| } | |
| } | |
| if ( update_mb_segmentation_map ) | |
| for ( i = 0; i < 3; i++ ) { | |
| **segment_prob_update** | u(1) |
| if ( segment_prob_update ) | |
| **segment_prob**[ i ] | P(8) |
| else | |

| | |
|---|---|
|         **segment_prob**[ i ] = 255 | |
|     } | |
| } | |

### 6.4.3 Loop filter adjustments syntax

| mb_lf_adjustments( ) { | **Type** |
|---|---|
|   **loop_filter_adj_enable** | u(1) |
|   if ( loop_filter_adj_enable ) { | |
|     **mode_ref_lf_delta_update** | u(1) |
|     if ( mode_ref_lf_delta_update ) { | |
|       for ( i = 0; i < 4; i++ ) { | |
|         **ref_frame_delta_update_flag** | u(1) |
|         if ( ref_frame_delta_update_flag ) { | |
|           **delta_magnitude** | u(6) |
|           **delta_sign** | u(1) |
|         } | |
|       } | |
|       for ( i = 0; i < 4; i++ ) { | |
|         **mb_mode_delta_update_flag** | u(1) |
|         if ( mb_mode_delta_update_flag ) { | |
|           **mode_delta_magnitude** | u(6) |
|           **mode_delta_sign** | u(1) |
|         } | |
|       } | |
|     } | |
|   } | |
| } | |

### 6.4.4 Quantizer indices syntax

| quant_indices( ) { | **Type** |
|---|---|
|   **y_ac_qi** | u(7) |
|   **y_dc_delta_present** | u(1) |
|   if ( y_dc_delta_present ) { | |
|     **y_dc_delta_magnitude** | u(4) |
|     **y_dc_delta_sign** | u(1) |
|   } | |
|   **y2_dc_delta_present** | u(1) |
|   if ( y2_dc_delta_present ) { | |
|     **y2_dc_delta_magnitude** | u(4) |
|     **y2_dc_delta_sign** | u(1) |
|   } | |
|   **y2_ac_delta_present** | u(1) |
|   if ( y2_ac_delta_present ) { | |

| | |
|---|---|
|     **y2_ac_delta_magnitude** | u(4) |
|     **y2_ac_delta_sign** | u(1) |
|   } | |
|   **uv_dc_delta_present** | u(1) |
|   if ( uv_dc_delta_present ) { | |
|     **uv_dc_delta_magnitude** | u(4) |
|     **uv_dc_delta_sign** | u(1) |
|   } | |
|   **uv_ac_delta_present** | u(1) |
|   if ( uv_ac_delta_present ) { | |
|     **uv_ac_delta_magnitude** | u(4) |
|     **uv_ac_delta_sign** | u(1) |
|   } | |
| } | |

### 6.4.5 Token probability update syntax

| token_prob_update( ) { | **Type** |
|---|---|
|   for ( i = 0; i < 4; i++ ) | |
|     for ( j = 0; j < 8; j++ ) | |
|       for ( k = 0; k < 3; k++ ) | |
|         for ( t = 0; t < 11; t++ ) { | |
|           **coeff_prob_update_flag** | B(CoeffUpdateProbs[ i ][ j ][ k ][ t ]) |
|           if ( coeff_prob_update_flag ) | |
|             **coeff_prob**[ i ][ j ][ k ][ t ] | P(8) |
|         } | |
| } | |

The contents of CoeffUpdateProbs are specified in Table 35.

### 6.4.6 MV probability update syntax

| mv_prob_update( ) { | **Type** |
|---|---|
|   for ( i = 0; i < 2; i++ ) | |
|     for ( j = 0; j < 19; j++ ) { | |
|       **mv_prob_update_flag** | B(MvUpdateProbs[ i ][ j ]) |
|       if ( mv_prob_update_flag ) | |
|         **mv_prob**[ i ][ j ] | P(7) |
|     } | |
| } | |

The contents of MvUpdateProbs are specified in Table 36.

## 6.5 Macroblock data syntax

### 6.5.1 Macroblock header syntax

| mb_header( row, col ) { | **Type** |
|---|---|
|   if ( update_mb_segmentation_map ) | |
|     **segment_id** | T |
|   if ( mb_has_skip_coeff ) | |
|     **mb_skip_coeff** | B(prob_skip_false) |
|   if ( !KeyFrame ) | |
|     **is_inter_mb** | B(prob_intra) |
|   if ( is_inter_mb ) { | |
|     **mb_ref_frame_sel1** | B(prob_ref0) |
|     if ( mb_ref_frame_sel1 ) | |
|       **mb_ref_frame_sel2** | B(prob_ref1) |
|     **mv_mode** | T |
|     if ( mv_mode == MV_SPLIT ) { | |
|       **mv_split_mode** | T |
|       for ( i = 0; i < numMvs; i++ ) { | |
|         **sub_mv_mode** | T |
|         if ( sub_mv_mode == NEW4X4 ) { | |
|           SubMvdy = mv_component( 0 ) | |
|           SubMvdx = mv_component( 1 ) | |
|         } | |
|       } | |
|     } else if ( mv_mode == MV_NEW ) { | |
|       Mvdy = mv_component( 0 ) | |
|       Mvdx = mv_component( 1 ) | |
|     } | |
|   } else { /* intra mb */ | |
|     **intra_y_mode** | T |
|     if ( intra_y_mode == B_PRED ) | |
|     for ( block = 0; block < 16; block++ ) | |
|       **intra_b_mode**[ block ] | T |
|     **intra_uv_mode** | T |
|   } | |
| } | |

#### 6.5.1.1 MV component syntax

| mv_component( cp ) { | **Type** |
|---|---|
|   **mv_long** | B(mv_prob[ cp ][ 0 ]) |
|   if ( mv_long ) { | |
|     c = 0 | |
|     for ( i = 0; i < 3; i++ ) { | |

| | |
|---|---|
| **mv_bit** | B(mv_prob[ cp ][ 9 + i ] ) |
| c += mv_bit << i | |
| } | |
| for ( i = 9; i > 3; i−− ) { | |
| **mv_bit** | B(mv_prob[ cp ][ 9 + i ]) |
| c += mv_bit << i | |
| } | |
| if ( c & 0xfff0 ) { | |
| **mv_bit** | B(mv_prob[ cp ][ 9 + 3 ]) |
| c += mv_bit << 3 | |
| } else | |
| c += 8 | |
| } else { | |
| **mv_short** | T |
| c = mv_short | |
| } | |
| if ( c ) { | |
| **mv_sign** | B(mv_prob[ cp ][ 1 ]) |
| if ( mv_sign ) | |
| c = −c | |
| } | |
| return c | |
| } | |

### 6.5.2 Prediction residual data syntax

| prediction_residual_data( row, col ) { | **Type** |
|---|---|
| HasNonZero = 0 | |
| if ( ( is_inter_mb && mv_mode != MV_SPLIT ) | |
| \|\| ( !is_inter_mb && intra_y_mode != B_PRED ) ) | |
| HasY2 = 1 | |
| else | |
| HasY2 = 0 | |
| if ( mb_skip_coeff == 0 ) { | |
| if ( HasY2 ) { | |
| prediction_residual_block( 24, 0 ) /* Y2 */ | |
| LeftToken[ 8 ] = AboveToken[ col ][ 8 ] = NonZero[ 24 ] | |
| for ( i = 0; i < 16; i++ ) | |
| prediction_residual_block( i, 1 ) /* Y */ | |
| } else | |
| for ( i = 0; i < 16; i++ ) | |
| prediction_residual_block( i, 0 ) /* Y */ | |
| for ( i = 0; i < 8; i++ ) | |
| prediction_residual_block( 16 + i, 0 ) /* 4 Cb, 4 Cr */ | |
| for ( i = 0; i < 4; i++ ) { | |

| | |
|---|---|
|             LeftToken[ i ] = NonZero[ i * 4 + 3 ] | |
|             AboveToken[ col ][ i ] = NonZero[ 12 + i ] | |
|         } | |
|         for ( i = 0; i < 2; i++ ) { | |
|             LeftToken[ 4 + i ] = NonZero[ 16 + i * 2 + 1 ] | |
|             AboveToken[ col ][ 4 + i ] = NonZero[ 16 + 2 + i ] | |
|             LeftToken[ 6 + i ] = NonZero[ 16 + 4 + i * 2 + 1 ] | |
|             AboveToken[ col ][ 6 + i ] = NonZero[ 16 + 4 + 2 + i ] | |
|         } | |
|     } else | |
|         for ( i = 0; i < ( HasY2 ? 9 : 8 ); i++ ) { | |
|             LeftToken[ i ] = 0 | |
|             AboveToken[ col ][ i ] = 0 | |
|         } | |
| } | |

## 6.5.3 Prediction residual block syntax

| prediction_residual_block( block, firstCoeff ) { | **Type** |
|---|---|
|     lastCoeff = 1 | |
|     breakLoop = 0 | |
|     for ( i = firstCoeff; i < 16 && !breakLoop; i++ ) { | |
|         **coeff_token** | T |
|         if ( coeff_token == EOB ) | |
|             breakLoop = 1 | |
|         else { | |
|             if ( coeff_token >= DCT_CAT1 ) { | |
|                 v = 0 | |
|                 for ( j = 0; j < catNum; j++ ) { | |
|                     **extra_bit** | B(catProbs[ coeff_token ][ j ]) |
|                     v += extra_bit | |
|                 } | |
|                 coeffVal = catBase + v | |
|             } else | |
|                 coeffVal = coeff_token | |
|             if ( coeffVal != 0 ) { | |
|                 **coef_sign** | u(1) |
|                 if ( coef_sign ) | |
|                     coeffVal = −coeffVal | |
|             } | |
|             lastCoeff = coeffVal | |
|             y = zigzag[ i ] >> 2 | |
|             x = zigzag[ i ] − y * 4 | |
|             TransCoeff[ block ][ x ][ y ] = coeffVal | |
|         } | |

| | |
|---|---|
|   } | |
|   if ( i == firstCoeff ) | |
|     NonZero[ block ] = 0 | |
|   else { | |
|     NonZero[ block ] = 1 | |
|     HasNonZero = 1 | |
|   } | |
|   for ( ; i < 16; i++ ) { | |
|     y = zigzag[ i ] >> 2 | |
|     x = zigzag[ i ] − y * 4 | |
|     TransCoeff[ block ][ x ][ y ] = 0 | |
|   } | |
| } | |

The zigzag scan is specified as follows:

| |
|---|
| zigzag[ 16 ] = { |
|   0, 1, 4, 8, 5, 2, 3, 6, 9, 12, 13, 10, 7, 11, 14, 15 |
| } |

## 7 Bitstream semantics

### 7.1 General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this clause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document. This clause specifies the meaning of the syntax elements.

### 7.2 Frame semantics

A bitstream consists of a sequence of one or more coded frames. The first frame in a bitstream shall be a key frame.

**partition_size_low**, **partition_size_mid**, and **partition_size_high** combine to give the number of bytes in the i$^{th}$ partition of the prediction residual data. The variable partitionSize[ i ] is specified as:

partitionSize[ i ] = partition_size_low + (partition_size_mid << 8) + (partition_size_high << 16)

### 7.3 Uncompressed header semantics

**frame_type** specifies whether the current frame is a P frame. The variable KeyFrame is specified as follows:

— If frame_type is equal to 0, KeyFrame is set equal to 1.

— Otherwise (frame_type is equal to 1), KeyFrame is set equal to 0.

A frame is a key frame if and only if KeyFrame is equal to 1.

When KeyFrame is equal to 1, the saved probability tables are reset to their default values as follows:

```
for ( i = 0; i < 4; i++ )
    SavedIntra16x16Prob[ i ] = DefaultIntra16x16Prob[ i ]
for ( i = 0; i < 3; i++ )
    SavedIntraChromaProb[ i ] = DefaultIntraChromaProb[ i ]
for ( i = 0; i < 4; i++ )
    for ( j = 0; j < 8; j++ )
        for ( k = 0; k < 3; k++ )
            for ( t = 0; t < 11; t++ )
                SavedCoeffProb[ i ][ j ][ k ][ t ] = DefaultCoeffProb[ i ][ j ][ k ][ t ]
for ( i = 0; i < 2; i++ )
    for ( j = 0; j < 19; j++ )
        SavedMvProb[ i ][ j ] = DefaultMvProb[ i ][ j ]
```

The contents of the arrays DefaultIntra16x16Prob and DefaultIntraChromaProb are specified in Table 37.

The contents of DefaultCoeffProb are specified in Table 38.

The contents of DefaultMvProb are specified in Table 39.

The saved probability tables persist across frames.

The first coded frame shall have KeyFrame equal to 1.

**version_number** specifies the filters used for reconstruction and loop filtering as shown in Table 4.

**Table 4 — Convolution filter and loop filter types according to version_number**

| version_number | Convolution Filter | Loop Filter |
|---|---|---|
| 0 | Bicubic | Normal |
| 1 | Bilinear | Simple |
| 2 | Bilinear | None |
| 3 | None | None |
| 4, 5, 6, 7 | Reserved for future use | |

The version_number specifies the worst case loop filter that may be present. A bitstream may have version_number equal to 0 and filter_type equal to 1 indicating the simple loop filter is to be used.

**show_frame** specifies whether the current frame is meant to be displayed.

**control_partition_size_low, control_partition_size_mid, control_partition_size_high** specify the size of the first partition, called the control partition, in bytes, excluding the uncompressed data chunk. The variable ControlPartitionSize is set equal to control_partition_size_low + (control_partition_size_mid << 3) + (control_partition_size_high << 11).

**sync_byte_0** shall be 0x9d.

**sync_byte_1** shall be 0x01.

**sync_byte_2** shall be 0x2a.

**horizontal_scale** and **vertical_scale** specify the frame scaling ratio for each dimension as shown in Table 5.

**Table 5 — Frame width and frame height scaling ratio specifications**

| Value | Up-scaling ratio |
|---|---|
| 0 | 1 : 1 |
| 1 | 5 : 4 |
| 2 | 5 : 3 |
| 3 | 2 : 1 |

NOTE 1 Up-scaling does not affect the reconstruction buffer, which is maintained at the encoded resolution.

**frame_width_low, frame_width_high** and **frame_height_low, frame_height_high** specify the dimensions of the key frame and the following P frames. The variables FrameWidth and FrameHeight are specified as follows.

— FrameWidth = (frame_width_high << 8) + frame_width_low

— FrameHeight = (frame_height_high << 8) + frame_height_low

Both FrameWidth and FrameHeight shall be greater than 0. The variables MbRows and MbCols are defined as follows.

— MbCols = ( FrameWidth + 15 ) >> 4

— MbRows = ( FrameHeight + 15 ) >> 4

The decoding process operates on complete macroblocks. In a decoder implementation, the current frame and reference frame buffers shall have enough storage to hold all the sample values for a frame that is MbCols macroblocks wide and MbRows macroblocks high.

NOTE 2 The frame dimensions are not necessarily evenly divisible by 16. The right-most MbCols * 16 − FrameWidth columns and the upper-most MbRows * 16 − FrameHeight rows are not actually part of the decoded frame and are discarded from the final output. However, these "excess samples" are maintained in the internal reconstruction buffer that is used to predict subsequent frames.

## 7.4 Compressed header semantics

### 7.4.1 General

**colour_space** shall be 0 for bitstreams conforming to this document. All other values are reserved for future use.

**clamping_type** specifies whether clamping can be ignored in the reconstruction process for this frame and the following P frames.

The sample value clamping type bit is interpreted as follows.

— 0: Reconstructed sample values need to be clamped between 0 and 255 (inclusive).

— 1: Reconstructed sample values are guaranteed to be between 0 and 255; no clamping is necessary.

When clamping_type is not present, it is inferred to be equal to the value in the previous frame.

**segmentation_enabled** specifies that segmentation is used for the current frame.

**filter_type** specifies the loop filter type. Two types are defined in this document: the “normal” filter type (corresponding to a value of 0) and the “simple” loop filter type (corresponding to a value of 1).

If version_number is equal to 1, then filter_type shall be equal to 1.

**loop_filter_level** specifies one of the control parameters of the deblocking filter.

**sharpness_level** specifies one of the control parameters of the deblocking filter.

**log2_nbr_of_dct_partitions** determines the number of separate partitions containing the DCT coefficients of the macroblocks. The variable NbrPartitions is set equal to 1 << log2_nbr_of_dct_partitions.

**refresh_entropy_probs** specifies whether the updated token probabilities are also used for subsequent frames.

**refresh_ref1_frame** specifies whether the current decoded frame refreshes the Ref1 reference frame. When not present, it is inferred equal to 1.

**refresh_ref2_frame** specifies whether the current decoded frame refreshes the Ref2 reference frame. When not present, it is inferred equal to 1.

**copy_buffer_to_ref1** specifies how to update the Ref1 reference frame. When not present, it is inferred equal to 0. The value is interpreted as follows.

— 0: no change is made to the Ref1 frame.

— 1: Ref0 is to be copied to the Ref1 frame.

— 2: Ref2 is to be copied to the Ref1 frame.

copy_buffer_to_ref1 shall not be equal to 3.

**copy_buffer_to_ref2** specifies how to update the Ref2 frame. When not present, it is inferred equal to 0. The value is interpreted as follows.

— 0: no change is made to the Ref2 frame.

— 1: Ref0 is to be copied to the Ref2 frame.

— 2: Ref1 frame is to be copied to the Ref2 frame.

copy_buffer_to_ref2 shall not be equal to 3.

**sign_bias_ref1** controls the sign of motion vectors when the Ref1 frame is referenced.

**sign_bias_ref2** controls the sign of motion vectors when the Ref2 frame is referenced.

The array entries of RefFrameSignBias are set as follows.

— RefFrameSignBias[ 1 ] = 0

— RefFrameSignBias[ 2 ] = sign_bias_ref1

— RefFrameSignBias[ 3 ] = sign_bias_ref2

**refresh_ref0** specifies whether the current decoded frame refreshes the Ref0 reference buffer. When refresh_ref0 is not present, it is inferred equal to 1.

**mb_has_skip_coeff** specifies whether mb_skip_coeff is present in the macroblock headers.

**prob_skip_false** specifies the probability that the macroblock is not skipped.

**prob_intra** specifies the probability that the macroblock uses intra coding.

**prob_ref0** specifies the probability that the Ref0 reference frame is used for inter prediction.

**prob_ref1** specifies the probability that the Ref1 reference frame is used for inter prediction.

**intra_16x16_prob_update_flag** specifies whether the branch probabilities used in the decoding of the luma intra prediction mode are updated.

**intra_16x16_prob[ i ]** specifies the branch probabilities of the luma intra prediction mode decoding tree. When not present in a bitstream, it is inferred to be equal to SavedIntra16x16Prob[ i ].

**intra_chroma_prob_update_flag** specifies whether the branch probabilities used in the decoding of the chroma intra-prediction mode are updated.

**intra_chroma_prob[ i ]** specifies the branch probabilities of the chroma intra prediction mode decoding tree. When not present in a bitstream, it is inferred to be equal to SavedIntraChromaProb[ i ].

### 7.4.2 Segment-based adjustments semantics

When segment adaptive adjustments are enabled, each macroblock is assigned a segment ID. Macroblocks with the same segment ID belong to the same segment and have the same adaptive adjustments over default baseline values for the frame. The adjustments can be quantizer level or loop filter strength.

**update_mb_segmentation_map** specifies whether the MB segmentation map is updated in the current frame. When not present in a bitstream, it is inferred to be 0.

**update_segment_feature_data** specifies whether the segment feature data is updated in the current frame.

**segment_feature_mode** specifies the feature data update mode, 0 for delta and 1 for the absolute value. When not present in a bitstream, it is inferred as follows.

— If KeyFrame is equal to 1, segment_feature_mode is set equal to 0.

— Otherwise, segment_feature_mode is left at the value it took in the previous frame.

**quantizer_update** specifies if the quantizer value is set for the $i^{th}$ segment.

**quantizer_update_value** specifies the update value for the segment quantizer.

**quantizer_update_sign** specifies the update sign for the segment quantizer.

The variable Quantizer[ i ] is set as follows:

— If update_segment_feature_data is equal to 0 and KeyFrame is equal to 1, Quantizer[ i ] is set equal to 0.

— Otherwise, if update_segment_feature_data is equal to 0, Quantizer[ i ] is left at the value it took in the previous frame.

— Otherwise, if quantizer_update is equal to 0, Quantizer[ i ] is set equal to 0.

— Otherwise, if quantizer_update_sign is equal to 0, Quantizer[ i ] is set equal to quantizer_update_value.

— Otherwise, Quantizer[ i ] is set equal to −quantizer_update_value (the negative of quantizer_update_value).

— loop_filter_update specifies if the loop filter level is updated for the $i^{th}$ segment.

— lf_update_value specifies the update value for the loop filter level.

— lf_update_sign specifies the update sign for the loop filter level.

The variable LfLevel[ i ] is set as follows:

— If update_segment_feature_data is equal to 0 and KeyFrame is equal to 1, LfLevel[ i ] is set equal to 0.

— Otherwise, if update_segment_feature_data is equal to 0, LfLevel[ i ] is left at the value it took in the previous frame.

— Otherwise, if lf_update_value is equal to 0, LfLevel[ i ] is set equal to 0.

— Otherwise, if lf_update_sign is equal to 0, LfLevel[ i ] is set equal to lf_update_value.

— Otherwise, LfLevel [ i ] is set equal to −lf_update_value (the negative of lf_update_value).

**segment_prob_update** specifies whether the branch probabilities used to decode the segment_id in the macroblock header are decoded from the stream or are set to the default value of 255.

**segment_prob[ i ]** specifies the branch probabilities of the segment_id decoding tree.

### 7.4.3 Loop filter adjustments semantics

The frame filter level may be adjusted per macroblock based on a macroblock's prediction mode and reference frame. The per-macroblock adjustment is done through delta values against the default loop filter level for the current frame.

**loop_filter_adj_enable** specifies whether the MB-level loop filter adjustment (based on the used reference frame and coding mode) is enabled for the current frame.

**mode_ref_lf_delta_update** specifies whether the delta values used in an adjustment are updated in the current frame.

**ref_frame_delta_update_flag** specifies whether the adjustment delta value corresponding to a certain used reference frame is updated.

**delta_magnitude** is the absolute value of the delta value.

**delta_sign** is the sign of the delta value.

The variable RefDelta[ i ] is set as follows:

— If loop_filter_adj_enable is equal to 0 and KeyFrame is equal to 1, RefDelta[ i ] is set equal to 0.

— Otherwise, if loop_filter_adj_enable is equal to 0, RefDelta[ i ] is left at the value it took in the previous frame.

— Otherwise, if ref_frame_delta_update_flag is equal to 0, RefDelta[ i ] is set equal to 0.

— Otherwise, if delta_sign is equal to 0, RefDelta[ i ] is set equal to delta_magnitude.

— Otherwise, RefDelta[ i ] is set equal to −delta_magnitude (the negative of delta_magnitude).

**mb_mode_delta_update_flag** specifies if the adjustment delta value corresponding to a certain MB prediction mode is updated.

**mode_delta_magnitude** is the absolute value of the delta value.

**mode_delta_sign** is the sign of the delta value.

The variable ModeDelta[ i ] is set as follows:

— If loop_filter_adj_enable is equal to 0 and KeyFrame is equal to 1, ModeDelta[ i ] is set equal to 0.

— Otherwise, if loop_filter_adj_enable is equal to 0, ModeDelta[ i ] is left at the value it took in the previous frame.

— Otherwise, if mb_mode_delta_update_flag is equal to 0, ModeDelta[ i ] is set equal to 0.

— Otherwise, if mode_delta_sign is equal to 0, ModeDelta[ i ] is set equal to mode_delta_magnitude.

— Otherwise, ModeDelta[ i ] is set equal to −mode_delta_magnitude (the negative of mode_delta_magnitude).

### 7.4.4 Quantizer indices semantics

All prediction residual signals are specified via a quantized 4x4 transform coefficients for the Y, Cb, Cr or Y2 sub-blocks of a macroblock. Before the inverse transform to the spatial domain, each decoded coefficient is multiplied by one of six dequantization factors, the choice of which depends on the plane (Y, chroma = Cb or Cr, Y2) and coefficient position (DC = coefficient 0, AC = coefficients 1...15). The six values are specified using 7-bit indices into fixed tables.

**y_ac_qi** is the dequantization table index used for the luma AC coefficients (and other coefficient groups if no delta value is present).

**y_dc_delta_present** specifies if the stream contains a delta value that is added to the baseline index to obtain the luma DC coefficient dequantization index.

**y_dc_delta_magnitude** is the magnitude of the delta value. When not present, the value is inferred equal to 0.

**y_dc_delta_sign** is the sign of the delta value. When not present, the value is inferred equal to 0.

**y2_dc_delta_present** specifies if the stream contains a delta value that is added to the baseline index to obtain the Y2 block DC coefficient dequantization index.

**y2_dc_delta_magnitude** is the magnitude of the delta value. When not present, the value is inferred equal to 0.

**y2_dc_delta_sign** is the sign of the delta value. When not present, the value is inferred equal to 0.

**y2_ac_delta_present** specifies if the stream contains a delta value that is added to the baseline index to obtain the Y2 block AC coefficient dequantization index.

**y2_ac_delta_magnitude** is the magnitude of the delta value. When not present, the value is inferred equal to 0.

**y2_ac_delta_sign** is the sign of the delta value. When not present, the value is inferred equal to 0.

**uv_dc_delta_present** specifies if the stream contains a delta value that is added to the baseline index to obtain the chroma DC coefficient dequantization index.

**uv_dc_delta_magnitude** is the magnitude of the delta value. When not present, the value is inferred equal to 0.

**uv_dc_delta_sign** is the sign of the delta value. When not present the value, is inferred equal to 0.

**uv_ac_delta_present** specifies if the stream contains a delta value that is added to the baseline index to obtain the chroma AC coefficient dequantization index.

**uv_ac_delta_magnitude** is the magnitude of the delta value. When not present, the value is inferred equal to 0.

**uv_ac_delta_sign** is the sign of the delta value. When not present the value, is inferred equal to 0.

The variables y_dc_delta_q, y2_dc_delta_q, y2_ac_delta_q, uv_dc_delta_q, uv_ac_delta_q are derived as follows.

— y_dc_delta_q = (y_dc_delta_sign ? −1 : 1) * y_dc_delta_magnitude

— y2_dc_delta_q = (y2_dc_delta_sign ? −1 : 1) * y2_dc_delta_magnitude

— y2_ac_delta_q = (y2_ac_delta_sign ? −1 : 1) * y2_ac_delta_magnitude

— uv_dc_delta_q = (uv_dc_delta_sign ? −1 : 1) * uv_dc_delta_magnitude

— uv_ac_delta_q = (uv_ac_delta_sign ? −1 : 1) * uv_ac_delta_magnitude

### 7.4.5 Token probability update semantics

**coeff_prob_update_flag** specifies if the corresponding branch probability is updated in the current frame.

**coeff_prob**[ i ][ j ][ k ][ t ] is the new branch probability. When coeff_prob is not present in a bitstream, it is inferred to be equal to SavedCoeffProb[ i ][ j ][ k ][ t ].

### 7.4.6 MV probability update semantics

**mv_prob_update_flag** specifies if the corresponding MV decoding probability is updated in the current frame.

**mv_prob**[ i ][ j ] is the updated probability. When mv_prob[ i ][ j ] is not present in a bitstream, it is inferred to be equal to SavedMvProb[ i ][ j ].

## 7.5 Macroblock data semantics

### 7.5.1 Macroblock header semantics

#### 7.5.1.1 General

**segment_id** specifies to which segment the macroblock belongs. When not present in a bitstream, it is inferred as follows.

— If KeyFrame is equal to 1, then segment_id is set equal to 0.

— Otherwise (KeyFrame is equal to 0), then segment_id is set equal to SavedSegmentId[ row ][ col ].

After segment_id has been extracted or inferred, it is saved in the array SavedSegmentId as follows.

— SavedSegmentId[ row ][ col ] = segment_id

**mb_skip_coeff** specifies whether the macroblock does not contain any coded coefficients. When not present in a bitstream, it is inferred to be equal to 0.

**is_inter_mb** specifies whether the macroblock is inter coded. When not present in a bitstream, it is inferred to be equal to 0.

If is_inter_mb is equal to 0, the variable MbRefFrame[ row ][ col ] is set equal to 0.

**mb_ref_frame_sel1** and **mb_ref_frame_sel2** specify the reference frame to be used as follows.

— If mb_ref_frame_sel1 is equal to 0, MbRefFrame[ row ][ col ] is set equal to 1 (Ref0 is used as the reference frame).

— Otherwise, if mb_ref_frame_sel2 is equal to 0, MbRefFrame[ row ][ col ] is set equal to 2 (Ref1 is used as the reference frame).

— Otherwise, MbRefFrame[ row ][ col ] is set equal to 3 (Ref2 is used as the reference frame).

**mv_mode** specifies the macroblock motion vector mode according to Table 6.

**Table 6 — Name association to mv_mode**

| mv_mode | Name of mv_mode |
|---|---|
| 5 | MV_NEAREST (use "nearest" motion vector for entire MB) |
| 6 | MV_NEAR (use "next nearest") |
| 7 | MV_ZERO (use zero motion vector) |
| 8 | MV_NEW (use an explicit offset) |
| 9 | MV_SPLIT (use multiple motion vectors) |

**mv_split_mode** specifies the macroblock partitioning according to Table 7.

**Table 7 — Name association to mv_split_mode**

| mv_split_mode | Name of mv_split_mode |
|---|---|
| 0 | MV_TOP_BOTTOM (two pieces { 0..7 } and { 8..15 }) |
| 1 | MV_LEFT_RIGHT ({ 0, 1, 4, 5, 8, 9, 12, 13 } and { 2, 3, 6, 7, 10, 11, 14, 15 }) |
| 2 | MV_QUARTERS ({ 0, 1, 4, 5 }, { 2, 3, 6, 7 }, { 8, 9, 12, 13 }, { 10, 11, 14, 15 }) |
| 3 | MV_16 (every sub-block gets its own vector { 0 }..{ 15 }) |

The variable NumMvs is derived as follows.

— If mv_split_mode is MV_TOP_BOTTOM or MV_LEFT_RIGHT, NumMvs is set equal to 2.

— Otherwise, if mv_split_mode is MV_QUARTERS, NumMvs is set equal to 4.

— Otherwise (mv_split_mode is MV_16), NumMvs is set equal to 16.

**sub_mv_mode** specifies the sub-block motion vector mode for macroblocks coded using the MV_SPLIT motion vector mode according to Table 8.

**Table 8 — Name association to sub_mv_mode**

| sub_mv_mode | Name of sub_mv_mode |
|---|---|
| 5 | LEFT4X4 (use already-coded MV left) |
| 6 | ABOVE4X4 (use already-coded MV above) |
| 7 | ZERO4X4 (use zero MV) |
| 8 | NEW4X4 (explicit offset from "best") |

**intra_y_mode** specifies the luma intra prediction mode according to Table 9.

**Table 9 — Name association to intra_y_mode**

| intra_y_mode | Name of intra_y_mode |
|---|---|
| 0 | DC_PRED (predict DC using the row above and column to the left) |
| 1 | V_PRED (predict rows using the row above) |
| 2 | H_PRED (predict columns using the column to the left) |
| 3 | TM_PRED (propagate second differences) |
| 4 | B_PRED (each Y sub-block is independently predicted) |

**intra_b_mode** specifies the sub-block luma prediction mode for macroblocks coded using B_PRED mode according to Table 10.

**Table 10 — Name association to intra_b_mode**

| intra_b_mode | Name of intra_b_mode |
|---|---|
| 0 | B_DC_PRED (predict DC using the row above and column to the left) |
| 1 | B_TM_PRED (propagate second differences, as in TM_PRED mode) |
| 2 | B_VE_PRED (predict rows using the row above) |
| 3 | B_HE_PRED (predict columns using the column to the left) |
| 4 | B_LD_PRED (left and down, 45° diagonal prediction) |
| 5 | B_RD_PRED (right and down, 45° diagonal prediction) |
| 6 | B_VR_PRED (vertical right diagonal prediction) |
| 7 | B_VL_PRED (vertical left diagonal prediction) |
| 8 | B_HD_PRED (horizontal down prediction) |
| 9 | B_HU_PRED (horizontal up prediction) |

**intra_uv_mode** specifies the chroma intra prediction mode using the values 0..3 with the same interpretation as in the semantics for intra_y_mode.

#### 7.5.1.2 MV component semantics

**mv_long** specifies whether long coding is used for the motion vector.

**mv_bit** specifies a single bit of a long coded motion vector component.

**mv_short** specifies a short coded motion vector component.

**mv_sign** specifies whether the motion vector component is negated.

The computed value for the motion vector component (c) will be a value between −1023 and 1023 and represents an offset in quarter sample units.

### 7.5.2 Prediction residual block semantics

firstCoeff is a parameter that specifies whether the DC coefficient is skipped in this block.

**coeff_token** defines the value of the coefficient, the value range of the coefficient or the end of block according to Table 11.

**Table 11 — Name association to token**

| coeff_token | Name of token |
|---|---|
| 0 | DCT_0 (value 0) |
| 1 | DCT_1 (value 1) |
| 2 | DCT_2 (value 2) |
| 3 | DCT_3 (value 3) |
| 4 | DCT_4 (value 4) |
| 5 | DCT_CAT1 (values 5 ... 6) |
| 6 | DCT_CAT2 (values 7 ... 10) |
| 7 | DCT_CAT3 (values 11 ...18) |
| 8 | DCT_CAT4 (values 19 ...34) |
| 9 | DCT_CAT5 (values 35 ... 66) |
| 10 | DCT_CAT6 (values 67 ... 2048) |
| 11 | DCT_EOB (end of block) |

The variables catBase, catNum and catProbs are derived as follows.

— If coeff_token is equal to DCT_CAT1, catBase is set equal to 5, catNum is set equal to 1 and catProbs is set equal to the array { 159 }.

— If coeff_token is equal to DCT_CAT2, catBase is set equal to 7, catNum is set equal to 2 and catProbs is set equal to the array { 165, 145 }.

— If coeff_token is equal to DCT_CAT3, catBase is set equal to 11, catNum is set equal to 3 and catProbs is set equal to the array { 173, 148, 140 }.

— If coeff_token is equal to DCT_CAT4, catBase is set equal to 19, catNum is set equal to 4 and catProbs is set equal to the array { 176, 155, 140, 135 }.

— If coeff_token is equal to DCT_CAT5, catBase is set equal to 35, catNum is set equal to 5 and catProbs is set equal to the array { 180, 157, 141, 134, 130 }.

— If coeff_token is equal to DCT_CAT6, catBase is set equal to 67, catNum is set equal to 11 and catProbs is set equal to the array { 254, 254, 243, 230, 196, 177, 153, 140, 133, 130, 129 }.

**extra_bit** specifies an increment to the value of the coefficient.

**coef_sign** indicates the sign of the coefficient.

## 8 Decoding process

### 8.1 General

Decoders shall produce output frames that are equal in value and have the same output order as those produced by the decoding process specified herein. Annex A normatively specifies the limits on bitstreams that shall be accepted by conforming decoders. Annex B informatively describes an encoder that produces a conforming bitstream.

Input to this process is a sequence of coded frames.

Output from this process is a sequence of decoded frames.

For each coded frame in turn, the decoding process operates as follows.

a) The syntax elements in the compressed header are extracted as specified in Clause 6 and Clause 7 and the header update process as specified in 8.2 is invoked as soon as all the syntax elements in compressed_header have been extracted.

b) The macroblock headers are extracted as specified in Clause 6 and Clause 7 and the coding context update process as specified in 8.3 is invoked for each macroblock as soon as all the syntax elements in mb_header have been extracted with the inputs StartRow = row and StartCol = col.

c) The sub-block motion vector reconstruction process as specified in 8.4.6.2 is invoked each time sub_mv_mode is decoded as soon as the associated MV component (if any) has been extracted.

d) The macroblock decoding process as specified in 8.4 is invoked for each macroblock as soon as all the syntax elements in prediction_residual_data have been extracted with the inputs StartRow = row and StartCol = col.

e) If loop_filter_level is not equal to 0 and version_number is less than or equal to 1, the loop filter process as specified in 8.5 is invoked once all macroblocks have been decoded.

f) If show_frame is equal to 1, the output process as specified in 8.6 is invoked.

g) A reference frame update process as specified in 8.7 is invoked.

## 8.2 Header update process

This process is invoked as soon as all the syntax elements in the compressed header have been extracted.

Inputs to this process are the syntax elements in the uncompressed and compressed header.

Output from this process is an updated set of saved probabilities.

When refresh_entropy_probs is equal to 1, the probabilities used for decoding mode, motion vectors and tokens are saved for use in subsequent frames as follows:

```
for ( i = 0; i < 4; i++ )
    SavedIntra16x16Prob[ i ] = intra_16x16_prob[ i ]
for ( i = 0; i < 3; i++ )
    SavedIntraChromaProb[ i ] = intra_chroma_prob[ i ]
for ( i = 0; i < 4; i++ )
    for ( j = 0; j < 8; j++ )
        for ( k = 0; k < 3; k++ )
            for ( t = 0; t < 11; t++ )
                SavedCoeffProb[ i ][ j ][ k ][ t ] = coeff_prob[ i ][ j ][ k ][ t ]
for ( i = 0; i < 2; i++ )
    for ( j = 0; j < 19; j++ )
        SavedMvProb[ i ][ j ] = mv_prob[ i ][ j ]
```

NOTE The probabilities are reset on key frames and persist across subsequent P frames. The syntax of each frame indicates whether to update the probabilities or to use the saved ones. When updated probabilities are used, refresh_entropy_probs controls whether to also use the update for subsequent frames.

## 8.3 Coding context update process

The coding context update process is invoked for each macroblock as soon as all the syntax elements in mb_header have been extracted.

The inputs to the process are the variables StartRow and StartCol specifying the macroblock location.

The outputs of the process are modifications to the context variables.

If KeyFrame is equal to 1, the intra mode contexts are updated as follows.

— If intra_y_mode is equal to DC_PRED, the following applies:

```
for ( i = 0; i < 4; i++ ) {
    LeftIntraMode[ i ] = B_DC_PRED
    AboveIntraMode[ StartCol ][ i ] = B_DC_PRED
}
```

— Otherwise, if intra_y_mode is equal to V_PRED, the following applies:

```
for ( i = 0; i < 4; i++ ) {
    LeftIntraMode[ i ] = B_VE_PRED
    AboveIntraMode[ StartCol ][ i ] = B_VE_PRED
  }
```

— Otherwise, if intra_y_mode is equal to H_PRED, the following applies:

```
  for ( i = 0; i < 4; i++ ) {
    LeftIntraMode[ i ] = B_HE_PRED
    AboveIntraMode[ StartCol ][ i ] = B_HE_PRED
  }
```

— Otherwise, if intra_y_mode is equal to TM_PRED, the following applies

```
  for ( i = 0; i < 4; i++ ) {
    LeftIntraMode[ i ] = B_TM_PRED
    AboveIntraMode[ StartCol ][ i ] = B_TM_PRED
  }
```

— Otherwise, if intra_y_mode is equal to B_PRED, the following applies:

```
  for ( i = 0; i < 4; i++ ) {
    LeftIntraMode[ i ] = intra_b_mode[ i * 4 + 3 ]
    AboveIntraMode[ StartCol ][ i ] = intra_b_mode[ 12 + i ]
  }
```

If is_inter_mb is equal to 1, the motion vector reconstruction process specified in 8.4.6 is invoked.

Depending on the value of is_inter_mb, the following applies:

— If is_inter_mb is equal to 0, the following applies:

```
    MbSplit[ StartRow ][ StartCol ] = 0
```

— Otherwise, if mv_mode is equal to MV_SPLIT, the following applies:

```
    MbSplit[ StartRow ][ StartCol ] = 1
    MbMvs[ StartRow ][ StartCol ] = SubMvs[ StartRow ][ StartCol ][ 15 ]
```

— Otherwise, the following applies:

```
    MbSplit[ StartRow ][ StartCol ] = 0
    MbMvs[ StartRow ][ StartCol ] = Mv
```

## 8.4 Macroblock decoding process

### 8.4.1 General

The macroblock decoding process is invoked for each macroblock in raster order.

The inputs to this process are the variables StartRow and StartCol specifying the macroblock location.

The following ordered steps apply.

a) The prediction_residual decoding process as specified in 8.4.2 is invoked.

b) Depending on the value of is_inter_mb, the following applies.

1) If is_inter_mb is equal to 0, the intra prediction process as specified in 8.4.2 is invoked with StartRow and StartCol as inputs.

2) Otherwise (is_inter_mb is equal to 1), the inter prediction process as specified in 8.4.3 is invoked with StartRow and StartCol as inputs.

### 8.4.2 Intra prediction process

#### 8.4.2.1 General

The intra prediction process is invoked for intra coded macroblocks in a P frame and for all macroblocks in a key frame.

Inputs to this process are:

— the already reconstructed samples in the current frame CurrPic;

— the prediction residual values for the current macroblock in PredictionResidual;

— the variables StartRow and StartCol specifying the location of the macroblock.

Outputs of this process are the samples in the current frame CurrPic corresponding to the current macroblock.

The intra prediction process is specified as follows.

— The block intra prediction process specified in 8.4.2.2 is invoked with the variable colourPlane set equal to 1, predMode set equal to intra_uv_mode, x set equal to StartCol * 8, y set equal to StartRow * 8 and the variable log2Size set equal to 3.

— The block intra prediction process specified in 8.4.2.2 is invoked with the variable colourPlane set equal to 2, predMode set equal to intra_uv_mode, x set equal to col * 8 y set equal to StartRow * 8, and the variable log2Size set equal to 3.

— Depending on the value of intra_y_mode, the following applies:

  — If intra_y_mode is equal to B_PRED, the following applies:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        The sub-block intra prediction process specified in 8.4.2.2 is invoked with
        variables x, y, StartRow and StartCol as inputs.
```

  — Otherwise, the block intra prediction process specified in 8.4.2.2 is invoked with the variable colourPlane set equal to 0, predMode set equal to intra_y_mode, x set equal to StartCol * 16, y set equal to StartRow * 16 and the variable log2Size set equal to 4.

#### 8.4.2.2 Block intra prediction process

The inputs to this process are:

— a variable colourPlane specifying which plane is being predicted;

— a variable predMode specifying the direction of intra prediction;

— variables x and y specifying the location of the top left sample of the current macroblock in the CurrPic[ colourPlane ] array;

— a variable log2Size specifying the size of the intra block.

Outputs of this process are the samples in the current frame CurrPic corresponding to the specified plane of the current macroblock.

The variable BlockSize is specified as 1 << log2Size.

The array aboveRow is specified depending on the variable y as:

— If y is equal to 0, aboveRow[ i ] is set equal to 127 with i = 0..BlockSize−1.

— Otherwise, aboveRow[ i ] is set equal to CurrPic[ colourPlane ][ x + i ][ y − 1] with i = 0..BlockSize−1.

The array leftCol is specified depending on the variable x as:

— If x is equal to 0, leftCol [ i ] is set equal to 129 with i = 0..BlockSize−1.

— Otherwise, leftCol [ i ] is set equal to CurrPic[ colourPlane ][ x − 1 ][ y + i ] with i = 0..BlockSize−1.

The variable aboveLeftPixel is specified depending on x and y as:

— If y is equal to 0, aboveLeftPixel is set equal to 127.

— Otherwise, if x is equal to 0, aboveLeftPixel is set equal to 129.

— Otherwise, aboveLeftPixel is set equal to CurrPic[ colourPlane ][ x − 1 ][ y − 1 ].

A prediction array is constructed as follows.

— If predMode is equal to V_PRED, prediction[ x2 ][ y2 ] is set equal to aboveRow[ x2 ] with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1. (Each row of the block is filled with a copy of aboveRow.)

— Otherwise, if predMode is equal to H_PRED, prediction[ x2 ][ y2 ] is set equal to leftCol[ y2 ] with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1. (Each column of the block is filled with a copy of leftCol.)

— Otherwise, if predMode is equal to DC_PRED and x is non-zero and y is non-zero, prediction[ x2 ][ y2 ] is set equal to avg with x2 = 0.. BlockSize −1 and y2 = 0.. BlockSize −1. The variable avg (the average of the samples in union of aboveRow and leftCol) is specified as follows:

```
sum = 0
    for ( i = 0; i < BlockSize; i++ ) {
        sum += leftCol[ i ]
        sum += aboveRow[ i ]
    }
avg = (sum + BlockSize) >> (log2Size + 1)
```

— Otherwise, if predMode is equal to DC_PRED and x is non-zero and y is zero, prediction[ x2 ][ y2 ] is set equal to leftAvg with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1. The variable leftAvg is specified as follows:

```
sum = 0
for ( i = 0; i < BlockSize; i++ )
```

```
        sum += leftCol[ i ]
    leftAvg = (sum + (1 << (log2Size − 1) ) ) >> log2Size
```

— Otherwise, if predMode is equal to DC_PRED and x is zero and y is non-zero, prediction[ x2 ][ y2 ] is set equal to aboveAvg with x2 = 0.. BlockSize −1 and y2 = 0.. BlockSize −1. The variable aboveAvg is specified as follows:

```
    sum = 0
    for ( i = 0; i < BlockSize; i++ )
        sum += aboveRow[ i ]
    aboveAvg = (sum + (1 << (log2Size − 1) ) ) >> log2Size
```

— Otherwise, if predMode is DC_PRED, prediction[ x2 ][ y2 ] is set equal to 128 with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1.

— Otherwise (predMode is equal to TM_PRED), prediction[ x2 ][ y2 ] is set equal to Clip1( leftCol[ y2 ] + aboveRow[ x2 ] − aboveLeftPixel ) with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1.

— Once the prediction array has been constructed, the macroblock is reconstructed by setting CurrPic[ colourPlane ][ x + x2][ y + y2] equal to Clip1( prediction[ x2 ][ y2] + PredictionResidual[ colourPlane ][ x2 ][ y2 ] ) with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1.

If clamping_type is equal to 1, prediction[ x2 ][ y2 ] + PredictionResidual[ colourPlane ][ x2 ][ y2 ] shall be in the range 0 to 255 with x2 = 0..BlockSize−1 and y2 = 0..BlockSize−1.

#### 8.4.2.3 Sub-block intra prediction process

The inputs to this process are:

— variables StartRow and StartCol specifying the current macroblock,

— variables x and y specifying which sub-block in that macroblock is being predicted.

The output is to set the values in CurrPic[ 0 ] corresponding to the current sub-block.

The array leftCol is specified depending on the variable x and StartCol as:

— If x is equal to 0 and StartCol is equal to 0, leftCol[ i ] is set equal to 129 with i = 0..3.

— Otherwise, leftCol[ i ] is set equal to CurrPic[ 0 ][ StartCol * 16 + x * 4 − 1 ][ StartRow * 16 + y * 4 + I ] with i = 0..3.

The variable aboveLeftPixel is specified depending on x, y, StartRow, StartCol as:

— If y is equal to 0 and StartRow is equal to 0, aboveLeftPixel is set equal to 127.

— Otherwise, if x is equal to 0 and StartCol is equal to 0, aboveLeftPixel is set equal to 129.

— Otherwise, aboveLeftPixel is set equal to CurrPic[ 0 ][ StartCol * 16 + x * 4 − 1 ][ StartRow * 16 + y * 4 − 1 ].

The first 4 entries of array aboveRow are specified as:

— If y is equal to 0 and StartRow is equal to 0, aboveRow[ i ] is set equal to 127 with i = 0..3.

— Otherwise, aboveRow[ i ] is set equal to CurrPic[ 0 ][ StartCol * 16 + x * 4 + i ][ StartRow * 16 + y * 4 − 1 ] with i = 0..3.

The second 4 entries of array aboveRow are specified as:

— If y is equal to 0 and StartRow is equal to 0, aboveRow[ i ] is set equal to 127 with i = 4..7.

— Otherwise, if x is equal to 3 and StartRow is equal to 0, aboveRow[ i ] is set equal to 127 with i = 4..7.

— Otherwise, if x is equal to 3 and StartCol is less than MbCols − 1, aboveRow[ i ] is set equal to CurrPic[ 0 ][ (StartCol + 1) * 16 + i − 4 ][ StartRow * 16 − 1 ] with i = 4..7.

— Otherwise, if x is equal to 3 (and StartCol is equal to MbCols − 1), aboveRow[ i ] is set equal to CurrPic[ 0 ][ StartCol * 16 + 15 ][ StartRow * 16 − 1 ] with i = 4..7.

— Otherwise, aboveRow[ i ] is set equal to CurrPic[ 0 ] [ StartCol * 16 + x * 4 + I ][ StartRow * 16 + y * 4 − 1 ] with i = 4..7.

The entry aboveRow[ −1 ] is set equal to aboveLeftPixel.

The variable IntraPredBlock is set equal to 4 * y + x.

The variable predMode is set equal to intra_b_mode[ IntraPredBlock ].

A prediction array is constructed as follows

— If predMode is B_VE_PRED, the following applies:

```
for ( y2 = 0; y2 < 4; y2++ )
    for ( x2 = 0; x2 < 4; x2++ )
        prediction[ x2 ][ y2 ] = (aboveRow[ x2 − 1 ] + 2 * aboveRow[ x2 ]
            + aboveRow[ x2 + 1 ] + 2) >> 2
```

— Otherwise, if predMode is B_HE_PRED, the following applies:

```
for ( y2 = 0; y2 < 4; y2++ )
    for ( x2 = 0; x2 < 4; x2++ )
        if ( x2 < 3 )
            prediction[ x2 ][ y2 ] = (leftCol[ y2 − 1 ] + 2 * leftCol[ y2 ]
                + leftCol[ y2 + 1 ] + 2) >> 2
        else
            prediction[ x2 ][ y2 ] = (leftCol[ y2 − 1 ] + 3 * leftCol[ y2 ] + 2) >> 2
```

— Otherwise, if predMode is B_DC_PRED, prediction[ x2 ][ y2 ] is set equal to avg with x2 = 0..3 and y2 = 0..3 where avg is specified as:

```
sum = 0
for ( i = 0; i < 4; i++ ) {
    sum += leftCol[ i ]
```

```
        sum += aboveRow[ i ]
    }
    avg = (sum + 4) >> 3
```

— Otherwise, if predMode is B_TM_PRED, the following applies:

```
    for ( y2 = 0; y2 < 4; y2++ )
        for ( x2 = 0; x2 < 4; x2++ )
            prediction[ x2 ][ y2 ] = Clip1( leftCol[ y2 ] + aboveRow[ x2 ] − aboveLeftPixel )
```

— Otherwise, if predMode is B_LD_PRED, the following applies:

```
    for ( y2 = 0; y2 < 4; y2++ )
        for ( x2 = 0; x2 < 4; x2++ )
            if ( x2 + y2 < 6 )
                prediction[ x2 ][ y2 ] = (aboveRow[ x2 + y2 ] + 2 * aboveRow[ x2 + y2 + 1 ]
                                          + aboveRow[ x2 + y2 + 2 ] + 2) >> 2
            else
                prediction[ x2 ][ y2 ] = (aboveRow[ x2 + y2 ] + 3 * aboveRow[ x2 + y2 + 1 ]
                                          + 2) >> 2
```

— Otherwise, if predMode is B_RD_PRED, the following applies:

```
    for ( y2 = 0; y2 < 4; y2++ )
        for ( x2 = 0; x2 < 4; x2++ )
            if ( x2 − y2 < 0 )
                    prediction[ x2 ][ y2 ] = (leftCol[ y2 − x2 ] + 2 * leftCol [ y2 − x2 − 1 ]
                                              + leftCol [ y2 − x2 − 2 ] + 2) >> 2
            else if ( x2 − y2 == 0)
                prediction[ x2 ][ y2 ] = (leftCol[ 0 ] + 2 * aboveLeftPixel
                                          + aboveRow[ 0 ] + 2) >> 2
            else
                prediction[ x2 ][ y2 ] = (aboveRow[ x2 − y2 − 2 ] + 2 * aboveRow[ x2 − y2 − 1 ]
                                          + aboveRow[ x2 − y2 ] + 2) >> 2
```

— Otherwise, if predMode is B_VR_PRED, the following applies:

prediction[ 0 ][ 3 ] = (leftCol[ 2 ] + 2 * leftCol[ 1 ] + leftCol[ 0 ] + 2) >> 2

prediction[ 0 ][ 2 ] = (leftCol[ 1 ] + 2 * leftCol[ 0 ] + aboveLeftPixel + 2) >> 2

prediction[ 1 ][ 3 ] = prediction[ 0 ][ 1 ] = (leftCol[ 0 ]
+ 2 * aboveLeftPixel + aboveRow[ 0 ] + 2) >> 2

prediction[ 1 ][ 2 ] = prediction[ 0 ][ 0 ] = (aboveLeftPixel + aboveRow[ 0 ] + 1) >> 1

prediction[ 2 ][ 3 ] = prediction[ 1 ][ 1 ] = (aboveLeftPixel + 2 * aboveRow[ 0 ]
+ aboveRow[ 1 ] + 2) >> 2

prediction[ 2 ][ 2 ] = prediction[ 1 ][ 0 ] = (aboveRow[ 0 ] + aboveRow[ 1 ] + 1) >> 1

prediction[ 3 ][ 3 ] = prediction[ 2 ][ 1 ] = (aboveRow[ 0 ] + 2 * aboveRow[ 1 ]
+ aboveRow[ 2 ] + 2) >> 2

prediction[ 3 ][ 2 ] = prediction[ 2 ][ 0 ] = (aboveRow[ 1 ] + aboveRow[ 2 ] + 1) >> 1

prediction[ 3 ][ 1 ] = (aboveRow[ 1 ] + 2 * aboveRow[ 2 ] + aboveRow[ 3 ] + 2) >> 2

prediction[ 3 ][ 0 ] = (aboveRow[ 2 ] + aboveRow[ 3 ] + 1) >> 1

— Otherwise, if predMode is B_VL_PRED, the following applies:

prediction[ 0 ][ 0 ] = (aboveRow[ 0 ] + aboveRow[ 1 ] + 1) >> 1

prediction[ 0 ][ 1 ] = (aboveRow[ 0 ] + 2 * aboveRow[ 1 ] + aboveRow[ 2 ] + 2) >> 2

prediction[ 0 ][ 2 ] = prediction[ 1 ][ 0 ] = (aboveRow[ 1 ] + aboveRow[ 2 ] + 1) >> 1

prediction[ 1 ][ 1 ] = prediction[ 0 ][ 3 ] = (aboveRow[ 1 ] + 2 * aboveRow[ 2 ]
+ aboveRow[ 3 ] + 2) >> 2

prediction[ 1 ][ 2 ] = prediction[ 2 ][ 0 ] = (aboveRow[ 2 ] + aboveRow[ 3 ] + 1) >> 1

prediction[ 1 ][ 3 ] = prediction[ 2 ][ 1 ] = (aboveRow[ 2 ] + 2 * aboveRow[ 3 ]
+ aboveRow[ 4 ] + 2) >> 2

prediction[ 2 ][ 2 ] = prediction[ 3 ][ 0 ] = (aboveRow[ 3 ] + aboveRow[ 4 ] + 1) >> 1

prediction[ 2 ][ 3 ] = prediction[ 3 ][ 1 ] = (aboveRow[ 3 ] + 2 * aboveRow[ 4 ]
+ aboveRow[ 5 ] + 2) >> 2

prediction[ 3 ][ 2 ] = (aboveRow[ 4 ] + 2 * aboveRow[ 5 ] + aboveRow[ 6 ] + 2) >> 2

prediction[ 3 ][ 3 ] = (aboveRow[ 5 ] + 2 * aboveRow[ 6 ] + aboveRow[ 7 ] + 2) >> 2

— Otherwise, if predMode is B_HD_PRED, the following applies:

prediction[ 0 ][ 3 ] = (leftCol[ 3 ] + leftCol[ 2 ] + 1) >> 1

prediction[ 1 ][ 3 ] = (leftCol[ 3 ] + 2 * leftCol[ 2 ] + leftCol[ 1 ] + 2) >> 2

prediction[ 0 ][ 2 ] = prediction[ 2 ][ 3 ] = (leftCol[ 2 ] + leftCol[ 1 ] + 1) >> 1

prediction[ 1 ][ 2 ] = prediction[ 3 ][ 3 ] = (leftCol [ 2 ] + 2 * leftCol [ 1 ] + leftCol[ 0 ] + 2) >> 2

prediction[ 2 ][ 2 ] = prediction[ 0 ][ 1 ] = (leftCol[ 1 ] + leftCol[ 0 ] + 1) >> 1

prediction[ 3 ][ 2 ] = prediction[ 1 ][ 1 ] = (leftCol [ 1 ] + 2 * leftCol [ 0 ] + aboveLeftPixel + 2) >> 2

prediction[ 2 ][ 1 ] = prediction[ 0 ][ 0 ] = (leftCol[ 0 ] + aboveLeftPixel + 1) >> 1

prediction[ 3 ][ 1 ] = prediction[ 1 ][ 0 ] = (leftCol [ 0 ] + 2 * aboveLeftPixel + aboveRow [ 0 ] + 2) >> 2

— Otherwise (predMode is B_HU_PRED), the following applies:

prediction[ 0 ][ 0 ] = (leftCol[ 0 ] + leftCol[ 1 ] + 1) >> 1

prediction[ 1 ][ 0 ] = (leftCol[ 0 ] + 2 * leftCol[ 1 ] + leftCol[ 2 ] + 2) >> 2

prediction[ 2 ][ 0 ] = prediction[ 0 ][ 1 ] = (leftCol[ 1 ] + leftCol[ 2 ] + 1) >> 1

prediction[ 3 ][ 0 ] = prediction[ 1 ][ 1 ] = (leftCol [ 1 ] + 2 * leftCol [ 2 ] + leftCol[ 3 ] + 2) >> 2

prediction[ 2 ][ 1 ] = prediction[ 0 ][ 2 ] = (leftCol[ 2 ] + leftCol[ 3 ] + 1) >> 1

prediction[ 3 ][ 1 ] = prediction[ 1 ][ 2 ] = (leftCol [ 2 ] + 3 * leftCol [ 3 ] + 2) >> 2

prediction[ 2 ][ 2 ] = prediction[ 3 ][ 2 ] = prediction[ 0 ][ 3 ] = prediction[ 1 ][ 3 ] = prediction[ 2 ][ 3 ] = prediction[ 3 ][ 3 ] = leftCol [ 3 ]

NOTE 1 The first four intra_b_modes are similar to their corresponding full block modes only acting on low-pass filtered previous samples.

NOTE 2 The remaining six "diagonal" modes subdivide the prediction buffer into diagonal lines. All the samples on each line are assigned the same value; this value is (a smoothed or synthetic version of) an already-constructed predictor value lying on the same line. The first two use lines at +/− 45° from horizontal (or, equivalently, vertical), that is, lines whose slopes are +/− 1. The remaining 4 diagonal modes use lines whose slopes are +/− 2 and +/− 0.5, which means there is often a need to "synthesize" predictor samples midway between two actual predictors by taking their average. All these “diagonal” modes are unique to sub-block prediction and have no full-block analogs.

Once the prediction array has been constructed, the sub-block is reconstructed by setting CurrPic[ 0 ][ col * 16 + x * 4 + x2 ][ row * 16 + y * 4 + y2 ] equal to Clip1( prediction[ x2 ][ y2 ] + PredictionResidual[ 0 ][ x * 4 + x2 ][ y * 4 + y2 ] ) with x2 = 0..3 and y2 = 0..3.

If clamping_type is equal to 1, prediction[ x2 ][ y2 ] + PredictionResidual[ colourPlane ][ x * 4 + x2 ][ y * 4 + y2 ] shall be in the range 0 to 255 with x2 = 0..3 and y2 = 0..3.

### 8.4.3 Inter prediction process

#### 8.4.3.1 General

Inputs to this process are:

— the previous reference frames in Ref0, Ref1 and Ref2;

— the variables StartRow and StartCol specifying the current macroblock.

Outputs from this process are the samples in the current frame CurrPic corresponding to the current macroblock.

The sub-block inter prediction process specified in 8.4.3.3 is invoked with colourPlane set equal to 0, mv set equal to SubMvs[ row ][ col ][ y * 4 + x ] and StartRow, StartCol, x, y as inputs with x = 0..3, y = 0..3.

The chroma prediction process specified in 8.4.3.2 is invoked with StartRow, StartCol, x, y as inputs with x = 0..1, y = 0..1.

#### 8.4.3.2 Chroma prediction process

Inputs to this process are:

— the variables x and y specifying which chroma sub-block is being decoded;

— the variables StartRow and StartCol specifying the current macroblock.

The output of this process is to predict the value in the chroma sub-blocks based on motion compensated filtering.

The chroma motion vector is computed in eighth-sample units by the following:

```
for ( comp = 0; comp < 2; comp++ ) {
    s = 0
    for ( y2 = 0; y2 < 2; y2++ )
        for ( x2 = 0; x2 < 2; x2++ )
            s += 2 * SubMvs[ StartRow ][ StartCol ][ x * 2 + x2 + 4 * ( y * 2 + y2 ) ][ comp ]
    chromaMv[ comp ] = (s >= 0) ? (s + 4) >> 3 : −((−s + 4) >> 3)
    if ( version_number == 3)
        chromaMv[ comp ] &= ~7
}
```

NOTE The shift divides by 8 (not 4) because chroma samples have twice the effective width and height of luma samples. The shifting of s is slightly cumbersome to make clear the behaviour for negative values.

The sub-block inter prediction process specified in 8.4.3.3 is invoked with mv set equal to chromaMv, colourPlane = 1..2 and x, y, StartRow, StartCol as inputs.

#### 8.4.3.3 Sub-block inter prediction process

The inputs to this process are:

— a variable colourPlane;

— a motion vector mv (in quarter luma sample units if colourPlane is equal to 0, otherwise, in eighth luma sample units);

— variables x and y giving the sub-block location;

— variables StartRow and StartCol giving the macroblock location.

Outputs from this process are the samples in the current frame CurrPic[ colourPlane ] corresponding to the current sub-block.

The motion vector and coordinates are adjusted as follows.

— If colourPlane is equal to 0, the following applies:

  mv[ 0 ] = mv[ 0 ] * 2

  mv[ 1 ] = mv[ 1 ] * 2

  xBase = StartCol * 16 + x * 4

  yBase = StartRow * 16 + y * 4

  lastX = MbCols * 16 − 1

  lastY = MbRows * 16 − 1

— Otherwise (colourPlane is not equal to 0), the following applies:

  xBase = StartCol * 8 + x * 4

  yBase = StartRow * 8 + y * 4

  lastX = MbCols * 8 − 1

  lastY = MbRows * 8 − 1

The variable ref is set to the reference frame as follows:

— If MbRefFrame[ StartRow ][ StartCol ] is equal to 1, ref is set to Ref0.

— Otherwise, if MbRefFrame[ StartRow ][ StartCol ] is equal to 2, ref is set to Ref1.

— Otherwise, (MbRefFrame[ StartRow ][ StartCol ] is equal to 3), ref is set to Ref2.

The sub-sample interpolation is effected via two one-dimensional convolutions. First, a horizontal filter is used to build up a temporary array and then this array is vertically filtered to obtain the final prediction. The fractional parts of the motion vectors determine the filtering process. If the fractional part is zero, then the filtering is equivalent to a straight sample copy. Table 12 specifies the contents of the arrays CubicFilters and BilinearFilters.

The filtering is applied as follows:

— The array horizFilter is set as follows:

  — If version_number is equal to 0, horizFilter is set equal to CubicFilters[ mv[ 1 ] & 7 ].

  — Otherwise, if version_number is less than 3, horizFilter is set equal to BilinearFilters[ mv[ 1 ] & 7 ].

  — Otherwise (version_number is equal to 3), horizFilter is set equal to BilinearFilters[ 0 ].

— The array vertFilter is set as follows:

  — If version_number is equal to 0, vertFilter is set equal to CubicFilters[ mv[ 0 ] & 7].

  — Otherwise, if version_number is less than 3, vertFilter is set equal to BilinearFilters[ mv[ 0 ] & 7].

  — Otherwise (version_number is equal to 3), vertFilter is set equal to BilinearFilters[ 0 ].

— The array intermediate is specified as follows

```
for ( y2 = −2; y2 < 7; y2++ )
  for ( x2 = 0; x2 < 4; x2++ ) {
    s = 64
    for ( t = 0; t < 6; t++ )
      s += horizFilter[ t ] * ref[ colourPlane ][ Clip3( 0, lastX, xBase + x2 + t − 2 + (mv[ 1 ] >> 3) ) ]
            [ Clip3( 0, lastY, yBase + y2 + (mv[ 0 ] >> 3) ) ]
    intermediate[ x2 ][ y2] = Clip1( s >> 7 )
  }
```

— The array predicted is specified as follows:

```
for ( y2 = 0; y2 < 4; y2++ )
  for ( x2 = 0; x2 < 4; x2++ ) {
    s = 64
    for ( t = 0; t < 6; t++ )
      s += vertFilter[ t ] * intermediate[ x2 ][ y2 + t − 2 ]
    prediction[ x2 ][ y2] = Clip1( s >> 7 )
  }
```

Once the prediction array has been constructed, the sub-block is reconstructed by setting CurrPic[ colourPlane ][ xBase + x2 ][ yBase + y2 ] equal to Clip1( prediction[ x2 ][ y2 ] + PredictionResidual[ colourPlane ][ x * 4 + x2 ][ y * 4 + y2 ] ) with x2 = 0..3 and y2 = 0..3.

If clamping_type is equal to 1, prediction[ x2 ][ y2 ] + PredictionResidual[ colourPlane ][ x * 4 + x2 ] [ y * 4 + y2 ] shall be in the range 0 to 255 with x2 = 0..3 and y2 = 0..3.

**Table 12 — Interpolation coefficients**

| |
|---|
| BilinearFilters[ 8 ][ 6 ] = { |
| { 0, 0, 128, 0, 0, 0 }, |
| { 0, 0, 112, 16, 0, 0 }, |
| { 0, 0, 96, 32, 0, 0 }, |
| { 0, 0, 80, 48, 0, 0 }, |
| { 0, 0, 64, 64, 0, 0 }, |
| { 0, 0, 48, 80, 0, 0 }, |
| { 0, 0, 32, 96, 0, 0 }, |
| { 0, 0, 16, 112, 0, 0 } |
| } |
| CubicFilters[ 8 ][ 6 ] = { /* indexed by displacement */ |
| { 0, 0, 128, 0, 0, 0 }, /* degenerate whole-sample */ |
| { 0, −6, 123, 12, −1, 0 }, /* 1÷8 position */ |
| { 2, −11, 108, 36, −8, 1 }, /* 1÷4 position */ |
| { 0, −9, 93, 50, −6, 0 }, /* 3÷8 position */ |
| { 3, −16, 77, 77, −16, 3 }, /* 1÷2 position (is symmetric) */ |
| { 0, −6, 50, 93, −9, 0 }, /* 5÷8 position (is reverse of 3÷8 position) */ |

**Table 12** *(continued)*

| |
|---|
| { 1, −8, 36, 108, −11, 2 }, /* 3÷4 position (is reverse of 1÷4 position) */ |
| { 0, −1, 12, 123, −6, 0 } /* 7÷8 position (is reverse of 1÷8 position) */ |
| } |

### 8.4.4 Prediction residual decoding process

#### 8.4.4.1 General

Inputs to this process are the syntax elements from the prediction residual data for this macroblock.

Outputs of this process are the prediction residual sample values in the array PredictionResidual[ colourPlane ][ x ][ y ].

The prediction residual is specified as follows:

— If mb_skip_coeff is equal to 1, the prediction residual is cleared as follows:

```
for ( y = 0; y < 16; y++ )
    for ( x = 0; x < 16; x++ )
        PredictionResidual[ 0 ][ x ][ y ] = 0
for ( colourPlane = 1; colourPlane < 3; colourPlane++ )
    for ( y = 0; y < 8; y++ )
        for ( x = 0; x < 8; x++ )
            PredictionResidual[ colourPlane ][ x ][ y ] = 0
```

— Otherwise, the following ordered steps apply:

  — The dequantization process as specified in 8.4.4.2 is invoked.

  — If HasY2 is equal to 1, the Walsh-Hadamard inversion process as specified in 8.4.4.3 is invoked.

  — The inverse transform process as specified in 8.4.4.4 is invoked.

#### 8.4.4.2 Dequantization

In this process, the transform coefficients are dequantized.

The input to this process is the syntax element segment_id and the variable HasY2.

The variable q is derived as follows:

— If segmentation_enabled is equal to 0, q is set equal to y_ac_qi.

— Otherwise, if segment_feature_mode is equal to 1, q is set equal to Quantizer[ segment_id ].

— Otherwise, q is set equal to y_ac_qi + Quantizer[ segment_id ].

The function dc_q( b ) is specified as the entry at index Clip3( 0, 127, b ) in Table 13.

**Table 13 — Dequantizaton factors for DC coefficients**

| |
|---|
| dc_qlookup[ 128 ] = { |
| 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, |
| 16, 17, 17, 18, 19, 20, 20, 21, 21, 22, 22, 23, 23, |

**Table 13** *(continued)*

| |
|---|
| 24, 25, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, |
| 36, 37, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 46, |
| 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, |
| 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, |
| 73, 74, 75, 76, 76, 77, 78, 79, 80, 81, 82, 83, 84, |
| 85, 86, 87, 88, 89, 91, 93, 95, 96, 98, 100, 101, 102, |
| 104, 106, 108, 110, 112, 114, 116, 118, 122, 124, 126, 128, 130, |
| 132, 134, 136, 138, 140, 143, 145, 148, 151, 154, 157, |
| } |

The function ac_q( b ) is specified as the entry at index Clip3( 0, 127, b ) in Table 14.

**Table 14 — Dequantization values for AC coefficients**

| ac_qlookup[ 128 ] = { |
|---|
| 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, |
| 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, |
| 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, |
| 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, |
| 56, 57, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, |
| 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, |
| 106, 108, 110, 112, 114, 116, 119, 122, 125, 128, 131, 134, 137, |
| 140, 143, 146, 149, 152, 155, 158, 161, 164, 167, 170, 173, 177, |
| 181, 185, 189, 193, 197, 201, 205, 209, 213, 217, 221, 225, 229, |
| 234, 239, 245, 249, 254, 259, 264, 269, 274, 279, 284, |
| } |

If HasY2 is equal to 1, the Walsh-Hadamard inputs are derived as follows:

```
dqf = Max( 8, (ac_q( q + y2_ac_delta_q ) * 155) / 100 )
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        Walsh[ x ][ y ] = dqf * TransCoeff[ 24 ][ x ][ y ]
Walsh[ 0 ][ 0 ] = dc_q( q + y2_dc_delta_q ) * 2 * TransCoeff[ 24 ][ 0 ][ 0 ]
```

The luma blocks are dequantized as follows:

```
for ( block = 0; block < 16; block++ ) {
    for ( y = 0; y < 4; y++ )
        for ( x = 0; x < 4; x++ )
            Dequant[ block ][ x ][ y ] = ac_q( y_ac_qi ) * TransCoeff[ block ][ x ][ y ]
    Dequant[ block ][ 0 ][ 0 ] = dc_q( q + y_dc_delta_q ) * TransCoeff[ block ][ 0 ][ 0 ]
}
```

The chroma blocks are dequantized as follows:

```
dqf = ac_q( q + uv_ac_delta_q )
for ( block = 16; block < 24; block++ ) {
    for ( y = 0; y < 4; y++ )
        for ( x = 0; x < 4; x++ )
            Dequant[ block ][ x ][ y ] = dqf * TransCoeff[ block ][ x ][ y ]
    Dequant[ block ][ 0 ][ 0 ] = Min( 132, dc_q( q + uv_dc_delta_q) ) * TransCoeff[ block ][ 0 ][ 0 ]
}
```

A bitstream shall not result in a value being stored in Walsh or Dequant that is less than −32768 or greater than 32767.

#### 8.4.4.3 Implementation of the WHT inversion

This process updates the luma DC values in Dequant[ block ][ 0 ][ 0 ] for block = 0..15.

The inputs to this process are the values in Walsh[ x ][ y ] for x = 0..3, y = 0..3.

The array input is specified as follows:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        input[ x + y * 4 ] = Walsh[ x ][ y ]
```

The inverse transform used to map the WHT coefficients to the spatial domain is specified in Table 15.

**Table 15 — Inverse 4x4 Walsh Hadamard transform**

| |
|---|
| vcb_short_inv_walsh4x4( input[ 16 ], output[ 16 ] ) { |
|   for ( i = 0; i < 4; i++ ) { |
|     a1 = input[ i + 0 ] + input[ i + 12 ] |
|     b1 = input[ i + 4 ] + input[ i + 8 ] |
|     c1 = input[ i + 4 ] − input[ i + 8 ] |
|     d1 = input[ i + 0 ] − input[ i + 12 ] |
|     output[ i + 0 ] = a1 + b1 |
|     output[ i + 4 ] = c1 + d1 |
|     output[ i + 8 ] = a1 − b1 |
|     output[ i + 12 ] = d1 − c1 |
|   } |
|   idx = 0 |
|   for ( i = 0; i < 4; i++ ) { |
|     a1 = output[ idx + 0 ] + output[ idx + 3 ] |
|     b1 = output[ idx + 1 ] + output[ idx + 2 ] |
|     c1 = output[ idx + 1 ] − output[ idx + 2 ] |
|     d1 = output[ idx + 0 ] − output[ idx + 3 ] |
|     a2 = a1 + b1 |
|     b2 = c1 + d1 |
|     c2 = a1 − b1 |
|     d2 = d1 − c1 |

**Table 15** *(continued)*

| |
|---|
| output[ idx + 0 ] = (a2 + 3) >> 3 |
| output[ idx + 1 ] = (b2 + 3) >> 3 |
| output[ idx + 2 ] = (c2 + 3) >> 3 |
| output[ idx + 3 ] = (d2 + 3) >> 3 |
| idx += 4 |
| } |
| } |

A bitstream shall not result in values being stored in the input or output arrays that are less than −32768 or greater than 32767.

The DC dequantized values are updated as follows:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        Dequant[ x + y * 4 ][ 0 ][ 0 ] = output[ x + y * 4 ]
```

### 8.4.4.4 DCT macroblock inversion process

#### 8.4.4.4.1 General

This process generates the values in PredictionResidual.

The inputs to this process are the dequantized coefficients in Dequant.

The following applies for bx = 0..3 and by = 0..3:

— An array input is specified as follows:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        input[ x + y * 4 ] = Dequant[ bx + by * 4 ][ x ][ y ]
```

— The DCT block inversion process specified in 8.4.4.4.2 is invoked.

— The prediction residual values are updated as follows:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        PredictionResidual[ 0 ][ bx * 4 + x ][ by * 4 + y ] = output[ x + y * 4 ]
```

The following applies for colourPlane = 1..2, bx = 0..1, and by = 0..1:

— An array input is specified as follows:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        input[ x + y * 4 ] = Dequant[ 16 + (colourPlane − 1) * 4 + bx + by * 2 ][ x ][ y ]
```

— The DCT block inversion process specified in 8.4.4.4.2 is invoked.

— The prediction residual values are updated as follows:

```
for ( y = 0; y < 4; y++ )
    for ( x = 0; x < 4; x++ )
        PredictionResidual[ colourPlane ][ bx * 4 + x ][ by * 4 + y] = output[ x + y * 4 ]
```

#### 8.4.4.4.2 DCT block inversion process

The input to this process is an array input with 16 elements.

The output from this process is an array output with 16 elements.

The DCT inversions are computed using a 2-D inverse transform, which can be implemented as two passes of a 1-D inverse transform.

The inverse 4x4 DCT is specified in Table 16.

**Table 16 — Inverse 4x4 discrete cosine tranform**

| |
|---|
| idct4x4( input[ 16 ], output[ 16 ] ) { |
| cospi8sqrt2minus1 = 20091 |
| sinpi8sqrt2 = 35468 |
| idx = 0 |
| for ( i = 0; i < 4; i++ ) { |
| a1 = input[ idx + 0 ] + input[ idx + 8 ] |
| b1 = input[ idx + 0 ] − input[ idx + 8 ] |
| temp1 = (input[ idx + 4 ] * sinpi8sqrt2) >> 16 |
| temp2 = input[ idx + 12 ] + ((input[ idx + 12 ] * cospi8sqrt2minus1) >> 16) |
| c1 = temp1 − temp2 |
| temp1 = input[ idx + 4 ] + ((input[ idx + 4 ] * cospi8sqrt2minus1) >> 16) |
| temp2 = (input[ idx + 12 ] * sinpi8sqrt2) >> 16 |
| d1 = temp1 + temp2 |
| output[ idx + 0 ] = a1 + d1 |
| output[ idx + 12 ] = a1 − d1 |
| output[ idx + 4 ] = b1 + c1 |
| output[ idx + 8 ] = b1 − c1 |
| idx++ |
| } |
| idx = 0 |
| for ( i = 0; i < 4; i++ ) { |
| a1 = output[ idx + 0 ] + output[ idx + 2 ] |
| b1 = output[ idx + 0 ] − output[ idx + 2 ] |
| temp1 = (output[ idx + 1 ] * sinpi8sqrt2) >> 16 |
| temp2 = output[ idx + 3 ] + ((output[ idx + 3 ] * cospi8sqrt2minus1) >> 16) |
| c1 = temp1 − temp2 |
| temp1 = output[ idx + 1 ] + ((output[ idx + 1 ] * cospi8sqrt2minus1) >> 16) |
| temp2 = (output[ idx + 3 ] * sinpi8sqrt2) >> 16 |
| d1 = temp1 + temp2 |
| output[ idx + 0 ] = (a1 + d1 + 4) >> 3 |
| output[ idx + 3 ] = (a1 − d1 + 4) >> 3 |
| output[ idx + 1 ] = (b1 + c1 + 4) >> 3 |

**Table 16** *(continued)*

| |
|---|
|         output[ idx + 2 ] = (b1 − c1 + 4) >> 3 |
|         idx += 4 |
|     } |
| } |

A bitstream shall not result in values being stored in the input or output arrays that are less than −32768 or greater than 32767.

## 8.4.5 Motion vector prediction process

### 8.4.5.1 General

This process is invoked when decoding mv_mode for a macroblock.

Inputs to this process are:

— the variables StartRow and StartCol specifying the location of the macroblock;

— previously decoded motion vectors.

The outputs are an array cnt and a list of motion vectors NearMvs.

The process is specified as:

```
numMvs = 0
for ( i = 0; i < 4; i++ ) {
  cnt[ i ] = 0
  NearMvs[ i ] = 0
}
if ( MbRefFrame[ StartRow − 1 ][ StartCol ] != 0 ) {
  if ( MbMvs[ StartRow − 1 ][ StartCol ] != 0 ) {
    numMvs++
    NearMvs[ numMvs ] = Bias( MbMvs[ StartRow − 1 ][ StartCol ],
                              MbRefFrame[ StartRow − 1 ][ StartCol ] )
  }
  cnt[ numMvs ] += 2
}
if ( MbRefFrame[ StartRow ][ StartCol − 1] != 0 ) {
  thisMv = Bias( MbMvs[ StartRow ][ StartCol − 1], MbRefFrame[ StartRow ][ StartCol − 1] )
  if ( thisMv != 0 ) {
    if ( thisMv != NearMvs[ numMvs ] ) {
      numMvs++
      NearMvs[ numMvs ] = thisMv
    }
    cnt[ numMvs ] += 2
    } else
        cnt[ 0 ] += 2
}
if ( MbRefFrame[ StartRow − 1 ][ StartCol − 1] != 0 ) {
    thisMv = Bias( MbMvs[ StartRow − 1 ][ StartCol − 1], MbRefFrame[ StartRow − 1 ][ StartCol − 1] )
    if ( thisMv != 0 ) {
        if ( thisMv != NearMvs[ numMvs ] ) {
            numMvs++
            NearMvs[ numMvs ] = thisMv
        }
        cnt[ numMvs ] += 1
    } else
        cnt[ 0 ] += 1
}
if ( cnt[ 3 ] && NearMvs[ 3 ] == NearMvs[ 1 ] )
    cnt[ 1 ] += 1
cnt[ 3 ] = 2 * MbSplit[ StartRow − 1][ StartCol ] + 2 * MbSplit[ StartRow ][ StartCol − 1]
            + MbSplit[ StartRow − 1][ StartCol − 1 ]
```

If cnt[ 2 ] is greater than cnt[ 1 ], then the contents of the array NearMvs at positions 1 and 2 are swapped and the contents of the array cnt at positions 1 and 2 are swapped.

If cnt[ 1 ] is greater than or equal to cnt[ 0 ], then NearMvs[ 0 ] is set equal to NearMvs[ 1 ].

NOTE The process of decoding the motion vectors generates a sorted list of distinct motion vectors adjacent to the search site. The label best_mv is used to refer to the vector (NearMvs[ 0 ]) with the highest score. The label mv_nearest is used to refer to the non-zero vector (NearMvs[ 1 ]) with the highest score. The label mv_near is used to refer to the non-zero vector (NearMvs[ 2 ]) with the next highest score.

The motion vectors in NearMvs are clamped as follows:

```
for ( i = 0; i < 3; i++ ) {
    NearMvs[ i ][ 1 ] = Clip3( −( (StartCol + 1) << 6), (MbCols − col) << 6, NearMvs[ i ][ 1 ] )
    NearMvs[ i ][ 0 ] = Clip3( −( (StartRow + 1) << 6), (MbRows − row) << 6, NearMvs[ i ][ 0 ] )
}
```

The function Bias( neighborMV, neighborRef ) returns a modified motion vector outMV as follows:

```
outMv = neighborMV
if ( ref_frame_sign_bias[ neighborRef ] != ref_frame_sign_bias[ MbRefFrame[ StartRow ][ StartCol ] ] ) {
    outMv[ 0 ] = −outMv[ 0 ]
    outMv[ 1 ] = −outMv[ 1 ]
}
```

### 8.4.5.2 Sub-block motion vector prediction process

This process is invoked when decoding sub_mv_mode.

Inputs to this process are:

— the variable i indicating which partition is being processed;

— the variables StartRow and StartCol specifying the current macroblock.

Outputs of this process are LeftMv and AboveMv.

The variable b is derived as follows:

— If mv_split_mode is equal to MV_TOP_BOTTOM, b is set equal to 8 * i.

— Otherwise, if mv_split_mode is equal to MV_LEFT_RIGHT, b is set equal to 2 * i.

— Otherwise, if mv_split_mode is equal to MV_QUARTERS, the following applies:

  — If i is equal to 0, b is set equal to 0.

  — Otherwise, if i is equal to 1, b is set equal to 2.

  — Otherwise, if i is equal to 2, b is set equal to 8.

  — Otherwise (i is equal to 3), b is set equal to 10.

  — Otherwise (mv_split_mode is equal to MV_16), b is set equal to i.

The variable LeftMv is derived as follows:

— If ( b & 3) is equal to 0, the following applies:

  — If StartCol is equal to 0 or MbRefFrame[ StartRow ][ StartCol − 1] is equal to 0, LeftMv is set to 0.

  — Otherwise, LeftMv is set equal to the contents of SubMvs[ StartRow ][ StartCol − 1 ][ b + 3 ].

— Otherwise, leftMv is set equal to SubMvs[ StartRow ][ StartCol ][ b − 1 ].

The variable AboveMv is derived as follows:

— If b is less than 4, the following applies:

  — If StartRow is equal to 0 or the MbRefFrame[ StartRow − 1 ][ StartCol ] is equal to 0, AboveMv is set to 0.

  — Otherwise, AboveMv is set equal to the contents of SubMvs[ StartRow − 1 ][ StartCol ][ b + 12 ].

— Otherwise, AboveMv is set equal to SubMvs[ StartRow ][ StartCol ][ b − 4 ].

## 8.4.6 Motion vector reconstruction process

### 8.4.6.1 General

Inputs to this process are:

— the predicted motion vectors NearMvs,

— the motion vector differences Mvdy, Mvdx, SubMvdy, SubMvdy.

Outputs of this process are the reconstructed motion vectors Mv and SubMvs.

If mv_mode is equal to MV_SPLIT, then this process does nothing.

Otherwise, the reconstructed motion vector depends on mv_mode as follows.

— If mv_mode is equal to MV_NEAREST, Mv is set equal to NearMvs[ 1 ].

— Otherwise, if mv_mode is equal to MV_NEAR, Mv is set equal to NearMvs[ 2 ].

— Otherwise, if mv_mode is equal to MV_ZERO, Mv[ 0 ] is set equal to 0 and Mv[ 1 ] is set equal to 0.

— Otherwise, if mv_mode is equal to MV_NEW, Mv[ 0 ] is set equal to Mvdy + NearMvs[ 0 ][ 0 ] and Mv[ 1 ] is set equal to Mvdx + NearMvs[ 0 ][ 1 ].

For x = 0..3 and y = 0..3, the value of SubMvs[ StartRow ][ StartCol ][ x + y * 4 ] is set equal to Mv.

### 8.4.6.2 Sub-block motion vector reconstruction process

This process updates the contents of SubMvs for the 4x4 blocks within the macroblock partition.

Inputs to this process are:

— the variable i indicating which partition is being processed,

— the predicted motion vectors LeftMv and AboveMv.

The variables b, w, h are derived as follows:

— If mv_split_mode is equal to MV_TOP_BOTTOM, b is set equal to 8 * i, w is set equal to 4, h is set equal to 2.

— Otherwise, if mv_split_mode is equal to MV_LEFT_RIGHT, b is set equal to 2 * i, w is set equal to 2, h is set equal to 4.

— Otherwise, if mv_split_mode is equal to MV_QUARTERS, w is set equal to 2, h is set equal to 2, and the following applies:

  — If i is equal to 0, b is set equal to 0.

  — Otherwise, if i is equal to 1, b is set equal to 2.

  — Otherwise, if i is equal to 2, b is set equal to 8.

  — Otherwise (i is equal to 3), b is set equal to 10.

— Otherwise (mv_split_mode is equal to MV_16), b is set equal to i, w is set equal to 1, h is set equal to 1.

A variable predMv is derived as follows:

— If sub_mv_mode is equal to LEFT4X4, predMv is set equal to LeftMv.

— Otherwise, if sub_mv_mode is equal to ABOVE4X4, predMv is set equal to AboveMv.

— Otherwise, if sub_mv_mode is equal to ZERO4X4, predMv is set equal to 0.

— Otherwise (sub_mv_mode is equal to NEW4X4), predMv[ 0 ] is set equal to NearMvs[ 0 ][ 0 ] + SubMvdy and predMv[ 1 ] is set equal to NearMvs[ 0 ][ 1 ] + SubMvdx.

For x = 0..w−1 and y = 0..h−1, the value of SubMvs[ StartRow ][ StartCol ][ b + x + y * 4 ] is set equal to predMv.

## 8.5 Loop filter process

### 8.5.1 General

The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi-independence of the coding of macroblocks and their constituent sub-blocks.

The loop filter is integral to decoding, in that the results of loop filtering are used in the prediction of subsequent frames.

The input to this process is the array CurrPic of reconstructed samples.

The output from this process is a modified array CurrPic containing deblocked samples.

The loop filter is applied for all macroblocks as follows:

```
for ( row = 0; row < MbRows; row++ )
    for ( col = 0; col < MbCols; col++ ) {
        The macroblock data syntax elements are set to the values extracted for the macroblock
        with matching row and col.
        If filter_type is equal to 0, the normal macroblock deblocking process specified in 8.5.2 is
        invoked with the variables row and col as inputs.
        Otherwise (if filter_type is equal to 1), the simple macroblock deblocking process specified
        in 8.5.2 is invoked with the variables row and col as inputs.
    }
```

The loop filter acts on the edges between adjacent macroblocks and on the edges between adjacent sub-blocks of a macroblock. All such edges are horizontal or vertical. For each sample position on an edge, a small number (two or three) of samples adjacent to either side of the position are examined and possibly modified. The displacements of these samples are at a right angle to the edge orientation; that is, for a horizontal edge, the samples immediately above and below the edge position are treated and for a vertical edge, the samples immediately to the left and right of the edge are treated.

This collection of samples associated to an edge position is referred to as a segment (in the context of loop filtering); the length of a segment is 2, 4, 6 or 8. Excepting that the normal filter uses a slightly different process for and either filter may apply different control parameters to, the edges between macroblocks and those between sub-blocks, the treatment of edges is quite uniform. All segments straddling an edge are treated identically; there is no distinction between the treatment of horizontal and vertical edges, whether between macroblocks or between sub-blocks.

As a consequence, adjacent sub-block edges within a macroblock may be concatenated and processed in their entirety. There is a single 8-sample-long vertical edge horizontally centered in each of the Cb and Cr blocks (the concatenation of upper and lower 4-sample edges between chroma sub-blocks) and three 16-sample-long vertical edges at horizontal positions one quarter, one half and three quarters of the width of the luma macroblock, each representing the concatenation of four 4-sample sub-edges between pairs of Y sub-blocks.

The macroblocks comprising the frame are processed in raster-scan order. Each macroblock is associated with the inter-macroblock edges immediately above and to the left of it (but not the edges below and to the right of it), as well as the edges between its sub-blocks.

Because many samples belong to segments straddling two or more edges and so will be filtered more than once, the order in which edges are processed given above needs to be respected by any implementation. Within a single edge, however, the segments straddling that edge are disjoint and the order in which these segments are processed is immaterial.

NOTE  The loop filter applies after all the macroblocks have been reconstructed (i.e. had their prediction summed with their prediction residual); correct decoding is predicated on the fact that already-constructed portions of the current frame referenced via intra prediction are not yet filtered.

### 8.5.2 Normal macroblock deblocking process

The inputs to this process are variables StartRow and StartCol specifying which macroblock is to be deblocked.

The filter level process specified in 8.5.3 is invoked with the variables StartRow and StartCol as inputs and the output assigned to the variable filterLevel.

If filterLevel is equal to 0, then this process immediately returns and no deblocking is applied to the current macroblock.

The variable FilterInternal and the threshold variables are calculated by the loop filter threshold process as specified in 8.5.11 is invoked with the variable filterLevel as input.

If StartCol is not equal to 0, the lefthand luma macroblock edge is deblocked by invoking the macroblock edge deblocking process specified in 8.5.5 with colourPlane set equal to 0, dx set equal to 1, dy set equal to 0, x set equal to StartCol * 16, y set equal to StartRow * 16 + i with i = 0..15.

If StartCol is not equal to 0, the lefthand chroma macroblock edge is deblocked by invoking the macroblock edge deblocking process specified in 8.5.5 with dx set equal to 1, dy set equal to 0, x set equal to StartCol * 8, y set equal to StartRow * 8 + i with i = 0..7 and colourPlane = 1..2.

If FilterInternal is equal to 1, the internal luma vertical edges are deblocked by invoking the sub-block edge deblocking process specified in 8.5.6 with colourPlane set equal to 0, dx set equal to 1, dy set equal to 0, x set equal to StartCol * 16 + b * 4, y set equal to StartRow * 16 + i with i = 0..15 and b = 1..3.

If FilterInternal is equal to 1, the internal chroma vertical edges are deblocked by invoking the sub-block edge deblocking process specified in 8.5.6 with dx set equal to 1, dy set equal to 0, x set equal to StartCol * 8 + 4, y set equal to StartRow * 8 + i with i = 0..7 and colourPlane = 1..2.

If StartRow is not equal to 0, the above luma macroblock edge is deblocked by invoking the macroblock edge deblocking process specified in 8.5.5 with colourPlane set equal to 0, dx set equal to 0, dy set equal to 1, x set equal to StartCol * 16 + i, y set equal to StartRow * 16 with i = 0..15.

If StartRow is not equal to 0, the above chroma macroblock edge is deblocked by invoking the macroblock edge deblocking process specified in 8.5.5 with dx set equal to 0, dy set equal to 1, x set equal to StartCol * 8 + i, y set equal to StartRow * 8 with i = 0..7 and colourPlane = 1..2.

If FilterInternal is equal to 1, the internal luma horizontal edges are deblocked by invoking the sub-block edge deblocking process specified in 8.5.6 with colourPlane set equal to 0, dx set equal to 0, dy set equal to 1, x set equal to StartCol * 16 + i, y set equal to StartRow * 16 + b * 4 with i = 0..15 and b = 1..3.

If FilterInternal is equal to 1, the internal chroma horizontal edges are deblocked by invoking the sub-block edge deblocking process specified in 8.5.6 with dx set equal to 0, dy set equal to 1, x set equal to StartCol * 8 + i, y set equal to StartRow * 8 + 4 with i = 0..7 and colourPlane = 1..2.

### 8.5.3 Filter level process

The inputs to this process are:

— the segment_id for the current macroblock,

— variables StartRow and StartCol specifying the current macroblock.

The output of this process is the variable filterLevel.

The variable filterLevel is specified by the following ordered steps.

a) filterLevel is set equal to loop_filter_level.

b) If segmentation_enabled is equal to 1, filterLevel is adjusted based on the segment_id as follows.

   1) If segment_feature_mode is equal to 0, filterLevel is incremented by LfLevel[ segment_id ].

   2) Otherwise, filterLevel is set equal to LfLevel[ segment_id ].

c) filterLevel is set equal to Clip3( 0, 63, filterLevel ).

d) If loop_filter_adj_enable is equal to 1, filterLevel is incremented by RefDelta[ MbRefFrame[ StartRow ] [ StartCol ] ].

e) If loop_filter_adj_enable is equal to 1 and is_inter_mb is equal to 0, the following applies.

   1) If intra_y_mode is equal to B_PRED, filterLevel is incremented by ModeDelta[ 0 ].

f) If loop_filter_adj_enable is equal to 1 and is_inter_mb is equal to 1, the following applies.

   1) If mv_mode is equal to MV_ZERO, filterLevel is incremented by ModeDelta[ 1 ].

   2) Otherwise, if mv_mode is equal to MV_SPLIT, filterLevel is incremented by ModeDelta[ 3 ].

   3) Otherwise, filterLevel is incremented by ModeDelta[ 2 ].

g) filterLevel is set equal to Clip3( 0, 63, filterLevel ).

### 8.5.4 Simple macroblock deblocking process

The inputs to this process are variables StartRow and StartCol specifying which macroblock is to be deblocked.

The filter level process specified in 8.5.3 is invoked with the variables StartRow and StartCol as inputs and the output assigned to the variable filterLevel.

If filterLevel is equal to 0, then this process immediately returns and no deblocking is applied to the current macroblock.

The variable FilterInternal and the threshold variables are calculated by the loop filter threshold process as specified in 8.5.11 invoked with the variable filterLevel as input.

The variable MbLimit is set equal to (EdgeLimit + 2) * 2 + InteriorLimit.

The variable BLimit is set equal to EdgeLimit * 2 + InteriorLimit.

If StartCol is not equal to 0, the lefthand luma macroblock edge is deblocked by invoking the simple edge deblocking process specified in 8.5.7 with limit set equal to MbLimit, dx set equal to 1, dy set equal to 0, x set equal to StartCol * 16, y set equal to StartRow * 16 + i with i = 0..15.

If FilterInternal is equal to 1, the internal luma vertical edges are deblocked by invoking the simple edge deblocking process specified in 8.5.7 with limit set equal to BLimit, dx set equal to 1, dy set equal to 0, x set equal to StartCol * 16 + b * 4, y set equal to StartRow * 16 + i with i = 0..7 and b = 1..3.

If StartRow is not equal to 0, the above luma macroblock edge is deblocked by invoking the simple edge deblocking process specified in 8.5.7 with limit set equal to MbLimit, dx set equal to 0, dy set equal to 1, x set equal to StartCol * 16 + i, y set equal to StartRow * 16 with i = 0..15.

If FilterInternal is equal to 1, the internal luma horizontal edges are deblocked by invoking the simple edge deblocking process specified in 8.5.7 with limit set equal to BLimit, dx set equal to 0, dy set equal to 1, x set equal to StartCol * 16 + i, y set equal to StartRow * 16 + b * 4 with i = 0..7 and b = 1..3.

### 8.5.5 Macroblock edge deblocking process

The inputs to this process are:

— a variable colourPlane specifying the colour component of the edge;

— variables dx and dy specifying the direction of the edge;

— variables x and y specifying the location of the edge.

The threshold process as specified in 8.5.8 is invoked with inputs colourPlane, dx, dy, x, y and limit set equal to EdgeLimit + 2 and the output is assigned to the variables belowThreshold and hevEdge.

The edge is deblocked as follows.

— If belowThreshold is equal to 1 and hevEdge is equal to 1, the common filtering process as specified in 8.5.9 is invoked with inputs colourPlane, dx, dy, x, y, useOuterTaps set equal to 1, and adjustOuterTaps set equal to 0.

— Otherwise, if belowThreshold is equal to 1 (and hevEdge is equal to 0), the macroblock edge filtering process as specified in 8.5.10 is invoked with inputs colourPlane, dx, dy, x, y.

— Otherwise (belowThreshold is equal to 0), no deblocking is performed on the edge.

### 8.5.6 Sub-block edge deblocking process

The inputs to this process are:

— a variable colourPlane specifying the colour component of the edge;

— variables dx and dy specifying the direction of the edge;

— variables x and y specifying the location of the edge.

The threshold process as specified in 8.5.8 is invoked with inputs colourPlane, dx, dy, x, y and limit set equal to EdgeLimit and the output is assigned to the variables belowThreshold and hevEdge.

The edge is deblocked as follows.

— If belowThreshold is equal to 1, the common filtering process as specified in 8.5.9 is invoked with inputs colourPlane, x, y, dx, dy, useOuterTaps set equal to hevEdge and adjustOuterTaps set equal to !hevEdge.

— Otherwise (belowThreshold is equal to 0), no deblocking is performed on the edge.

### 8.5.7 Simple edge deblocking process

The inputs to this process are:

— variables dx and dy specifying the direction of the edge;

— variables x and y specifying the location of the edge;

— a variable limit specifying the level at which deblocking is applied.

The variables p[ i ] and q[ i ] are specified for i = 0..1 as follows:

p[ i ] = CurrPic[ colourPlane ][ x − dx − dx * i ][ y − dy − dy * i ]

q[ i ] = CurrPic[ colourPlane ][ x + dx * i ][ y + dy * i ]

The variable belowThreshold is set by the following ordered steps.

a) belowThreshold is set equal to 1.

b) If ( Abs( p[ 0 ] − q[ 0 ] ) * 2 + (Abs( p[ 1 ] − q[ 1 ] ) >> 1) ) is greater than limit, belowThreshold is set equal to 0.

The edge is deblocked as follows.

— If belowThreshold is equal to 1, the common filtering process as specified in 8.5.9 is invoked with colourPlane set equal to 0, useOuterTaps set equal to 1, adjustOuterTaps set equal to 0 and the variables dx, dy, x, y as inputs.

— Otherwise (belowThreshold is equal to 0), no deblocking is performed on the edge.

### 8.5.8 Threshold process

The inputs to this process are:

— a variable colourPlane specifying the colour component of the edge;

— variables dx and dy specifying the direction of the edge;

— variables x and y specifying the location of the edge;

— a variable limit specifying the maximum size before deblocking is disabled.

The output is a variable belowThreshold indicating whether the edge is deblocked and a variable hevEdge indicating whether the edge has high edge variance.

The variables p[ i ] and q[ i ] are specified for i = 0..3 as follows:

p[ i ] = CurrPic[ colourPlane ][ x − dx − dx * i ][ y − dy − dy * i ]

q[ i ] = CurrPic[ colourPlane ][ x + dx * i ][ y + dy * i ]

The variable simpleLimit is set equal to 2 * limit + InteriorLimit.

The output variable belowThreshold is set by the following ordered steps.

a) belowThreshold is set equal to 1.

b) If ( Abs( p[ 0 ] − q[ 0 ] ) * 2 + ( Abs( p[ 1 ] − q[ 1 ] ) >> 1) ) is greater than simpleLimit, belowThreshold is set equal to 0.

c) If filter_type is equal to 0, and Abs( p[ i + 1 ] − p[ i ] ) is greater than InteriorLimit for any of i = 0..2, belowThreshold is set equal to 0.

d) If filter_type is equal to 0, and Abs( q[ i + 1 ] − q[ i ] ) is greater than InteriorLimit for any of i = 0..2, belowThreshold is set equal to 0.

The output variable hevEdge is set by the following ordered steps.

a) hevEdge is set equal to 0.

b) If Abs( p[ 1 ] − p[ 0 ] ) is greater than HevThreshold, hevEdge is set equal to 1.

c) If Abs( q[ 1 ] − q[ 0 ] ) is greater than HevThreshold, hevEdge is set equal to 1.

### 8.5.9 Common filtering process

The inputs to this process are:

— a variable colourPlane specifying the colour component of the edge;

— variables dx and dy specifying the direction of the edge;

— variables x and y specifying the location of the edge;

— a variable useOuterTaps specifying whether the outer taps are examined;

— a variable adjustOuterTaps specifying whether the outer taps are modified.

The variables p[ i ] and q[ i ] are specified for i = 0..1 as follows:

```
p[ i ] = CurrPic[ colourPlane ][ x − dx − dx * i ][ y − dy − dy * i ]
q[ i ] = CurrPic[ colourPlane ][ x + dx * i ][ y + dy * i ]
```

The variables a, f1, f2, f3 are specified as:

```
a = 3 * (q[ 0 ] − p[ 0 ] )
if ( useOuterTaps )
    a += Clip3( −128, 127, p[ 1 ] − q[ 1 ] )
a = Clip3( −128, 127, a )
f1 = ((a + 4 > 127) ? 127 : a + 4) >> 3
f2 = ((a + 3 > 127) ? 127 : a + 3) >> 3
f3 = (f1 + 1) >> 1
```

The edge samples are modified as follows:

```
CurrPic[ colourPlane ][ x − dx ][ y − dy ] = Clip1( p[ 0 ] + f2 )
CurrPic[ colourPlane ][ x ][ y ] = Clip1( q[ 0 ] − f1 )
```

If adjustOuterTaps is equal to 1, then the additional edge samples are modified as follows:

```
CurrPic[ colourPlane ][ x − 2 * dx ][ y − 2 * dy ] = Clip1( p[ 1 ] + f3 )
CurrPic[ colourPlane ][ x + dx ][ y + dy ] = Clip1( q[ 1 ] − f3 )
```

### 8.5.10 Macroblock edge filtering process

The inputs to this process are:

— a variable colourPlane specifying the colour component of the edge;

— variables dx and dy specifying the direction of the edge;

— variables x and y specifying the location of the edge.

The variables p[ i ] and q[ i ] are specified for i = 0..2 as follows:

```
p[ i ] = CurrPic[ colourPlane ][ x − dx − dx * i ][ y − dy − dy * i ]
q[ i ] = CurrPic[ colourPlane ][ x + dx * i ][ y + dy * i ]
```

The variables a, w, f[ 0 ], f[ 1 ], f[ 2 ] are specified as:

```
a = Clip3( −128, 127, p[ 1 ] − q[ 1 ] )
w = Clip3( −128, 127, a + 3 * (q[ 0 ] − p[ 0 ]) )
f[ 0 ] = (27 * w + 63) >> 7
f[ 1 ] = (18 *w + 63) >> 7
f[ 2 ] = (9 * w + 63) >> 7
```

The edge samples are modified as follows:

```
for ( i = 0; i < 3; i++ ) {
    CurrPic[ colourPlane ][ x − dx − dx * i ][ y − dy − dy * i ] = Clip1( p[ i ] + f[ i ] )
    CurrPic[ colourPlane ][ x + dx * i ][ y + dy * i ] = Clip1( q[ i ] − f[ i ] )
}
```

### 8.5.11 Loop filter threshold process

The input to this process is a variable filterLevel.

The output of this process is to set the threshold variables EdgeLimit, InteriorLimit and HevThreshold and to set the FilterInternal variable that controls whether sub-block edges are filtered.

The variable EdgeLimit is set equal to filterLevel.

The variable InteriorLimit is set depending on sharpness_level as follows.

— If sharpness_level is zero, InteriorLimit is set equal to Max( 1,filterLevel ).

— Otherwise, if sharpness_level is greater than 4, the following applies:

InteriorLimit = Clip3( 1, 9 − sharpness_level, filterLevel >> 2 ).

— Otherwise, the following applies:

InteriorLimit = Clip3( 1, 9 − sharpness_level, filterLevel >> 1 ).

The variable HevThreshold is specified by the following ordered steps.

a) HevThreshold is set equal to 0.

b) If filterLevel is greater than or equal to 15, HevThreshold is incremented by 1.

c) If filterLevel is greater than or equal to 40, HevThreshold is incremented by 1.

d) If filterLevel is greater than or equal to 20 and KeyFrame is equal to 0, HevThreshold is incremented by 1.

The variable FilterInternal is set as follows.

— If HasNonZero is equal to 1 for the current macroblock, FilterInternal is set equal to 1.

— Otherwise, if is_inter_mb is equal to 0 and intra_y_mode is equal to B_PRED, FilterInternal is set to 1.

— Otherwise, if is_inter_mb is equal to 1 and imv_mode is equal to MV_SPLIT, FilterInternal is set to 1.

— Otherwise, FilterInternal is set equal to 0.

## 8.6 Output process

This process is invoked after the loop filtering has taken place and if show_frame is equal to 1.

Inputs to this process are the samples in the current frame CurrPic[ colourPlane ][ x ][ y ].

The output from this process is a frame to be displayed.

The decoder should provide the values of horizontal_scale and vertical_scale to allow appropriate scaling to take place. The methods for upscaling and displaying a frame are outside the scope of this document.

The decoder provides an output frame consisting of a Y component, a Cb component and a Cr component.

The Y component is FrameWidth samples wide by FrameHeight samples high and the sample at location x samples from the left side of the component and y samples from the top of the component is given by CurrPic[ 0 ][ x ][ y ] with x = 0..FrameWidth − 1 and y = 0..FrameHeight − 1.

The Cb component is (FrameWidth + 1) >> 1 samples wide by (FrameHeight + 1) >> 1 samples high and the sample at location x samples from the left side of the component and y samples from the top of the component is given by CurrPic[ 1 ][ x ][ y ] with x = 0..((FrameWidth + 1) >> 1) − 1 and y = 0..((FrameHeight + 1) >> 1) − 1.

The Cr component is (FrameWidth + 1) >> 1 samples wide by (FrameHeight + 1) >> 1 samples high and the sample at location x samples from the left side of the component and y samples from the top of the component is given by CurrPic[ 2 ][ x ][ y ] with x = 0..((FrameWidth + 1) >> 1) − 1 and y = 0..((FrameHeight + 1) >> 1) − 1.

## 8.7 Reference frame update process

### 8.7.1 General

This process is invoked as the final step in decoding a frame.

Inputs to this process are the samples in the current frame CurrPic[ colourPlane ][ x ][ y ].

The output from this process is an updated set of reference frames.

The following ordered steps apply.

a) Depending on the value of copy_buffer_to_ref2, the following applies.

   1) If copy_buffer_to_ref2 is equal to 1, the copy frame process as specified in 8.7.2 is invoked with the source as Ref0 and the destination as Ref2.

   2) Otherwise, if copy_buffer_to_ref2 is equal to 2, the copy frame process as specified in 8.7.2 is invoked with the source as Ref1 and the destination as Ref2.

   3) Otherwise, no change is made to Ref2.

b) Depending on the value of copy_buffer_to_ref1, the following applies.

   1) If copy_buffer_to_ref1 is equal to 1, the copy frame process as specified in 8.7.2 is invoked with the source as Ref0 and the destination as Ref1.

   2) Otherwise, if copy_buffer_to_ref2 is equal to 2, the copy frame process as specified in 8.7.2 is invoked with the source as Ref2 and the destination as Ref1.

   3) Otherwise, no change is made to Ref1.

c) If refresh_ref1_frame is equal to 1, the copy frame process as specified in 8.7.2 is invoked with the source as CurrPic and the destination as Ref1.

d) If refresh_ref2_frame is equal to 1, the copy frame process as specified in 8.7.2 is invoked with the source as CurrPic and the destination as Ref2.

   NOTE These steps happen in order, so if both copy_buffer_to_ref2 and copy_buffer_to_ref1 are equal to 2, then firstly, the old Ref1 will be copied to Ref2, followed secondly by copying the updated Ref2 to Ref1. In this case, both Ref2 and Ref1 will end up with copies of the previous Ref1.

e) If refresh_ref0 is equal to 1, the copy frame process as specified in 8.7.2 is invoked with the source as CurrPic and the destination as Ref0.

### 8.7.2 Copy frame process

The inputs to this process are:

— a source array Src;

— a destination array Dst.

The outputs from this process are modifications made to the destination array.

This process copies the contents of the Src array into the Dst array for all macroblocks as follows:

```
for (y = 0; y < 16 * MbRows; y++ )
    for (x = 0; x < 16 * MbCols; x++ )
        Dst[ 0 ][ x ][ y ] = Src[ 0 ][ x ][ y ]
for (colourPlane = 1; colourPlane <= 2; colourPlane++ )
    for (y = 0; y < 8 * MbRows; y++ )
        for (x = 0; x < 8 * MbCols; x++ )
            Dst[ colourPlane ][ x ][ y ] = Src[ colourPlane ][ x ][ y ]
```

# 9 Parsing process

## 9.1 Parsing process for f(n)

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to f(n).

The next n bits are read from the bitstream.

This process is specified as follows:

```
x = 0
for ( i = 0; i < n; i++ )
    x = 2 * x + read_bit( )
```

read_bit( ) reads the next bit from a bitstream and advances the bitstream pointer by 1. If a bitstream is provided as a series of bytes, then the first bit is given by the most significant bit of the first byte.

The value for the syntax element is given by x.

## 9.2 Parsing process for Boolean decoder

### 9.2.1 General

Aside from the uncompressed header and the partition sizes, the entire VCB bitstream is entropy coded. The entropy decoder is referred to as the "Boolean decoder" and the functions init_bool( b, sz ) and read_bool( p ) are used in this document to indicate the entropy decoding operation.

### 9.2.2 Initialization process for Boolean decoder

This process is invoked when the function init_bool( b, sz ) is called from the syntax structure.

A bitstream shall not contain data that results in this process being called with sz < 1.

The variable boolStart is set to the current bit position.

NOTE 1 The bit position will always be byte aligned when init_bool is invoked because the uncompressed header and the data partitions are always a whole number of bytes long.

The variable boolEnd is set to boolStart + 8 * sz.

The variable BoolValue[ b ] is read using the f(8) parsing process.

The variable BoolPosition[ b ] is set to boolStart + 8.

The variable BoolRange[ b ] is set to 255.

The variable BoolMaxBits[ b ] is set to 8 * sz − 8.

The current bit position is set to boolEnd.

NOTE 2 The init_bool function skips over the entire content of the partition. Syntax elements read with f( n ) will be read from after the partition, while Boolean decoded syntax elements will be read from within the partition.

### 9.2.3 Boolean decoder selection process

This process is invoked when the function set_bool( b ) is called from the syntax structure.

The current Boolean Decoder is changed to be b as follows:

CurrBool = b

### 9.2.4 Boolean decoding process

This process is invoked when the function read_bool( p ) is called from the syntax structure.

A variable split is set to 1 + (((BoolRange[ CurrBool ] − 1) * p) >> 8)

The variables BoolRange[ CurrBool ], BoolValue[ CurrBool ] and ReadBit are updated as follows:

— If BoolValue[ CurrBool ] is less than split, then the following applies:

BoolRange[ CurrBool ] = split

ReadBit = 0

— Otherwise, (BoolValue[ CurrBool ] is greater than or equal to split), then the following applies:

BoolRange[ CurrBool ] −= split

BoolValue[ CurrBool ] −= split

ReadBit = 1

While BoolRange[ CurrBool ] is less than 128 the following applies:

— A variable newBit is derived as:

  — If BoolMaxBits[ CurrBool ] > 0, the following ordered steps apply:

    — newBit is set to the contents of a bitstream at position BoolPosition[ CurrBool ].

    — BoolMaxBits[ CurrBool ] −= 1

    — BoolPosition[ CurrBool ] += 1

  — Otherwise, newBit is set equal to 0.

— BoolRange[ CurrBool ] is doubled

— BoolValue[ CurrBool ] is set as follows:

```
BoolValue[ CurrBool ] = (BoolValue[ CurrBool ] << 1) + newBit
```

The return value for the function is given by the variable ReadBit.

### 9.2.5 Parsing process for read_literal

This process is invoked when the function read_literal( n ) is invoked.

This process is specified as follows:

```
x = 0
for ( i = 0; i < n; i++ )
    x = 2 * x + read_bool( 128 )
```

The return value for the function is given by the resulting value of x.

## 9.3 Parsing process for tree encoded syntax elements

### 9.3.1 General

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to T.

The decoding of a syntax element depends on a tree and a list of probabilities.

9.3.2 specifies how a tree is chosen for each syntax element.

9.3.3 specifies how the probabilities are chosen for each syntax element.

9.3.4 specifies how the value of the syntax element is decoded based on the chosen tree and probabilities.

### 9.3.2 Tree selection process

The input to this process is the name of a syntax element.

The output from this process is a tree and a starting node.

The starting node is 0 for all syntax elements except coeff_token and short_mv.

The tree is chosen based on the syntax element as follows:

segment_id: the tree is mbSegmentTree as shown in Table 17.

**Table 17 — segment_id decoding tree**

| |
|---|
| mbSegmentTree [ 2 * (4 − 1) ] = { |
| 2, 4, /* root: "0", "1" subtrees */ |
| −0, −1, /* "00" = 0th value, "01" = 1st value */ |
| −2, −3 /* "10" = 2nd value, "11" = 3rd value */ |
| } |

intra_y_mode: The tree depends on the value of KeyFrame (see Table 18).

— If KeyFrame is equal to 1, the tree is kfYmodeTree.

— Otherwise (KeyFrame is equal to 0), the tree is ymodeTree (see Table 19).

**Table 18 — intra_y_mode decoding tree in key frames**

| | |
|---|---|
| kfYmodeTree[ 8 ] = { | |
| −B_PRED, 2, | /* root: B_PRED = "0", "1" subtree */ |
| 4, 6, | /* "1" subtree has 2 descendant subtrees */ |
| −DC_PRED, −V_PRED, | /* "10" subtree: DC_PRED = "100", V_PRED = "101" */ |
| −H_PRED, −TM_PRED | /* "11" subtree: H_PRED = "110",TM_PRED = "111" */ |
| } | |

**Table 19 — intra_y_mode decoding tree in P frames**

| | |
|---|---|
| ymodeTree[ 8 ] = { | |
| −DC_PRED, 2, | /* root: DC_PRED = "0", "1" subtree */ |
| 4, 6, | /* "1" subtree has 2 descendant subtrees */ |
| −V_PRED, −H_PRED, | /* "10" subtree: V_PRED = "100", H_PRED = "101" */ |
| −TM_PRED, −B_PRED | /* "11" subtree: TM_PRED = "110", B_PRED = "111" */ |
| } | |

intra_b_mode: the tree is bmodeTree as shown in Table 20.

**Table 20 — intra_b_mode decoding tree**

| | |
|---|---|
| bmodeTree[ 18 ] = { | |
| −B_DC_PRED, 2, | /* B_DC_PRED = "0" */ |
| −B_TM_PRED, 4, | /* B_TM_PRED = "10" */ |
| −B_VE_PRED, 6, | /* B_VE_PRED = "110" */ |
| 8, 12, | |
| −B_HE_PRED, 10, | /* B_HE_PRED = "11100" */ |
| −B_RD_PRED, | /* B_RD_PRED = "111010" */ |
| −B_VR_PRED, | /* B_VR_PRED = "111011" */ |
| −B_LD_PRED, 14, | /* B_LD_PRED = "111110" */ |
| −B_VL_PRED, 16, | /* B_VL_PRED = "1111110" */ |
| −B_HD_PRED, −B_HU_PRED | /* HD = "11111110", HU = "11111111" */ |
| } | |

intra_uv_mode: the tree is uvModeTree as shown in Table 21.

**Table 21 — intra_uv_mode decoding tree**

| | |
|---|---|
| uvModeTree[ 6 ] = { | |
| −DC_PRED, 2, | /* root: DC_PRED = "0", "1" subtree */ |
| −V_PRED, 4, | /* "1" subtree: V_PRED = "10", "11" subtree */ |
| −H_PRED, −TM_PRED | /* "11" subtree: H_PRED = "110", TM_PRED = "111" */ |
| } | |

mv_mode: the tree is mvModeTree as shown in Table 22.

**Table 22 — mv_mode decoding tree**

```
mvModeTree[ 8 ] = {
        −MV_ZERO, 2,               /* zero = "0" */
        −MV_NEAREST, 4,            /* nearest = "10" */
        −MV_NEAR, 6,               /* near = "110" */
        −MV_NEW, −MV_SPLIT   /* new = "1110", split = "1111" */
}
```

mv_split_mode: the tree is mvSplitTree as shown in Table 23.

**Table 23 — mv_split_mode decoding tree**

```
mvSplitTree[ 6 ] = {
        −MV_16, 2,                  /* MV_16 = "0" */
        −MV_QUARTERS, 4,            /* MV_QUARTERS = "10" */
        −MV_TOP_BOTTOM,             /* MV_TOP_BOTTOM = "110" */
        −MV_LEFT_RIGHT              /* MV_LEFT_RIGHT= "111" */
}
```

sub_mv_mode: the tree is subMvModeTree as shown in Table 24.

**Table 24 — sub_mv_mode decoding tree**

```
subMvModeTree[ 6 ] = {
        −LEFT4X4, 2,               /* LEFT = "0" */
        −ABOVE4X4, 4,              /* ABOVE = "10" */
        −ZERO4X4, −NEW4X4     /* ZERO = "110", NEW = "111" */
}
```

mv_short: the tree is mvShortTree as shown in Table 25.

**Table 25 — mv_short decoding tree**

```
mvShortTree[ 18 ] = {
        0, 0,                /* Unused */
        0, 0,                /* Unused */
        2 + 4, 8 + 4,          /* "0" subtree, "1" subtree */
        4 + 4, 6 + 4,          /* "00" subtree, "01" subtree */
        −0, −1,                /* 0 = "000", 1 = "001" */
        −2, −3,                /* 2 = "010", 3 = "011" */
        10 + 4, 12 + 4,     /* "10" subtree, "11" subtree */
        −4, −5,                /* 4 = "100", 5 = "101" */
        −6, −7               /* 6 = "110", 7 = "111" */
}
```

The starting node is 4.

NOTE A non-zero starting node is used to allow the tree decode to share the array mv_prob used for decoding the other parts of the MV component.

coeff_token: the tree is tokenTree as shown in Table 26.

**Table 26 — coeff_token decoding tree**

| |
|---|
| tokenTree[ 22 ] = { |
| −dct_eob, 2, /* eob = "0" */ |
| −DCT_0, 4, /* 0 = "10" */ |
| −DCT_1, 6, 8, 12, /* 1 = "110" */ |
| −DCT_2, 10, /* 2 = "11100" */ |
| −DCT_3, −DCT_4, /* 3 = "111010", 4 = "111011" */ |
| 14, 16, |
| −DCT_CAT1, −DCT_CAT2, /* cat1 = "111100", cat2 = "111101" */ |
| 18, 20, |
| −DCT_CAT3, −DCT_CAT4, /* cat3 = "1111100", cat4 = "1111101" */ |
| −DCT_CAT5, −DCT_CAT6 /* cat5 = "1111110", cat6 = "1111111" */ |
| } |

The starting node depends on the value of lastCoeff.

— If lastCoeff is equal to 0, the starting node is 2.

— Otherwise (lastCoeff is not equal to 0), the starting node is 0.

### 9.3.3 Probability selection process

The input to this process is the name of a syntax element.

The output from this process is an array of probabilities.

The probabilities depend on the syntax element as follows:

segment_id: the probabilities are given by the array segment_prob.

intra_y_mode: The probabilities depend on the value of KeyFrame:

— If KeyFrame is equal to 1, the probabilities are given by the array kfYmodeProb (see Table 27).

— Otherwise (KeyFrame is equal to 0), the probabilities are given by the array intra_16x16_prob.

**Table 27 — intra_y_mode decoding probabilities**

| |
|---|
| kfYmodeProb[ 4 ] = { 145, 156, 163, 128 } |

intra_b_mode: the probabilities depend on the value of KeyFrame, the variable ParseBlock, the variable ParseCol, the variable ParseRow and the intra coding context in LeftIntraMode and AboveIntraMode as follows (see Table 28).

— The variable leftMode is specified as:

  — If ( ParseBlock & 3) is not equal to 0, leftMode is set equal to intra_b_mode[ ParseBlock − 1 ].

  — Otherwise, if ParseCol is equal to 0, leftMode is set equal to B_DC_PRED.

  — Otherwise, leftMode is set equal to LeftIntraMode[ block >> 2 ].

— The variable aboveMode is specified as:

  — If ParseBlock is greater than 3, aboveMode is set equal to intra_b_mode[ ParseBlock − 4 ].

  — Otherwise, if ParseRow is equal to 0, aboveMode is set equal to B_DC_PRED.

  — Otherwise, aboveMode is set equal to AboveIntraMode[ ParseCol ][ ParseBlock ].

The probabilities depend on variables aboveMode and leftMode as follows:

— If KeyFrame is equal to 1, the probabilities are given by array kfBmodeProb[ aboveMode ][ leftMode ].

— Otherwise (KeyFrame is equal to 0), the probabilities are given by the array bmodeProb. The contents of bmodeProb are specified in Table 29.

**Table 28 — intra_b_mode decoding probabilities**

| |
|---|
| kfBmodeProb[ 10 ][ 10 ][ 9 ] = { |
| { |
| { 231, 120, 48, 89, 115, 113, 120, 152, 112 }, |
| { 152, 179, 64, 126, 170, 118, 46, 70, 95 }, |
| { 175, 69, 143, 80, 85, 82, 72, 155, 103 }, |
| { 56, 58, 10, 171, 218, 189, 17, 13, 152 }, |
| { 144, 71, 10, 38, 171, 213, 144, 34, 26 }, |
| { 114, 26, 17, 163, 44, 195, 21, 10, 173 }, |
| { 121, 24, 80, 195, 26, 62, 44, 64, 85 }, |
| { 170, 46, 55, 19, 136, 160, 33, 206, 71 }, |
| { 63, 20, 8, 114, 114, 208, 12, 9, 226 }, |
| { 81, 40, 11, 96, 182, 84, 29, 16, 36 } |
| }, |
| { |
| { 134, 183, 89, 137, 98, 101, 106, 165, 148 }, |
| { 72, 187, 100, 130, 157, 111, 32, 75, 80 }, |
| { 66, 102, 167, 99, 74, 62, 40, 234, 128 }, |
| { 41, 53, 9, 178, 241, 141, 26, 8, 107 }, |
| { 104, 79, 12, 27, 217, 255, 87, 17, 7 }, |
| { 74, 43, 26, 146, 73, 166, 49, 23, 157 }, |
| { 65, 38, 105, 160, 51, 52, 31, 115, 128 }, |
| { 87, 68, 71, 44, 114, 51, 15, 186, 23 }, |
| { 47, 41, 14, 110, 182, 183, 21, 17, 194 }, |
| { 66, 45, 25, 102, 197, 189, 23, 18, 22 } |
| }, |
| { |
| { 88, 88, 147, 150, 42, 46, 45, 196, 205 }, |
| { 43, 97, 183, 117, 85, 38, 35, 179, 61 }, |
| { 39, 53, 200, 87, 26, 21, 43, 232, 171 }, |
| { 56, 34, 51, 104, 114, 102, 29, 93, 77 }, |
| { 107, 54, 32, 26, 51, 1, 81, 43, 31 }, |
| { 39, 28, 85, 171, 58, 165, 90, 98, 64 }, |
| { 34, 22, 116, 206, 23, 34, 43, 166, 73 }, |
| { 68, 25, 106, 22, 64, 171, 36, 225, 114 }, |
| { 34, 19, 21, 102, 132, 188, 16, 76, 124 }, |
| { 62, 18, 78, 95, 85, 57, 50, 48, 51 } |
| }, |
| { |
| { 193, 101, 35, 159, 215, 111, 89, 46, 111 }, |

**Table 28** *(continued)*

| |
|---|
| { 60, 148, 31, 172, 219, 228, 21, 18, 111 }, |
| { 112, 113, 77, 85, 179, 255, 38, 120, 114 }, |
| { 40, 42, 1, 196, 245, 209, 10, 25, 109 }, |
| { 100, 80, 8, 43, 154, 1, 51, 26, 71 }, |
| { 88, 43, 29, 140, 166, 213, 37, 43, 154 }, |
| { 61, 63, 30, 155, 67, 45, 68, 1, 209 }, |
| { 142, 78, 78, 16, 255, 128, 34, 197, 171 }, |
| { 41, 40, 5, 102, 211, 183, 4, 1, 221 }, |
| { 51, 50, 17, 168, 209, 192, 23, 25, 82 } |
| }, |
| { |
| { 125, 98, 42, 88, 104, 85, 117, 175, 82 }, |
| { 95, 84, 53, 89, 128, 100, 113, 101, 45 }, |
| { 75, 79, 123, 47, 51, 128, 81, 171, 1 }, |
| { 57, 17, 5, 71, 102, 57, 53, 41, 49 }, |
| { 115, 21, 2, 10, 102, 255, 166, 23, 6 }, |
| { 38, 33, 13, 121, 57, 73, 26, 1, 85 }, |
| { 41, 10, 67, 138, 77, 110, 90, 47, 114 }, |
| { 101, 29, 16, 10, 85, 128, 101, 196, 26 }, |
| { 57, 18, 10, 102, 102, 213, 34, 20, 43 }, |
| { 117, 20, 15, 36, 163, 128, 68, 1, 26 } |
| }, |
| { |
| { 138, 31, 36, 171, 27, 166, 38, 44, 229 }, |
| { 67, 87, 58, 169, 82, 115, 26, 59, 179 }, |
| { 63, 59, 90, 180, 59, 166, 93, 73, 154 }, |
| { 40, 40, 21, 116, 143, 209, 34, 39, 175 }, |
| { 57, 46, 22, 24, 128, 1, 54, 17, 37 }, |
| { 47, 15, 16, 183, 34, 223, 49, 45, 183 }, |
| { 46, 17, 33, 183, 6, 98, 15, 32, 183 }, |
| { 65, 32, 73, 115, 28, 128, 23, 128, 205 }, |
| { 40, 3, 9, 115, 51, 192, 18, 6, 223 }, |
| { 87, 37, 9, 115, 59, 77, 64, 21, 47 } |
| }, |
| { |
| { 104, 55, 44, 218, 9, 54, 53, 130, 226 }, |
| { 64, 90, 70, 205, 40, 41, 23, 26, 57 }, |
| { 54, 57, 112, 184, 5, 41, 38, 166, 213 }, |
| { 30, 34, 26, 133, 152, 116, 10, 32, 134 }, |
| { 75, 32, 12, 51, 192, 255, 160, 43, 51 }, |
| { 39, 19, 53, 221, 26, 114, 32, 73, 255 }, |
| { 31, 9, 65, 234, 2, 15, 1, 118, 73 }, |
| { 88, 31, 35, 67, 102, 85, 55, 186, 85 }, |
| { 56, 21, 23, 111, 59, 205, 45, 37, 192 }, |
| { 55, 38, 70, 124, 73, 102, 1, 34, 98 } |

**Table 28** *(continued)*

| |
|---|
| }, |
| { |
| { 102, 61, 71, 37, 34, 53, 31, 243, 192 }, |
| { 69, 60, 71, 38, 73, 119, 28, 222, 37 }, |
| { 68, 45, 128, 34, 1, 47, 11, 245, 171 }, |
| { 62, 17, 19, 70, 146, 85, 55, 62, 70 }, |
| { 75, 15, 9, 9, 64, 255, 184, 119, 16 }, |
| { 37, 43, 37, 154, 100, 163, 85, 160, 1 }, |
| { 63, 9, 92, 136, 28, 64, 32, 201, 85 }, |
| { 86, 6, 28, 5, 64, 255, 25, 248, 1 }, |
| { 56, 8, 17, 132, 137, 255, 55, 116, 128 }, |
| { 58, 15, 20, 82, 135, 57, 26, 121, 40 } |
| }, |
| { |
| { 164, 50, 31, 137, 154, 133, 25, 35, 218 }, |
| { 51, 103, 44, 131, 131, 123, 31, 6, 158 }, |
| { 86, 40, 64, 135, 148, 224, 45, 183, 128 }, |
| { 22, 26, 17, 131, 240, 154, 14, 1, 209 }, |
| { 83, 12, 13, 54, 192, 255, 68, 47, 28 }, |
| { 45, 16, 21, 91, 64, 222, 7, 1, 197 }, |
| { 56, 21, 39, 155, 60, 138, 23, 102, 213 }, |
| { 85, 26, 85, 85, 128, 128, 32, 146, 171 }, |
| { 18, 11, 7, 63, 144, 171, 4, 4, 246 }, |
| { 35, 27, 10, 146, 174, 171, 12, 26, 128 } |
| }, |
| { |
| { 190, 80, 35, 99, 180, 80, 126, 54, 45 }, |
| { 85, 126, 47, 87, 176, 51, 41, 20, 32 }, |
| { 101, 75, 128, 139, 118, 146, 116, 128, 85 }, |
| { 56, 41, 15, 176, 236, 85, 37, 9, 62 }, |
| { 146, 36, 19, 30, 171, 255, 97, 27, 20 }, |
| { 71, 30, 17, 119, 118, 255, 17, 18, 138 }, |
| { 101, 38, 60, 138, 55, 70, 43, 26, 142 }, |
| { 138, 45, 61, 62, 219, 1, 81, 188, 64 }, |
| { 32, 41, 20, 117, 151, 142, 20, 21, 163 }, |
| { 112, 19, 12, 61, 195, 128, 48, 4, 24 } |
| } |
| } |

**Table 29 — intra_y_mode decoding probabilities**

| |
|---|
| bmodeProb[ 9 ] = { |
| 120, 90, 79, 133, 87, 85, 80, 111, 151 |
| } |

intra_uv_mode: The probabilities depend on the value of KeyFrame as follows:

— If KeyFrame is equal to 1, the probabilities are given by the array kfUvModeProb (see Table 30).

— Otherwise (KeyFrame is equal to 0), the probabilities are given by the array intra_chroma_prob.

**Table 30 — intra_uv_mode decoding probabilities**

| |
|---|
| kfUvModeProb[ 3 ] = { 142, 114, 183 } |

mv_mode: The probabilities are given by the array mvModeProb. The motion vector prediction process specified in 8.4.5 is invoked and the output assigned to array cnt. The array mvModeProb is computed based on the array cnt as follows:

mvModeProb[ 0 ] = vcbModeContexts[ cnt[ 0 ] ][ 0 ]

mvModeProb[ 1 ] = vcbModeContexts[ cnt[ 1 ] ][ 1 ]

mvModeProb[ 2 ] = vcbModeContexts[ cnt[ 2 ] ][ 2 ]

mvModeProb[ 3 ] = vcbModeContexts[ cnt[ 3 ] ][ 3 ]

where the array vcb_mode_contexts is as shown in Table 31.

**Table 31 — mv_mode decoding probabilities**

| |
|---|
| vcbModeContexts[ 6 ][ 4 ] = { |
| { 7, 1, 1, 143 }, |
| { 14, 18, 14, 107 }, |
| { 135, 64, 57, 68 }, |
| { 60, 56, 128, 65 }, |
| { 159, 134, 128, 34 }, |
| { 234, 188, 128, 28 } |
| } |

mv_split_mode: The probabilities are given by the array mvSplitProbs in Table 32.

**Table 32 — mv_split_mode decoding probabilities**

| |
|---|
| mvSplitProbs[ 3 ] = { 110, 111, 150 } |

sub_mv_mode: The sub-block motion vector prediction process as specified in 8.4.5.2 is invoked to generate the variables LeftMv and AboveMv.

mvContext is derived as follows:

— If LeftMv is equal to 0 and AboveMv is equal to 0, mvContext is set equal to 4.

— Otherwise, if LeftMv is equal to AboveMv, mvContext is set equal to 3.

— Otherwise, if AboveMv is equal to 0, mvContext is set equal to 2.

— Otherwise, if LeftMv is equal to 0, mvContext is set equal to 1.

— Otherwise, mvContext is set equal to 0.

The probabilities are given by the array subMvModeProb[mvContext] as shown in Table 33.

**Table 33 — sub_mv_mode decoding probabilities**

| subMvModeProb[ 5 ][ 3 ] = { |
|---|
| { 147, 136, 18 }, |
| { 106, 145, 1 }, |
| { 179, 121, 1 }, |
| { 223, 1, 34 }, |
| { 208, 1, 1 } |
| } |

mv_short: The probabilities are given by the array mv_prob[ cp ].

coeff_token: The probabilities depend on the variables ParseBlock, firstCoeff, i and the token context.

The variable ParseType is specified as:

— If firstCoeff is equal to 1, ParseType is set equal to 0 (luma coefficients when DC coded separately).

— Otherwise, if ParseBlock is less than 16, ParseType is set equal to 3 (luma coefficients).

— Otherwise, if ParseBlock is less than 24, ParseType is set equal to 2 (Cb or Cr coefficients).

— Otherwise, ParseType is set equal to 1 (Walsh-Hadamard coefficients).

The variable ParseBand is set equal to coeffBands[ i ] where coeffBands is specified as shown in Table 34.

**Table 34 — Mapping of coefficient position to coefficient band**

| coeffBands[ 16 ] = { 0, 1, 2, 3, 6, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7 } |
|---|

The variable ParseComplexity is set as follows.

— If i is greater than firstCoeff, the following applies:

  — If lastCoeff is equal to 0, ParseComplexity is set equal to 0.

  — Otherwise, if Abs( lastCoeff ) is equal to 1, ParseComplexity is set equal to 1.

  — Otherwise, ParseComplexity is set equal to 2.

— Otherwise, the following applies.

  — If ParseType is equal to 1, ParseComplexity is set to LeftToken[ 8 ] + AboveToken[ ParseRow ][ 8 ].

  — Otherwise, the following ordered steps apply:

    — ParseComplexity is set equal to 0.

    — The left ParseComplexity is applied as follows:

      — If ParseBlock is 0, 4, 8, or 12, ParseComplexity is set to LeftToken[ ParseBlock / 4 ].

      — Otherwise, if ParseBlock is 16, 18, 20, or 22, ParseComplexity is set to LeftToken[ 4 + (ParseBlock − 16) / 2 ].

      — Otherwise, ParseComplexity is set to NonZero[ ParseBlock − 1 ].

    — The above ParseComplexity is applied as follows.

      — If ParseBlock is 0, 1, 2, or 3, the following applies: