**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC1/SC29/WG11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11 N17732**

**July 2018, Ljubljana, SI**

|  |  |
| --- | --- |
| **Source** | **JVET** |
| **Status** | **Approved** |
| **Title** | **Working Draft 2 of Versatile Video Coding** |

|  |  |
| --- | --- |
| **Joint Video Experts Team (JVET)**  **of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11**  11th Meeting: Ljubljana, SI, 10–18 July 2018 | Document: JVET-K1001-v7 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **Versatile Video Coding (Draft 2)** | | |
| *Status:* | Output document approved by JVET | | |
| *Purpose:* | Draft text of video coding specification | | |
| *Author(s) or Contact(s):* | Benjamin Bross Jianle Chen Shan Liu | Email: | firstname.lastname@hhi.fraunhofer.de jianle.chen@huawei.com shanl@tencent.com |
| *Source:* | Editors | | |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Abstract

Since this is an early draft, topics outside of the specific aspects that have been established by recorded meeting agreements are not included in the specification. Such aspects are to be determined by further development of the VVC project in JVET. The high-level syntax for the standard is yet to be developed. The aspects of high-level syntax in this early draft are provided only to show how certain features are likely to be controlled by some high-level syntax that may have a sequence level, a picture level, and a slice level (a picture spatial region level that includes a subset of the CTUs of the picture).

Draft 1 of Versatile Video Coding.

Ed. Notes:

* Incorporated basic definitions, abbreviations and conventions
* Incorporated a basic high-level syntax (HLS) with NAL units, SPS, PPS and slice header.
* Incorporated block partitioning by a quadtree with nested multi-type tree using binary and ternary splits with
  + CU leaf nodes
  + Prediction at CU level
  + Transform at CU level
  + Minimum CU size with 4x4 luma coding block and corresponding chroma coding blocks (2x2 for 4:2:0)
  + Maximum TU size with 64x64 luma transform block and corresponding chroma transform blocks (32x32 for 4:2:0)
  + Minimum TU size with 4x4 luma transform block and corresponding chroma transform blocks (2x2 for 4:2:0)
  + Single tree for luma and chroma

Draft 2 of Versatile Video Coding.

Ed. Notes:

* Incorporated JVET-K0230: Separate trees for intra slices (without multi-DMs) with an implicit split to 64x64;
* Incorporated JVET-K0556: Prohibit ternary split of something bigger than 64 in width or height (and not send the bit to indicate ternary type at that level).
* Incorporated JVET-K0351 (test c): Keep only the TT restriction (preventing binary split with same orientation in center partition of the ternary split)
* Incorporated JVET-K0554: Implicit splitting at picture boundaries and ensure MinQTSize at boundary splits
* Fixed bug [#65](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/65) typos and unused variables in section 6.4
* Fixed bug [#67](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/67) implicit vertical BT split at picture boundary issue
* Incorporated JVET-K0072: Dependent quantization with fallback switch at the picture level and modified entropy coding supporting dependent quantization including:
  + adapted scaling to non-square transform blocks,
  + added binarization process for abs\_remainder,
  + specified CoeffMin and CoeffMax with fixed values,
  + added 0-th order Exp-Golomb code parsing process.
* Incorporated JVET-K0310: Sign data hiding (can only be used when dependent quantization is disabled).
* Incorporated JVET-K0529: Intra prediction using 3MPM on 67 prediction modes (Planar, DC and 65 angular modes)
* Incorporated JVET-K0122: DC prediction without division.
* Incorporated JVET-K0500: Wide-angle intra prediction.
* Incorporated JVET-K0063: Position-dependent intra prediction combination
* Incorporated JVET-K0190: Cross-component linear model intra prediction
* Incorporated multiple transfrom selection (MTS) for both intra and inter, each controlled by an SPS flag.
* Incorporated transform skip.
* Fixed bug [#68](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/68) various typos
* Fixed bug [#71](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/71) various typos
* Fixed bug [#72](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/72) on CCLM
* Incorporated JVET-K0357: adaptive motion vector resolution (AMVR)
* Incorpor,ated JVET-K0565: affine motion compensation (MC) including:
  + JVET-K0052: Affine merge bug fix
  + JVET-K0184: Affine MC (CE4.1.1a 4x4 fixed subblock size).
  + JVET-K0337: Affine MC coding and models (4.1.3a, affine MVP list construction, and 4.1.3b, MV difference coding, and 4.1.3c, 4/6 parameter model, no slice level switch).
  + JVET-K0367/JVET-K0052/JVET-K0103: Restriction of affine merge mode to CU sizes >= 8x8
* Incorporated 1/16 motion compensation (MC) including:
  + 1/16 MV storage
  + 1/16 merge and affine MVs
  + MVDs in AMVR accuracy (1/4,1,4) shifted to 1/16
  + Inter MVP candidates rounded to AMVR accuracy (1/4,1,4) and shifted to 1/16
  + 1/16 luma and 1/32 chroma interpolation filters
* Incorporated subblock-based temporal merging candidates with 8x8 motion vector storage (JVET-K0346).
* Incorporated JVET-K0371: 4x4 block classification based Adaptive Loop Filter (ALF).
* Fixed bug [#75](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/75) regarding a bottom and right boundary partition issue.
* Fixed bug [#90](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/90) typos in copying the control point vectors to temporal notion vectors.
* Fixed bug [#86](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/86) in intra reference sample filtering.
* Incorporated JVET-K0325: High Level Syntax (HLS) starting point.
* Fixed bug [#82](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/82) on zeroing-out high frequency transform coefficients for larger TUs (>32x32).
* Fixed bug [#85](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/85) on MTS index coding.

**CONTENTS**

*Page*

Abstract i

1 Scope 1

2 Normative references 1

2.1 Identical Recommendations | International Standards 1

2.2 Paired Recommendations | International Standards equivalent in technical content 1

2.3 Additional references 1

3 Definitions 1

4 Abbreviations 5

5 Conventions 8

5.1 General 8

5.2 Arithmetic operators 8

5.3 Logical operators 8

5.4 Relational operators 8

5.5 Bit-wise operators 8

5.6 Assignment operators 9

5.7 Range notation 9

5.8 Mathematical functions 9

5.9 Order of operation precedence 10

5.10 Variables, syntax elements and tables 11

5.11 Text description of logical operations 12

5.12 Processes 13

6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 14

6.1 Bitstream formats 14

6.2 Source, decoded and output picture formats 14

6.3 Partitioning of pictures, slices, CTUs 16

6.3.1 Partitioning of pictures into slices 16

6.3.2 Block, quadtree and multi-type tree structures 17

6.3.3 Spatial or component-wise partitionings 17

6.4 Availability processes 17

6.4.1 Allowed binary split process 17

6.4.2 Allowed ternary split process 18

6.5 Scanning processes 19

6.5.1 CTB raster and scanning process 19

6.5.2 Up-right diagonal scan order array initialization process 19

7 Syntax and semantics 20

7.1 Method of specifying syntax in tabular form 20

7.2 Specification of syntax functions and descriptors 21

7.3 Syntax in tabular form 22

7.3.1 NAL unit syntax 22

7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 23

7.3.3 Slice header syntax 25

7.3.4 Slice data syntax 27

7.4 Semantics 36

7.4.1 General 36

7.4.2 NAL unit semantics 36

7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 38

7.4.4 Slice header semantics 41

7.4.5 Slice data semantics 44

8 Decoding process 52

8.1 General decoding process 52

8.2 Decoding process for coding units coded in intra prediction mode 52

8.2.1 General decoding process for coding units coded in intra prediction mode 52

8.2.2 Derivation process for luma intra prediction mode 53

8.2.3 Derivation process for chroma intra prediction mode 54

8.2.4 Decoding process for intra blocks 55

8.3 Decoding process for coding units coded in inter prediction mode 68

8.3.1 General decoding process for coding units coded in inter prediction mode 68

8.3.2 Derivation process for motion vector components and reference indices 70

8.3.3 Derivation process for affine motion vector components and reference indices 89

8.3.4 Decoding process for inter blocks 99

8.3.5 Decoding process for the residual signal of coding blocks coded in inter prediction mode 105

8.4 Scaling, transformation and array construction process 106

8.4.1 Derivation process for quantization parameters 106

8.4.2 Scaling and transformation process 106

8.4.3 Scaling process for transform coefficients 107

8.4.4 Transformation process for scaled transform coefficients 108

8.4.5 Picture reconstruction process 116

8.5 In-loop Filter Process 116

8.5.1 General 116

8.5.2 Deblocking filter process 116

8.5.3 Adaptive loop filter process 116

9 Parsing process 121

9.1 General 121

9.2 Parsing process for 0-th order Exp-Golomb codes 121

9.2.1 General 121

9.2.2 Mapping process for signed Exp-Golomb codes 123

9.3 CABAC parsing process for slice segment data 124

9.3.1 General 124

9.3.2 Initialization process 124

9.3.3 Binarization process 124

9.3.4 Decoding process flow 130

# 

INTERNATIONAL STANDARD

ISO/IEC VVC

ITU-T Rec. H.VVC

ITU-T RECOMMENDATION

Versatile video coding

# Scope

This Recommendation | International Standard specifies versatile video coding.

# Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

## Identical Recommendations | International Standards

– None

## Paired Recommendations | International Standards equivalent in technical content

– None

## Additional references

– [Ed. (BB): Add references as needed.]

# Definitions

[Ed. (BB) included basic definitions to be updated.]

For the purposes of this Recommendation | International Standard, the following definitions apply.

* 1. **access unit**: A set of *NAL units* that are associated with each other according to a specified classification rule, are consecutive in *decoding order,* and contain exactly one *coded picture*.
  2. **AC transform coefficient**: Any *transform coefficient* for which the *frequency index* in at least one of the two dimensions is non-zero.
  3. **bin**: One bit of a *bin string*.
  4. **binarization**: A set of *bin strings* for all possible values of a *syntax element*.
  5. **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.
  6. **binary split**: A split of a rectangular MxN *block* of samples into two *blocks* where a vertical split results in a first (M / 2)xN *block* and a second (M / 2)xN *block*, and a horizontal split results in a first Mx(N / 2) *block* and a second Mx(N / 2) *block*.
  7. **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.
  8. **bi-predictive (B) slice**: A *slice* that is decoded using *intra* *prediction* or using *inter prediction* with at most two *motion vectors* and *reference indices* to *predict* the sample values of each *block*.
  9. **bitstream**: A sequence of bits, in the form of a *NAL unit stream* or a *byte stream*, that forms the representation of *coded pictures* and associated data forming one or more coded video sequences *(CVSs)*.
  10. **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.
  11. **byte**: A sequence of 8 bits, within which, when written or read as a sequence of bit values, the left-most and right-most bits represent the most and least significant bits, respectively.
  12. **byte-aligned**: A position in a *bitstream* is byte-aligned when the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream*, and a bit or *byte* or *syntax element* is said to be byte-aligned when the position at which it appears in a *bitstream* is byte-aligned.
  13. **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex TBD.
  14. **can**: A term used to refer to behaviour that is allowed, but not necessarily required*.*
  15. **chroma**: An adjective, represented by the symbols Cb and Cr, specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours.

NOTE – The term chroma is used rather than the term chrominance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term chrominance.

* 1. **coded picture**: A *coded representation* of a *picture* containing all *CTUs* of the *picture*.
  2. **coded representation**: A data element as represented in its coded form.
  3. **coded video sequence (CVS)**: A sequence of *access units* that consists, in *decoding order*, of an *IRAP access unit*, followed by zero or more *access* *units* that are not *IRAP access units*, including all subsequent *access units* up to but not including any subsequent *access unit* that is an *IRAP access unit*.
  4. **coding block**: An MxN *block* of samples for some values of M and N such that the division of a *CTB* into *coding blocks* is a *partitioning*.
  5. **coding tree block (CTB)**: An NxN *block* of samples for some value of N such that the division of a *component* into *CTBs* is a *partitioning*.
  6. **coding tree unit (CTU)**: A *CTB* of *luma* samples, two corresponding *CTBs* of *chroma* samples of a *picture* that has three sample arrays, or a *CTB* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.
  7. **coding unit (CU)**: A *coding block* of *luma* samples, two corresponding *coding blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.
  8. **component**: An array or single sample from one of the three arrays (*luma* and two *chroma*) that compose a *picture* in 4:2:0, 4:2:2, or 4:4:4 colour format or the array or a single sample of the array that compose a *picture* in monochrome format.
  9. **context variable**: A variable specified for the *adaptive binary arithmetic decoding* *process* of a *bin* by an equation containing recently decoded *bins*.
  10. **decoded picture**: A *decoded picture* is derived by decoding a *coded picture*.
  11. **decoder**: An embodiment of a *decoding process*.
  12. **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.
  13. **decoding process**: The process specified in this Specification that reads a *bitstream* and derives *decoded* *pictures* from it.
  14. **emulation prevention byte**: A *byte* equal to 0x03 that is present within a *NAL unit* when the *syntax elements* of the *bitstream* form certain patterns of *byte* values in a manner that ensures that no sequence of consecutive *byte-aligned* *bytes* in the *NAL unit* can contain a *start code prefix*.
  15. **encoder**: An embodiment of an *encoding process*.
  16. **encoding process**: A process not specified in this Specification that produces a *bitstream* conforming to this Specification.
  17. **flag**: A variable or single-bit *syntax element* that can take one of the two possible values: 0 and 1.
  18. **frequency index**: A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to the application of a *transform* in the *decoding process.*
  19. **informative**: A term used to refer to content provided in this Specification that does not establish any mandatory requirements for conformance to this Specification and thus is not considered an integral part of this Specification.
  20. **inter coding**: Coding of a *coding block*, *slice*, or *picture* that uses *inter prediction*.
  21. **inter prediction**: A *prediction* derived in a manner that is dependent on data elements (e.g., sample values or motion vectors) of one or more *reference* *pictures*.

NOTE – A prediction from a reference picture that is the current picture itself is also inter prediction.

* 1. **intra coding**: Coding of a *coding block, slice*, or *picture* that uses *intra prediction*.
  2. **intra prediction**: A *prediction* derived from only data elements (e.g., sample values) of the same decoded *slice* without referring to a *reference picture*.
  3. **intra random access point (IRAP) access unit**: An *access unit* in which the *coded picture* is an *IRAP picture*.
  4. **intra random access point (IRAP) picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type equal to IRAP\_NUT.
  5. **intra (I) slice**: A *slice* that is decoded using *intra prediction* only.
  6. **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.
  7. **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this Specification, or the value of a *transform coefficient* prior to *scaling*.

NOTE – The same set of levels is defined for all profiles, with most aspects of the definition of each level being in common across different profiles. Individual implementations may, within the specified constraints, support a different level for each supported profile.

* 1. **list 0 (list 1) motion vector**: A *motion vector* associated with a *reference index* pointing into *reference picture list 0* (*list 1*).
  2. **list 0 (list 1) prediction**: *Inter prediction* of the content of a *slice* using a *reference index* pointing into *reference picture list 0* (*list 1*).
  3. **luma**: An adjective, represented by the symbol or subscript Y or L, specifying that a sample array or single sample is representing the monochrome signal related to the primary colours.

NOTE – The term luma is used rather than the term luminance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term luminance. The symbol L is sometimes used instead of the symbol Y to avoid confusion with the symbol y as used for vertical location.

* 1. **may**: A term that is used to refer to behaviour that is allowed, but not necessarily required*.*

NOTE – In some places where the optional nature of the described behaviour is intended to be emphasized, the phrase "may or may not" is used to provide emphasis.

* 1. **motion vector**: A two-dimensional vector used for *inter prediction* that provides an offset from the coordinates in the *decoded picture* to the coordinates in a *reference picture*.
  2. **multi-type tree**: A *tree* in which a parent node can be split either into two child nodes using a *binary split* or into three child nodes using a *ternary split*, each of which may become parent node for another split into either two or three child nodes.
  3. **must**: A term that is used in expressing an observation about a requirement or an implication of a requirement that is specified elsewhere in this Specification (used exclusively in an *informative* context).
  4. **network abstraction layer (NAL) unit**: A *syntax structure* containing an indication of the type of data to follow and *bytes* containing that data in the form of an *RBSP* interspersed as necessary with *emulation prevention bytes*.
  5. **network abstraction layer (NAL) unit stream**: A sequence of *NAL units*.
  6. **non-IRAP picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type equal to NON\_IRAP\_NUT.
  7. **non-VCL NAL unit**: A *NAL unit* that is not a *VCL NAL unit*.
  8. **note**: A term that is used to prefix *informative* remarks (used exclusively in an *informative* context).
  9. **output order**: The order in which the *decoded* *pictures* are output from the *decoded picture buffer* (for the *decoded pictures* that are to be output from the *decoded picture buffer*).
  10. **parameter**: A *syntax element* of a *sequence parameter set (SPS)* or *picture parameter set (PPS)*, or the second word of the defined term *quantization parameter*.
  11. **partitioning**: The division of a set into subsets such that each element of the set is in exactly one of the subsets.
  12. **picture**: An array of *luma* samples in monochrome format or an array of *luma* samples and two corresponding arrays of *chroma* samples in 4:2:0, 4:2:2, and 4:4:4 colour format.

NOTE – A picture may be either a frame or a field. However, in one CVS, either all pictures are frames or all pictures are fields.

* 1. **picture parameter set (PPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *coded pictures* as determined by a *syntax element* found in each *slice header.*
  2. **picture order count (POC)**: A variable that is associated with each *picture*, uniquely identifies the associated *picture* among all *pictures* in the *CVS*, and, when the associated *picture* is to be output from the *decoded picture buffer*, indicates the position of the associated *picture* in *output order* relative to the *output order* positions of the other *pictures* in the same *CVS* that are to be output from the *decoded picture buffer*.
  3. **prediction**: An embodiment of the *prediction process*.
  4. **prediction process**: The use of a *predictor* to provide an estimate of the data element (e.g., sample value or motion vector) currently being decoded.
  5. **predictive (P) slice**: A *slice* that is decoded using *intra* *prediction* or using *inter prediction* with at most one *motion vector* and *reference index* to *predict* the sample values of each *block*.
  6. **predictor**: A combination of specified values or previously decoded data elements (e.g., sample value or motion vector) used in the *decoding process* of subsequent data elements.
  7. **profile**: A specified subset of the syntax of this Specification.
  8. **pulse code modulation (PCM)**: Coding of the samples of a *block* by directly representing the sample values without *prediction* or application of a transform.
  9. **quadtree**: A *tree* in which a parent node can be split into four child nodes, each of which may become parent node for another split into four child nodes.
  10. **quantization parameter**: A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.
  11. **random access**: The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.
  12. **raster scan**: A mapping of a rectangular two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the first 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.
  13. **raw byte sequence payload (RBSP)**: A *syntax structure* containing an integer number of *bytes* that is encapsulated in a *NAL unit* and that is either empty or has the form of a *string of data bits* containing *syntax elements* followed by an *RBSP stop bit* and zero or more subsequent bits equal to 0.
  14. **raw byte sequence payload (RBSP) stop bit**: A bit equal to 1 present within a *raw byte sequence payload (RBSP)* after a *string of data bits*, for which the location of the end within an *RBSP* can be identified by searching from the end of the *RBSP* for the *RBSP stop bit*, which is the last non-zero bit in the *RBSP.*
  15. **reference index**: An index into a *reference picture list*.
  16. **reference picture**: A *picture* that is a *short-term reference picture*.

NOTE – A reference picture contains samples that may be used for inter prediction in the decoding process of subsequent pictures in decoding order.

* 1. **reference picture list**: A list of *reference pictures* that is used for *inter prediction* of a *P* or *B slice.*

NOTE – For the decoding process of a P slice, there is one reference picture list – reference picture list 0. For the decoding process of a B slice, there are two reference picture lists – reference picture list 0 and reference picture list 1.

* 1. **reference picture list 0**: The *reference picture list* used for *inter prediction* of a *P* or the first *reference picture list* used for *inter prediction* of a *B* *slice*.
  2. **reference picture list 1**: The second *reference picture list* used for *inter prediction* of a *B slice*.
  3. **reserved**: A term that may be used to specify that some values of a particular *syntax element* are for future use by ITU-T | ISO/IEC and shall not be used in *bitstreams* conforming to this version of this Specification, but may be used in bitstreams conforming to future extensions of this Specification by ITU‑T | ISO/IEC.
  4. **residual**: The decoded difference between a *prediction* of a sample or data element and its decoded value.
  5. **scaling**: The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.
  6. **sequence parameter set (SPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *CVSs* as determined by the content of a *syntax element* found in the *PPS* referred to by a *syntax element* found in each *slice header.*
  7. **shall**: A term used to express mandatory requirements for conformance to this Specification.

NOTE – When used to express a mandatory constraint on the values of syntax elements or on the results obtained by operation of the specified decoding process, it is the responsibility of the encoder to ensure that the constraint is fulfilled. When used in reference to operations performed by the decoding process, any decoding process that produces identical cropped decoded pictures to those output from the decoding process described in this Specification conforms to the decoding process requirements of this Specification.

* 1. **short-term reference picture**: A *picture* that is marked as "used for short-term reference".
  2. **should**: A term used to refer to behaviour of an implementation that is encouraged to be followed under anticipated ordinary circumstances, but is not a mandatory requirement for conformance to this Specification.
  3. **slice**: An integer number of *CTUs* ordered consecutively in the *raster scan* and contained in a single *NAL unit*.
  4. **slice header**: A part of a coded *slice* containing the data elements pertaining to the first or all *CTUs* represented in the *slice*.
  5. **source**: A term used to describe the video material or some of its attributes before encoding.
  6. **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit*.

NOTE – The location of a start code prefix can be used by a decoder to identify the beginning of a new NAL unit and the end of a previous NAL unit. Emulation of start code prefixes is prevented within NAL units by the inclusion of emulation prevention bytes.

* 1. **string of data bits (SODB)**: A sequence of some number of bits representing *syntax elements* present within a *raw byte sequence payload* prior to the *raw byte sequence payload stop bit*, where the left-most bit is considered to be the first and most significant bit, and the right-most bit is considered to be the last and least significant bit.
  2. **syntax element**: An element of data represented in the *bitstream*.
  3. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
  4. **ternary split**: A split of a rectangular MxN *block* of samples into three *blocks* where a vertical split results in a first (M / 4)xN *block*, a second (M / 2)xN *block*, a third (M / 4)xN *block*, and a horizontal split results in a first Mx(N / 4) *block*, a second Mx(N / 2) *block*, a third Mx(N / 4) *block*.
  5. **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.
  6. **transform block**: A rectangular MxN *block* of samples resulting from a *transform* in the *decoding process*.
  7. **transform coefficient**: A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in a *transform* in the *decoding process*.
  8. **transform coefficient level**: An integer quantity representing the value associated with a particular two‑dimensional frequency index in the *decoding process* prior to *scaling* for computation of a *transform coefficient* value.
  9. **transform unit (TU)**: A *transform block* of *luma* samples and two corresponding *transform blocks* of *chroma* samples of a *picture* and *syntax structures* used to transform the *transform block* samples.
  10. **tree**: A tree is a finite set of nodes with a unique root node.
  11. **unspecified**: A term that may be used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.
  12. **video coding layer (VCL) NAL unit**: A collective term for *coded slice NAL units* and the subset of *NAL units* that have *reserved* values of nal\_unit\_type that are classified as VCL NAL units in this Specification.

# Abbreviations

[Ed. (BB) included some basic definitions (some of which are not currently used), to be updated.]

For the purposes of this Recommendation | International Standard, the following abbreviations apply.

ALF Adaptive Loop Filter

AMVR Adaptive Motion Vector Resolution

B Bi-predictive

CABAC Context-based Adaptive Binary Arithmetic Coding

CB Coding Block

CBR Constant Bit Rate

CPB Coded Picture Buffer

CRC Cyclic Redundancy Check

CTB Coding Tree Block

CTU Coding Tree Unit

CU Coding Unit

CVS Coded Video Sequence

DPB Decoded Picture Buffer

EG Exponential-Golomb

EGk k-th order Exponential-Golomb

FCC Federal Communications Commission (of the United States)

FIFO First-In, First-Out

FIR Finite Impulse Response

FL Fixed-Length

GBR Green, Blue and Red

I Intra

IRAP Intra Random Access Point

LPS Least Probable Symbol

LSB Least Significant Bit

MPS Most Probable Symbol

MSB Most Significant Bit

MTS Multiple Transform Selection

MVP Motion Vector Prediction

NAL Network Abstraction Layer

NTSC National Television System Committee (of the United States)

P Predictive

PCM Pulse Code Modulation

POC Picture Order Count

PPS Picture Parameter Set

QP Quantization Parameter

RBSP Raw Byte Sequence Payload

RGB Same as GBR

RPS Reference Picture Set

SAR Sample Aspect Ratio

SEI Supplemental Enhancement Information

SMPTE Society of Motion Picture and Television Engineers

SODB String Of Data Bits

SPS Sequence Parameter Set

TR Truncated Rice

UCS Universal Coded Character Set

UTF UCS Transmission Format

VBR Variable Bit Rate

VCL Video Coding Layer

# Conventions

## General

NOTE – The mathematical operators used in this Specification are similar to those used in the C programming language. However, the results of integer division and arithmetic shift operations are defined more precisely, and additional operations are defined, such as exponentiation and real-valued division. Numbering and counting conventions generally begin from 0, e.g., "the first" is equivalent to the 0-th, "the second" is equivalent to the 1-th, etc.

## Arithmetic operators

The following arithmetic operators are defined as follows:

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

## Logical operators

The following logical operators are defined as follows:

x && y Boolean logical "and" of x and y

x | | y Boolean logical "or" of x and y

! Boolean logical "not"

x ? y : z If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

## Relational operators

The following relational operators are defined as follows:

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

= = Equal to

!= Not equal to

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

## Bit-wise operators

The following bit-wise operators are defined as follows:

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

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

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

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

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

## Assignment operators

The following arithmetic operators are defined as follows:

= Assignment operator

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

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

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

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

## Range notation

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

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

## Mathematical functions

The following mathematical functions are defined:

Abs( x ) = (5‑1)

Asin( x ) the trigonometric inverse sine function, operating on an argument x that is  
in the range of −1.0 to 1.0, inclusive, with an output value in the range of   
−π÷2 to π÷2, inclusive, in units of radians (5‑2)

Atan( x ) the trigonometric inverse tangent function, operating on an argument x, with  
an output value in the range of −π÷2 to π÷2, inclusive, in units of radians (5‑3)

Atan2( y, x ) = (5‑4)

Ceil( x ) the smallest integer greater than or equal to x. (5‑5)

Clip1Y( x ) = Clip3( 0, ( 1 << BitDepthY ) − 1, x ) (5‑6)

Clip1C( x ) = Clip3( 0, ( 1 << BitDepthC ) − 1, x ) (5‑7)

Clip3( x, y, z ) = (5‑8)

Cos( x ) the trigonometric cosine function operating on an argument x in units of radians. (5‑9)

Floor( x ) the largest integer less than or equal to x. (5‑10)

GetCurrMsb( a, b, c, d ) = (5‑11)

Ln( x ) the natural logarithm of x (the base-e logarithm, where e is the natural logarithm base constant 2.718 281 828...). (5‑12)

Log2( x ) the base-2 logarithm of x. (5‑13)

Log10( x ) the base-10 logarithm of x. (5‑14)

Min( x, y ) = (5‑15)

Max( x, y ) = (5‑16)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (5‑17)

Sign( x ) = (5‑18)

Sin( x ) the trigonometric sine function operating on an argument x in units of radians (5‑19)

Sqrt( x ) = (5‑20)

Swap( x, y ) = ( y, x ) (5‑21)

Tan( x ) the trigonometric tangent function operating on an argument x in units of radians (5‑22)

## Order of operation precedence

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

– Operations of a higher precedence are evaluated before any operation of a lower precedence.

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

Table 5‑1 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

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

Table 5‑1 – Operation precedence from highest (at top of table) to lowest (at bottom of table)

|  |
| --- |
| **operations (with operands x, y, and z)** |
| "x++", "x− −" |
| "!x", "−x" (as a unary prefix operator) |
| xy |
| "x \* y", "x / y", "x ÷ y", "", "x % y" |
| "x + y", "x − y" (as a two-argument operator), "" |
| "x  <<  y", "x  >>  y" |
| "x < y", "x  <=  y", "x > y", "x  >=  y" |
| "x  = =  y", "x  !=  y" |
| "x & y" |
| "x | y" |
| "x  &&  y" |
| "x  | |  y" |
| "x ? y : z" |
| "x..y" |
| "x = y", "x  +=  y", "x  −=  y" |

## Variables, syntax elements and tables

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

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

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

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in clause 7.2 and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in clause 5.8) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[ x ][ y ] or as syx. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[ x ].

A specification of values of the entries in rows and columns of an array may be denoted by { {...} {...} }, where each inner pair of brackets specifies the values of the elements within a row in increasing column order and the rows are ordered in increasing row order. Thus, setting a matrix s equal to { { 1 6 } { 4 9 }} specifies that s[ 0 ][ 0 ] is set equal to 1, s[ 1 ][ 0 ] is set equal to 6, s[ 0 ][ 1 ] is set equal to 4, and s[ 1 ][ 1 ] is set equal to 9.

Binary notation is indicated by enclosing the string of bit values by single quote marks. For example, '01000001' represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation when the number of bits is an integer multiple of 4. For example, 0x41 represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Numerical values not enclosed in single quotes and not prefixed by "0x" are decimal values.

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

## Text description of logical operations

In the text, a statement of logical operations as would be described mathematically in the following form:

if( condition 0 )  
 statement 0  
else if( condition 1 )  
 statement 1  
...  
else /\* informative remark on remaining condition \*/  
 statement n

may be described in the following manner:

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

– If condition 0, statement 0

– Otherwise, if condition 1, statement 1

– ...

– Otherwise (informative remark on remaining condition), statement n

Each "If ... Otherwise, if ... Otherwise, ..." statement in the text is introduced with "... as follows" or "... the following applies" immediately followed by "If ... ". The last condition of the "If ... Otherwise, if ... Otherwise, ..." is always an "Otherwise, ...". Interleaved "If ... Otherwise, if ... Otherwise, ..." statements can be identified by matching "... as follows" or "... the following applies" with the ending "Otherwise, ...".

In the text, a statement of logical operations as would be described mathematically in the following form:

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

may be described in the following manner:

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

– If all of the following conditions are true, statement 0:

– condition 0a

– condition 0b

– Otherwise, if one or more of the following conditions are true, statement 1:

– condition 1a

– condition 1b

– ...

– Otherwise, statement n

In the text, a statement of logical operations as would be described mathematically in the following form:

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

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

## Processes

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

When invoking a process, the assignment of variables is specified as follows:

– If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.

– Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

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

# Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships

## Bitstream formats

This clause specifies the relationship between the network abstraction layer (NAL) unit stream and byte stream, either of which are referred to as the bitstream.

The bitstream can be in one of two formats: the NAL unit stream format or the byte stream format. The NAL unit stream format is conceptually the more "basic" type. It consists of a sequence of syntax structures called NAL units. This sequence is ordered in decoding order. There are constraints imposed on the decoding order (and contents) of the NAL units in the NAL unit stream.

The byte stream format can be constructed from the NAL unit stream format by ordering the NAL units in decoding order and prefixing each NAL unit with a start code prefix and zero or more zero-valued bytes to form a stream of bytes. The NAL unit stream format can be extracted from the byte stream format by searching for the location of the unique start code prefix pattern within this stream of bytes. Methods of framing the NAL units in a manner other than use of the byte stream format are outside the scope of this Specification. The byte stream format is specified in Annex TBD.

## Source, decoded and output picture formats

This clause specifies the relationship between source and decoded pictures that is given via the bitstream.

The video source that is represented by the bitstream is a sequence of pictures in decoding order.

The source and decoded pictures are each comprised of one or more sample arrays:

– Luma (Y) only (monochrome).

– Luma and two chroma (YCbCr or YCgCo).

– Green, blue, and red (GBR, also known as RGB).

– Arrays representing other unspecified monochrome or tri-stimulus colour samplings (for example, YZX, also known as XYZ).

For convenience of notation and terminology in this Specification, the variables and terms associated with these arrays are referred to as luma (or L or Y) and chroma, where the two chroma arrays are referred to as Cb and Cr; regardless of the actual colour representation method in use. The actual colour representation method in use can be indicated in syntax that is specified in Annex TBD.

The variables SubWidthC and SubHeightC are specified in Table 6‑1, depending on the chroma format sampling structure, which is specified through chroma\_format\_idc and separate\_colour\_plane\_flag. Other values of chroma\_format\_idc, SubWidthC and SubHeightC may be specified in the future by ITU‑T | ISO/IEC.

Table 6‑1 – SubWidthC and SubHeightC values derived from  
chroma\_format\_idc and separate\_colour\_plane\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **chroma\_format\_idc** | **separate\_colour\_plane\_flag** | **Chroma format** | **SubWidthC** | **SubHeightC** |
| 0 | 0 | Monochrome | 1 | 1 |
| 1 | 0 | 4:2:0 | 2 | 2 |
| 2 | 0 | 4:2:2 | 2 | 1 |
| 3 | 0 | 4:4:4 | 1 | 1 |
| 3 | 1 | 4:4:4 | 1 | 1 |

In monochrome sampling there is only one sample array, which is nominally considered the luma array.

In 4:2:0 sampling, each of the two chroma arrays has half the height and half the width of the luma array.

In 4:2:2 sampling, each of the two chroma arrays has the same height and half the width of the luma array.

In 4:4:4 sampling, depending on the value of separate\_colour\_plane\_flag, the following applies:

– If separate\_colour\_plane\_flag is equal to 0, each of the two chroma arrays has the same height and width as the luma array.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), the three colour planes are separately processed as monochrome sampled pictures.

The number of bits necessary for the representation of each of the samples in the luma and chroma arrays in a video sequence is in the range of 8 to 16, inclusive, and the number of bits used in the luma array may differ from the number of bits used in the chroma arrays.

When the value of chroma\_format\_idc is equal to 1, the nominal vertical and horizontal relative locations of luma and chroma samples in pictures are shown in Figure 6‑1. Alternative chroma sample relative locations may be indicated in video usability information (see Annex TBD).



Figure 6‑1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture

When the value of chroma\_format\_idc is equal to 2, the chroma samples are co-sited with the corresponding luma samples and the nominal locations in a picture are as shown in Figure 6‑2.



Figure 6‑2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture

When the value of chroma\_format\_idc is equal to 3, all array samples are co-sited for all cases of pictures and the nominal locations in a picture are as shown in Figure 6‑3.



Figure 6‑3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture

## Partitioning of pictures, slices, CTUs

### Partitioning of pictures into slices

This subclause specifies how a picture is partitioned into slices. Pictures are divided into slices. A slice is a a sequence of CTUs.

For example, a picture may be divided into two slices as shown in Figure 6‑4. In this example, the first slice contains 60 CTUs and the second slice contains the remaining 39 CTUs of the picture.

When a picture is coded using three separate colour planes (separate\_colour\_plane\_flag is equal to 1), a slice contains only CTUs of one colour component being identified by the corresponding value of colour\_plane\_id, and each colour component array of a picture consists of slices having the same colour\_plane\_id value. Coded slices with different values of colour\_plane\_id within a picture may be interleaved with each other under the constraint that for each value of colour\_plane\_id, the coded slice NAL units with that value of colour\_plane\_id shall be in the order of increasing CTU address in raster scan order for the first CTU of each coded slice NAL unit.

NOTE 1 – When separate\_colour\_plane\_flag is equal to 0, each CTU of a picture is contained in exactly one slice. When separate\_colour\_plane\_flag is equal to 1, each CTU of a colour component is contained in exactly one slice (i.e., information for each CTU of a picture is present in exactly three slices and these three slices have different values of colour\_plane\_id).



Figure 6‑4 – A picture with 11 by 9 luma CTUs that is partitioned into two slices (informative)

### Block, quadtree and multi-type tree structures

The samples are processed in units of CTBs. The array size for each luma CTB in both width and height is CtbSizeY in units of samples. The width and height of the array for each chroma CTB are CtbWidthC and CtbHeightC, respectively, in units of samples.

[Ed. (BB): Revise the following for QT+MTT.]

Each CTB is assigned a partition signalling to identify the block sizes for intra or inter prediction and for transform coding. The partitioning is a recursive quadtree partitioning. The root of the quadtree is associated with the CTB. The quadtree is split until a leaf is reached, which is referred to as the quadtree leaf. When the component width is not an integer number of the CTB size, the CTBs at the right component boundary are incomplete. When the component height is not an integer multiple of the CTB size, the CTBs at the bottom component boundary are incomplete.

The coding block is the root node of two trees, the prediction tree and the transform tree. The prediction tree specifies the position and size of prediction blocks. The transform tree specifies the position and size of transform blocks. The splitting information for luma and chroma is identical for the prediction tree and may or may not be identical for the transform tree.

The blocks and associated syntax structures are grouped into "unit" structures as follows:

– One transform block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three transform blocks (luma and chroma components of a picture in 4:2:0, 4:2:2 or 4:4:4 colour format) and the associated transform syntax structures units are associated with a transform unit.

– One coding block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three coding blocks (luma and chroma), the associated coding syntax structures and the associated transform units are associated with a coding unit.

– One CTB (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three CTBs (luma and chroma), the associated coding tree syntax structures and the associated coding units are associated with a CTU.

### Spatial or component-wise partitionings

The following divisions of processing elements of this Specification form spatial or component-wise partitioning:

– The division of each picture into components

– The division of each component into CTBs

– The division of each picture into slices

– The division of each slice into CTUs

– The division of each CTU into CTBs

– The division of each CTB into coding blocks, except that the CTBs are incomplete at the right component boundary when the component width is not an integer multiple of the CTB size and the CTBs are incomplete at the bottom component boundary when the component height is not an integer multiple of the CTB size

– The division of each CTU into coding units, except that the CTUs are incomplete at the right picture boundary when the picture width in luma samples is not an integer multiple of the luma CTB size and the CTUs are incomplete at the bottom picture boundary when the picture height in luma samples is not an integer multiple of the luma CTB size

– The division of each coding unit into transform units

– The division of each coding unit into coding blocks

– The division of each coding block into transform blocks

– The division of each transform unit into transform blocks.

## Availability processes

[Ed. (BB): Define appropriate availability checking process.]

### Allowed binary split process

Input to this process is a binary split mode btSplit, a coding block width cbWidth, a coding block height cbHeight, a location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, a multi-type tree depth mttDepth, a MaxMttDepth offset depthOffset and a partition index partIdx.

Output of this process is the variable allowBtSplit.

Table 6‑2 – Specification of parallelTtSplit, perpCbPos, perpCbSize, perpMaxPicSize and cbSize based on btSplit.

|  |  |  |
| --- | --- | --- |
|  | **btSplit = = SPLIT\_BT\_VER** | **btSplit = = SPLIT\_BT\_HOR** |
| **parallelTtSplit** | SPLIT\_TT\_VER | SPLIT\_TT\_HOR |
| **cbSize** | cbWidth | cbHeight |

The variables parallelTtSplit and cbSize are derived as specified in  Table 6‑2.

The variable allowBtSplit is derived as follows:

* If one or more of the following conditions are true, allowBtSplit is set equal to FALSE:
* cbSize is less than or equal to MinBtSizeY
* cbWidth is greater than MaxBtSizeY
* cbHeight is greater than MaxBtSizeY
* mttDepth is greater than or equal to MaxMttDepth + depthOffset
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_VER
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_HOR
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples
* y0 + cbHeight is smaller than or equal to pic\_height\_in\_luma\_samples
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE:
* mttDepth is greater than 0
* partIdx is equal to 1
* MttSplitMode[ x0 ][ y0 ][ mttDepth − 1 ] is equal to parallelTtSplit

– Otherwise, allowBtSplit is set equal to TRUE.

### Allowed ternary split process

Input to this process is a ternary split mode ttSplit, a coding block width cbWidth, a coding block height cbHeight, a location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, a multi-type tree depth mttDepth and a MaxMttDepth offset depthOffset.

Output of this process is the variable allowTtSplit.

Table 6‑3 – Specification of cbSize based on ttSplit.

|  |  |  |
| --- | --- | --- |
|  | **ttSplit = = SPLIT\_TT\_VER** | **ttSplit = = SPLIT\_TT\_HOR** |
| **cbSize** | cbWidth | cbHeight |

The variable cbSize is derived as specified in Table 6‑3.

The variable allowTtSplit is derived as follows:

* If one or more of the following conditions are true, allowTtSplit is set equal to FALSE:
* cbSize is less than or equal to 2 \* MinTtSizeY
* cbWidth is greater than MaxTtSizeY
* cbHeight is greater than MaxTtSizeY
* mttDepth is greater than or equal to MaxMttDepth + depthOffset
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples

– Otherwise, allowTtSplit is set equal to TRUE.

## Scanning processes

### CTB raster and scanning process

[Ed. (BB): Define appropriate scanning process.]

### Up-right diagonal scan order array initialization process

Input to this process is a block width blkWidth and a block size height blkHeight.

Output of this process is the array diagScan[ sPos ][ sComp ]. The array index sPos specify the scan position ranging from 0 to ( blkWidth \* blkHeight ) − 1. The array index sComp equal to 0 specifies the horizontal component and the array index sComp equal to 1 specifies the vertical component. Depending on the value of blkWidth and blkHeight, the array diagScan is derived as follows:

i = 0  
x = 0  
y = 0  
stopLoop = FALSE  
while( !stopLoop ) {  
 while( y >= 0 ) {  
 if( x < blkWidth && y < blkHeight ) { (6‑1)  
 diagScan[ i ][ 0 ] = x  
 diagScan[ i ][ 1 ] = y  
 i++  
 }  
 y− −  
 x++  
 }  
 y = x  
 x = 0  
 if( i >= blkWidth \* blkHeight )  
 stopLoop = TRUE  
}

# Syntax and semantics

## Method of specifying syntax in tabular form

The syntax tables specify a superset of the syntax of all allowed bitstreams. Additional constraints on the syntax may be specified, either directly or indirectly, in other clauses.

NOTE – An actual decoder should implement some means for identifying entry points into the bitstream and some means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified in this Specification.

The following table lists examples of the syntax specification format. When **syntax\_element** appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

|  |  |
| --- | --- |
|  | Descriptor |
| /\* A statement can be a syntax element with an associated descriptor or can be an expression used to specify conditions for the existence, type and quantity of syntax elements, as in the following two examples \*/ |  |
| **syntax\_element** | ue(k) |
| conditioning statement |  |
|  |  |
| /\* A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. \*/ |  |
| { |  |
| statement |  |
| statement |  |
| ... |  |
| } |  |
|  |  |
| /\* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true \*/ |  |
| while( condition ) |  |
| statement |  |
|  |  |
| /\* A "do ... while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true \*/ |  |
| do |  |
| statement |  |
| while( condition ) |  |
|  |  |
| /\* An "if ... else" structure specifies a test of whether a condition is true and, if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed \*/ |  |
| if( condition ) |  |
| primary statement |  |
| else |  |
| alternative statement |  |
|  |  |
| /\* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. \*/ |  |
| for( initial statement; condition; subsequent statement ) |  |
| primary statement |  |

## Specification of syntax functions and descriptors

The functions presented here are used in the syntactical description. These functions are expressed in terms of the value of a bitstream pointer that indicates the position of the next bit to be read by the decoding process from the bitstream.

byte\_aligned( ) is specified as follows:

– If the current position in the bitstream is on a byte boundary, i.e., the next bit in the bitstream is the first bit in a byte, the return value of byte\_aligned( ) is equal to TRUE.

– Otherwise, the return value of byte\_aligned( ) is equal to FALSE.

more\_data\_in\_byte\_stream( ), which is used only in the byte stream NAL unit syntax structure specified in Annex TBD, is specified as follows:

– If more data follow in the byte stream, the return value of more\_data\_in\_byte\_stream( ) is equal to TRUE.

– Otherwise, the return value of more\_data\_in\_byte\_stream( ) is equal to FALSE.

more\_data\_in\_payload( ) is specified as follows:

– If byte\_aligned( ) is equal to TRUE and the current position in the sei\_payload( ) syntax structure is 8 \* payloadSize bits from the beginning of the sei\_payload( ) syntax structure, the return value of more\_data\_in\_payload( ) is equal to FALSE.

– Otherwise, the return value of more\_data\_in\_payload( ) is equal to TRUE.

more\_rbsp\_data( ) is specified as follows:

– If there is no more data in the raw byte sequence payload (RBSP), the return value of more\_rbsp\_data( ) is equal to FALSE.

– Otherwise, the RBSP data are searched for the last (least significant, right-most) bit equal to 1 that is present in the RBSP. Given the position of this bit, which is the first bit (rbsp\_stop\_one\_bit) of the rbsp\_trailing\_bits( ) syntax structure, the following applies:

– If there is more data in an RBSP before the rbsp\_trailing\_bits( ) syntax structure, the return value of more\_rbsp\_data( ) is equal to TRUE.

– Otherwise, the return value of more\_rbsp\_data( ) is equal to FALSE.

The method for enabling determination of whether there is more data in the RBSP is specified by the application (or in Annex TBD for applications that use the byte stream format).

more\_rbsp\_trailing\_data( ) is specified as follows:

– If there is more data in an RBSP, the return value of more\_rbsp\_trailing\_data( ) is equal to TRUE.

– Otherwise, the return value of more\_rbsp\_trailing\_data( ) is equal to FALSE.

next\_bits( n ) provides the next bits in the bitstream for comparison purposes, without advancing the bitstream pointer. Provides a look at the next n bits in the bitstream with n being its argument. When used within the byte stream format as specified in Annex TBD and fewer than n bits remain within the byte stream, next\_bits( n ) returns a value of 0.

payload\_extension\_present( ) is specified as follows:

– If the current position in the sei\_payload( ) syntax structure is not the position of the last (least significant, right-most) bit that is equal to 1 that is less than 8 \* payloadSize bits from the beginning of the syntax structure (i.e., the position of the payload\_bit\_equal\_to\_one syntax element), the return value of payload\_extension\_present( ) is equal to TRUE.

– Otherwise, the return value of payload\_extension\_present( ) is equal to FALSE.

read\_bits( n ) reads the next n bits from the bitstream and advances the bitstream pointer by n bit positions. When n is equal to 0, read\_bits( n ) is specified to return a value equal to 0 and to not advance the bitstream pointer.

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

– ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in clause TBD.

– b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read\_bits( 8 ).

– f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ).

– i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a two's complement integer representation with most significant bit written first.

– se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k equal to 0.

– st(v): null-terminated string encoded as universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in ISO/IEC 10646. The parsing process is specified as follows: st(v) begins at a byte-aligned position in the bitstream and reads and returns a series of bytes from the bitstream, beginning at the current position and continuing up to but not including the next byte-aligned byte that is equal to 0x00, and advances the bitstream pointer by ( stringLength + 1 ) \* 8 bit positions, where stringLength is equal to the number of bytes returned.

NOTE – The st(v) syntax descriptor is only used in this Specification when the current position in the bitstream is a byte-aligned position.

– tb(v): truncated binary using up to maxVal bits with maxVal defined in the semantics of the symtax element. The parsing process for this descriptor is specified in clause 9.4.

– tu(v): truncated unary using up to maxVal bits with maxVal defined in the semantics of the symtax element. The parsing process for this descriptor is specified in clause 9.3.

– u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a binary representation of an unsigned integer with most significant bit written first.

– ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k equal to 0.

– uek(v): unsigned integer k-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k defined in the semantics of the symtax element.

## Syntax in tabular form

### NAL unit syntax

#### General NAL unit syntax

|  |  |
| --- | --- |
| nal\_unit( NumBytesInNalUnit ) { | Descriptor |
| nal\_unit\_header( ) |  |
| NumBytesInRbsp = 0 |  |
| for( i = 2; i < NumBytesInNalUnit; i++ ) |  |
| if( i + 2 < NumBytesInNalUnit && next\_bits( 24 ) = = 0x000003 ) { |  |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| i += 2 |  |
| **emulation\_prevention\_three\_byte** /\* equal to 0x03 \*/ | f(8) |
| } else |  |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| } |  |

#### NAL unit header syntax

|  |  |
| --- | --- |
| nal\_unit\_header( ) { | Descriptor |
| **forbidden\_zero\_bit** | f(1) |
| **nal\_unit\_type** | u(5) |
| **nuh\_temporal\_id\_plus1** | u(3) |
| **nuh\_reserved\_zero\_7bits** | u(7) |
| } |  |

### Raw byte sequence payloads, trailing bits and byte alignment syntax

#### Sequence parameter set RBSP syntax

|  |  |
| --- | --- |
| seq\_parameter\_set\_rbsp( ) { | Descriptor |
| **sps\_seq\_parameter\_set\_id** | ue(v) |
| **chroma\_format\_idc** | ue(v) |
| if( chroma\_format\_idc = = 3 ) |  |
| **separate\_colour\_plane\_flag** | u(1) |
| **pic\_width\_in\_luma\_samples** | ue(v) |
| **pic\_height\_in\_luma\_samples** | ue(v) |
| **bit\_depth\_luma\_minus8** | ue(v) |
| **bit\_depth\_chroma\_minus8** | ue(v) |
| **qtbtt\_dual\_tree\_intra\_flag** | ue(v) |
| **log2\_ctu\_size\_minus2** | ue(v) |
| **log2\_min\_qt\_size\_intra\_slices\_minus2** | ue(v) |
| **log2\_min\_qt\_size\_inter\_slices\_minus2** | ue(v) |
| **max\_mtt\_hierarchy\_depth\_inter\_slices** | ue(v) |
| **max\_mtt\_hierarchy\_depth\_intra\_slices** | ue(v) |
| **sps\_cclm\_enabled\_flag** | u(1) |
| **sps\_alf\_enabled\_flag** | u(1) |
| **sps\_temporal\_mvp\_enabled\_flag** | u(1) |
| if( sps\_temporal\_mvp\_enabled\_flag ) |  |
| **sps\_sbtmvp\_enabled\_flag** | u(1) |
| if( sps\_sbtmvp\_enabled\_flag ) |  |
| **log2\_sbtmvp\_default\_size\_minus2** | u(1) |
| **sps\_amvr\_enabled\_flag** | u(1) |
| **sps\_affine\_enabled\_flag** | u(1) |
| if( sps\_affine\_enabled\_flag ) |  |
| **sps\_affine\_type\_flag** | u(1) |
| **sps\_mts\_intra\_enabled\_flag** | u(1) |
| **sps\_mts\_inter\_enabled\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Picture parameter set RBSP syntax

|  |  |
| --- | --- |
| pic\_parameter\_set\_rbsp( ) { | Descriptor |
| **pps\_pic\_parameter\_set\_id** | ue(v) |
| **pps\_seq\_parameter\_set\_id** | ue(v) |
| **transform\_skip\_enabled\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### End of sequence RBSP syntax

|  |  |
| --- | --- |
| end\_of\_seq\_rbsp( ) { | Descriptor |
| } |  |

#### End of bitstream RBSP syntax

|  |  |
| --- | --- |
| end\_of\_bitstream\_rbsp( ) { | Descriptor |
| } |  |

#### Slice layer RBSP syntax

|  |  |
| --- | --- |
| slice\_layer\_rbsp( ) { | Descriptor |
| slice\_header( ) |  |
| slice\_data( ) |  |
| rbsp\_slice\_trailing\_bits( ) |  |
| } |  |

#### RBSP slice trailing bits syntax

|  |  |
| --- | --- |
| rbsp\_slice\_trailing\_bits( ) { | Descriptor |
| rbsp\_trailing\_bits( ) |  |
| while( more\_rbsp\_trailing\_data( ) ) |  |
| **cabac\_zero\_word** /\* equal to 0x0000 \*/ | f(16) |
| } |  |

#### RBSP trailing bits syntax

|  |  |
| --- | --- |
| rbsp\_trailing\_bits( ) { | Descriptor |
| **rbsp\_stop\_one\_bit** /\* equal to 1 \*/ | f(1) |
| while( !byte\_aligned( ) ) |  |
| **rbsp\_alignment\_zero\_bit** /\* equal to 0 \*/ | f(1) |
| } |  |

#### Byte alignment syntax

|  |  |
| --- | --- |
| byte\_alignment( ) { | Descriptor |
| **alignment\_bit\_equal\_to\_one** /\* equal to 1 \*/ | f(1) |
| while( !byte\_aligned( ) ) |  |
| **alignment\_bit\_equal\_to\_zero** /\* equal to 0 \*/ | f(1) |
| } |  |

### Slice header syntax

#### General slice header syntax

|  |  |
| --- | --- |
| slice\_header( ) { | Descriptor |
| **slice\_pic\_parameter\_set\_id** | ue(v) |
| **slice\_address** | u(v) |
| **slice\_type** | ue(v) |
| if ( slice\_type != I ) { |  |
| **log2\_diff\_ctu\_max\_bt\_size** | ue(v) |
| if( sps\_sbtmvp\_enabled\_flag ) { |  |
| **sbtmvp\_size\_override\_flag** | u(1) |
| if( sbtmvp\_size\_override\_flag ) |  |
| **log2\_sbtmvp\_active\_size\_minus2** | u(3) |
| } |  |
| if( sps\_temporal\_mvp\_enabled\_flag ) |  |
| **slice\_temporal\_mvp\_enabled\_flag** | u(1) |
| if( slice\_type = = B ) |  |
| **mvd\_l1\_zero\_flag** | u(1) |
| if( slice\_temporal\_mvp\_enabled\_flag ) { |  |
| if( slice\_type = = B ) |  |
| **collocated\_from\_l0\_flag** | u(1) |
| } |  |
| **six\_minus\_max\_num\_merge\_cand** | ue(v) |
| } |  |
| if ( sps\_alf\_enabled\_flag ) { |  |
| **slice\_alf\_enabled\_flag** | u(1) |
| if( slice\_alf\_enabled\_flag ) |  |
| alf\_data( ) |  |
| } |  |
| **dep\_quant\_enabled\_flag** | u(1) |
| if( !dep\_quant\_enabled\_flag ) |  |
| **sign\_data\_hiding\_enabled\_flag** | u(1) |
| byte\_alignment( ) |  |
| } |  |

#### Adaptive loop filter data syntax

|  |  |
| --- | --- |
| alf\_data( ) { | Descriptor |
| **alf\_chroma\_idc** | tu(v) |
| **alf\_luma\_num\_filters\_signalled\_minus1** | tb(v) |
| **alf\_luma\_type\_flag** | u(1) |
| if( alf\_luma\_num\_filters\_signalled\_minus1 > 0 ) { |  |
| for( filtIdx = 0; filtIdx < NumAlfFilters; filtIdx++ ) |  |
| **alf\_luma\_coeff\_delta\_idx**[ filtIdx ] | tb(v) |
| } |  |
| **alf\_luma\_coeff\_delta\_flag** | u(1) |
| if ( !alf\_luma\_coeff\_delta\_flag && alf\_luma\_num\_filters\_signalled\_minus1 > 0 ) |  |
| **alf\_luma\_coeff\_delta\_prediction\_flag** | u(1) |
| **alf\_luma\_min\_eg\_order\_minus1** | ue(v) |
| for( i = 0; i < ( alf\_luma\_type\_flag = = 1 ) ? 2 : 3; i++ ) |  |
| **alf\_luma\_eg\_order\_increase\_flag**[ i ] | u(1) |
| if ( alf\_luma\_coeff\_delta\_flag ) { |  |
| for( sigFiltIdx = 0; sigFiltIdx <= alf\_luma\_num\_filters\_signalled\_minus1; sigFiltIdx++ ) |  |
| **alf\_luma\_coeff\_flag**[ sigFiltIdx ] | u(1) |
| } |  |
| for( sigFiltIdx = 0; sigFiltIdx <= alf\_luma\_num\_filters\_signalled\_minus1; sigFiltIdx++ ) { |  |
| if ( alf\_luma\_coeff\_flag[ sigFiltIdx ] ) { |  |
| for ( j = 0; j < ( alf\_luma\_type\_flag = = 1 ) ? 6 : 12; j++ ) { |  |
| **alf\_luma\_coeff\_delta\_abs**[ sigFiltIdx ][ j ] | uek(v) |
| if( alf\_luma\_coeff\_delta\_abs[ sigFiltIdx ][ j ] ) |  |
| **alf\_luma\_coeff\_delta\_sign**[ sigFiltIdx ][ j ] | u(1) |
| } |  |
| } |  |
| } |  |
| if ( alf\_chroma\_idc > 0 ) { |  |
| **alf\_chroma\_min\_eg\_order\_minus1** | ue(v) |
| for( i = 0; i < 2; i++ ) |  |
| **alf\_chroma\_eg\_order\_increase\_flag**[ i ] | u(1) |
| **alf\_chroma\_ctb\_present\_flag** | u(1) |
| for( j = 0; j < 6; j++ ) { |  |
| **alf\_chroma\_coeff\_abs**[ j ] | uek(v) |
| if( alf\_chroma\_coeff\_abs[ j ] > 0 ) |  |
| **alf\_chroma\_coeff\_sign**[ j ] | u(1) |
| } |  |
| } |  |
| } |  |

### Slice data syntax

#### General slice data syntax

|  |  |
| --- | --- |
| slice\_data( ) { | Descriptor |
| do { |  |
| coding\_tree\_unit( ) |  |
| **end\_of\_slice\_flag** | ae(v) |
| CtbAddrInRs++ |  |
| } while( !end\_of\_slice\_flag ) |  |
| } |  |

#### Coding tree unit syntax

|  |  |
| --- | --- |
| coding\_tree\_unit( ) { | Descriptor |
| xCtb = ( CtbAddrInRs % PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| yCtb = ( CtbAddrInRs / PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| if( slice\_alf\_enable\_flag ){ |  |
| **alf\_ctb\_flag**[ 0 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| if( alf\_chroma\_ctb\_present\_flag ) { |  |
| if( alf\_chroma\_idc  = =  1  | |  alf\_chroma\_idc  = =  3 ) |  |
| **alf\_ctb\_flag**[ 1 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| if( alf\_chroma\_idc  = =  2  | |  alf\_chroma\_idc  = =  3 ) |  |
| **alf\_ctb\_flag**[ 2 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| } |  |
| } |  |
| if( slice\_type = = I && qtbtt\_dual\_tree\_intra\_flag ) { |  |
| dual\_tree\_implicit\_qt\_split ( xCtb, yCtb, CtbLog2SizeY, 0 ) |  |
| else |  |
| coding\_quadtree( xCtb, yCtb, CtbLog2SizeY, 0, SINGLE\_TREE ) |  |
| } |  |

|  |  |
| --- | --- |
| dual\_tree\_implicit\_qt\_split( x0, y0, log2CbSize, cqtDepth ) { | Descriptor |
| if( log2CbSize > 6 ) { |  |
| x1 = x0 + ( 1  <<  ( log2CbSize − 1 ) ) |  |
| y1 = y0 + ( 1  <<  ( log2CbSize − 1 ) ) |  |
| dual\_tree\_implicit\_qt\_split( x0, y0, log2CbSize − 1, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x1, y0, log2CbSize − 1, cqtDepth + 1 ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x0, y1, log2CbSize − 1, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples && y1 < pic\_height\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x1, y1, log2CbSize − 1, cqtDepth + 1 ) |  |
| } else { |  |
| coding\_quadtree( x0, y0, log2CbSize, cqtDepth, DUAL\_TREE\_LUMA ) |  |
| coding\_quadtree( x0, y0, log2CbSize, cqtDepth, DUAL\_TREE\_CHROMA ) |  |
| } |  |
| } |  |

#### Coding quadtree syntax

|  |  |
| --- | --- |
| coding\_quadtree( x0, y0, log2CbSize, cqtDepth, treeType ) { | Descriptor |
| if( ( ( ( x0 + ( 1  <<  log2CbSize ) <= pic\_width\_in\_luma\_samples )  ?  1  :  0 ) +   ( ( y0 + ( 1  <<  log2CbSize ) <= pic\_height\_in\_luma\_samples )  ?  1  :  0 ) +   ( ( ( 1 << log2CbSize ) <= MaxBtSizeY )  ?  1  :  0 ) ) >= 2 &&   log2CbSize > MinQtLog2SizeY ) |  |
| **qt\_split\_cu\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( qt\_split\_cu\_flag[ x0 ][ y0 ] ) { |  |
| x1 = x0 + ( 1  <<  ( log2CbSize − 1 ) ) |  |
| y1 = y0 + ( 1  <<  ( log2CbSize − 1 ) ) |  |
| coding\_quadtree( x0, y0, log2CbSize − 1, cqtDepth + 1, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_quadtree( x1, y0, log2CbSize − 1, cqtDepth + 1, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_quadtree( x0, y1, log2CbSize − 1, cqtDepth + 1, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples && y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_quadtree( x1, y1, log2CbSize − 1, cqtDepth + 1, treeType ) |  |
| } else |  |
| multi\_type\_tree( x0, y0, 1  <<  log2CbSize, 1  <<  log2CbSize, 0, 0, 0, treeType ) |  |
| } |  |

#### Multi-type tree syntax

|  |  |
| --- | --- |
| multi\_type\_tree( x0, y0, cbWidth, cbHeight, mttDepth, depthOffset,  partIdx, treeType ) { | Descriptor |
| if( ( allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor ) &&   ( x0 + cbWidth <= pic\_width\_in\_luma\_samples ) &&   (y0 + cbHeight <= pic\_height\_in\_luma\_samples ) ) |  |
| **mtt\_split\_cu\_flag** | ae(v) |
| if( mtt\_split\_cu\_flag ) { |  |
| if( ( allowSplitBtHor | | allowSplitTtHor ) &&   ( allowSplitBtVer | | allowSplitTtVer ) ) |  |
| **mtt\_split\_cu\_vertical\_flag** | ae(v) |
| if( ( allowSplitBtVer && allowSplitTtVer && mtt\_split\_cu\_vertical\_flag ) | |   ( allowSplitBtHor && allowSplitTtHor && !mtt\_split\_cu\_vertical\_flag ) ) |  |
| **mtt\_split\_cu\_binary\_flag** | ae(v) |
| if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_BT\_VER ) { |  |
| depthOffset  +=  ( x0 + cbWidth  >  pic\_width\_in\_luma\_samples ) ? 1 : 0 |  |
| x1 = x0 + ( cbWidth / 2 ) |  |
| multi\_type\_tree( x0, y0, cbWidth / 2, cbHeight, mttDepth + 1, depthOffset, 0, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| multi\_type\_tree( x1, y0, cbWidth / 2, cbHeightY, mttDepth + 1, depthOffset, 1, treeType ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_BT\_HOR ) { |  |
| depthOffset  +=  ( y0 + cbHeight  >  pic\_height\_in\_luma\_samples ) ? 1 : 0 |  |
| y1 = y0 + ( cbHeight / 2 ) |  |
| multi\_type\_tree( x0, y0, cbWidth, cbHeight / 2, mttDepth + 1, depthOffset, 0, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| multi\_type\_tree( x0, y1, cbWidth, cbHeight / 2, mttDepth + 1, depthOffset, 1, treeType ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_TT\_VER ) { |  |
| x1 = x0 + ( cbWidth / 4 ) |  |
| x2 = x0 + ( 3 \* cbWidth / 4 ) |  |
| multi\_type\_tree( x0, y0, cbWidth / 4, cbHeight, mttDepth + 1, depthOffset, 0, treeType ) |  |
| multi\_type\_tree( x1, y0, cbWidth / 2, cbHeight, mttDepth + 1, depthOffset, 1, treeType ) |  |
| multi\_type\_tree( x2, y0, cbWidth / 4, cbHeight, mttDepth + 1, depthOffset, 2, treeType ) |  |
| } else { /\* SPLIT\_TT\_HOR \*/ |  |
| y1 = y0 + ( cbHeight / 4 ) |  |
| y2 = y0 + ( 3 \* cbHeight / 4 ) |  |
| multi\_type\_tree( x0, y0, cbWidth, cbHeight / 4, mttDepth + 1, depthOffset, 0, treeType ) |  |
| multi\_type\_tree( x0, y1, cbWidth, cbHeight / 2, mttDepth + 1, depthOffset, 1, treeType ) |  |
| multi\_type\_tree( x0, y2, cbWidth, cbHeight / 4, mttDepth + 1, depthOffset, 2 , treeType) |  |
| } |  |
| } else |  |
| coding\_unit( x0, y0, cbWidth, cbHeight, treeType ) |  |
| } |  |

#### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, cbWidth, cbHeight, treeType ) { | Descriptor |
| if( slice\_type != I ) { |  |
| **cu\_skip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( cu\_skip\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **pred\_mode\_flag** | ae(v) |
| } |  |
| if( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA ) { |  |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| **intra\_luma\_mpm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_luma\_mpm\_flag[ x0 ][ y0 ] ) |  |
| **intra\_luma\_mpm\_idx**[ x0 ][ y0 ] | ae(v) |
| else |  |
| **intra\_luma\_mpm\_remainder**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA ) |  |
| **intra\_chroma\_pred\_mode**[ x0 ][ y0 ] | ae(v) |
| } else { /\* MODE\_INTER \*/ |  |
| if( cu\_skip\_flag[ x0 ][ y0 ] ) { |  |
| if( sps\_affine\_enabled\_flag && cbWidth >= 8 && cbHeight >= 8 &&  ( MotionModelIdc[ x0 − 1 ][ y0 + cbHeight − 1 ] != 0 | |   MotionModelIdc[ x0 − 1 ][ y0 + cbHeight ] != 0 | |   MotionModelIdc[ x0 − 1 ][ y0 − 1 ] != 0 | |   MotionModelIdc[ x0 + cbWidth − 1 ][ y0 − 1 ] != 0 | |   MotionModelIdc[ x0 + cbWidth ][ y0 − 1 ]] != 0 ) ) |  |
| **merge\_affine\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_affine\_flag[ x0 ][ y0 ] = = 0 && MaxNumMergeCand > 1 ) |  |
| **merge\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| **merge\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_flag[ x0 ][ y0 ] ) { |  |
| if( sps\_affine\_enabled\_flag && cbWidth >= 8 && cbHeight >= 8 &&  ( MotionModelIdc[ x0 − 1 ][ y0 + cbHeight − 1 ] != 0 | |   MotionModelIdc[ x0 − 1 ][ y0 + cbHeight ] != 0 | |   MotionModelIdc[ x0 − 1 ][ y0 − 1 ] != 0 | |   MotionModelIdc[ x0 + cbWidth − 1 ][ y0 − 1 ] != 0 | |   MotionModelIdc[ x0 + cbWidth ][ y0 − 1 ]] != 0 ) ) |  |
| **merge\_affine\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_affine\_flag[ x0 ][ y0 ] = = 0 && MaxNumMergeCand > 1 ) |  |
| **merge\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( slice\_type = = B ) |  |
| **inter\_pred\_idc**[ x0 ][ y0 ] | ae(v) |
| if( sps\_affine\_enabled\_flag && cbWidth >= 16 && cbHeight >= 16 ) { |  |
| **inter\_affine\_flag**[ x0 ][ y0 ] | ae(v) |
| if( sps\_affine\_type\_flag && inter\_affine\_flag[ x0 ][ y0 ] ) |  |
| **cu\_affine\_type\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( inter\_pred\_idc[ x0 ][ y0 ] != PRED\_L1 ) { |  |
| if( num\_ref\_idx\_l0\_active\_minus1 > 0 ) |  |
| **ref\_idx\_l0**[ x0 ][ y0 ] | ae(v) |
| mvd\_coding( x0, y0, 0, 0 ) |  |
| if( MotionModelIdc[ x0 ][ y0 ] > 0 ) |  |
| mvd\_coding( x0, y0, 0, 1 ) |  |
| if(MotionModelIdc[ x0 ][ y0 ] > 1 ) |  |
| mvd\_coding( x0, y0, 0, 2 ) |  |
| **mvp\_l0\_flag**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| MvdL0[ x0 ][ y0 ][ 0 ] = 0 |  |
| MvdL0[ x0 ][ y0 ][ 1 ] = 0 |  |
| } |  |
| if( inter\_pred\_idc[ x0 ][ y0 ] != PRED\_L0 ) { |  |
| if( num\_ref\_idx\_l1\_active\_minus1 > 0 ) |  |
| **ref\_idx\_l1**[ x0 ][ y0 ] | ae(v) |
| if( mvd\_l1\_zero\_flag && inter\_pred\_idc[ x0 ][ y0 ] = = PRED\_BI ) { |  |
| MvdL1[ x0 ][ y0 ][ 0 ] = 0 |  |
| MvdL1[ x0 ][ y0 ][ 1 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 0 ][ 0 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 0 ][ 1 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 1 ][ 0 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 1 ][ 1 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 2 ][ 0 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 2 ][ 1 ] = 0 |  |
| } else { |  |
| mvd\_coding( x0, y0, 1, 0 ) |  |
| if( MotionModelIdc[ x0 ][ y0 ] > 0 ) |  |
| mvd\_coding( x0, y0, 1, 1 ) |  |
| if(MotionModelIdc[ x0 ][ y0 ] > 1 ) |  |
| mvd\_coding( x0, y0, 1, 2 ) |  |
| **mvp\_l1\_flag**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| MvdL1[ x0 ][ y0 ][ 0 ] = 0 |  |
| MvdL1[ x0 ][ y0 ][ 1 ] = 0 |  |
| } |  |
| if( sps\_amvr\_enabled\_flag && inter\_affine\_flag = = 0 &&  ( MvdL0[ x0 ][ y0 ][ 0 ] != 0 | | MvdL0[ x0 ][ y0 ][ 1 ] != 0 | |  MvdL1[ x0 ][ y0 ][ 0 ] != 0 | | MvdL1[ x0 ][ y0 ][ 1 ] != 0 ) ) |  |
| **amvr\_mode**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| if( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA && cu\_skip\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **cu\_cbf** | ae(v) |
| if( cu\_cbf ) { |  |
| transform\_tree( x0, y0, cbWidth, cbHeight, treeType ) |  |
| } |  |

#### Motion vector difference syntax

|  |  |
| --- | --- |
| mvd\_coding( x0, y0, refList ,cpIdx ) { | **Descriptor** |
| **abs\_mvd\_greater0\_flag**[ 0 ] | ae(v) |
| **abs\_mvd\_greater0\_flag**[ 1 ] | ae(v) |
| if( abs\_mvd\_greater0\_flag[ 0 ] ) |  |
| **abs\_mvd\_greater1\_flag**[ 0 ] | ae(v) |
| if( abs\_mvd\_greater0\_flag[ 1 ] ) |  |
| **abs\_mvd\_greater1\_flag**[ 1 ] | ae(v) |
| if( abs\_mvd\_greater0\_flag[ 0 ] ) { |  |
| if( abs\_mvd\_greater1\_flag[ 0 ] ) |  |
| **abs\_mvd\_minus2**[ 0 ] | ae(v) |
| **mvd\_sign\_flag**[ 0 ] | ae(v) |
| } |  |
| if( abs\_mvd\_greater0\_flag[ 1 ] ) { |  |
| if( abs\_mvd\_greater1\_flag[ 1 ] ) |  |
| **abs\_mvd\_minus2**[ 1 ] | ae(v) |
| **mvd\_sign\_flag**[ 1 ] | ae(v) |
| } |  |
| } |  |

#### Transform tree syntax

|  |  |
| --- | --- |
| transform\_tree( x0, y0, tbWidth, tbHeight , treeType) { | Descriptor |
| if( tbWidth > MaxTbSizeY | | tbHeight > MaxTbSizeY ) { |  |
| trafoWidth = ( tbWidth > MaxTbSizeY ) ? (tbWidth / 2) : tbWidth |  |
| trafoHeight = ( tbHeight > MaxTbSizeY ) ? (tbHeight / 2) : tbHeight |  |
| transform\_tree( x0, y0, trafoWidth,  trafoHeight ) |  |
| if( tbWidth > MaxTbSizeY ) |  |
| transform\_tree( x0 + trafoWidth, y0, trafoWidth, trafoHeight, treeType ) |  |
| if( tbHeight > MaxTbSizeY ) |  |
| transform\_tree( x0, y0 + trafoHeight, trafoWidth, trafoHeight, treeType ) |  |
| if( tbWidth > MaxTbSizeY && tbHeight > MaxTbSizeY ) |  |
| transform\_tree( x0 + trafoWidth, y0 + trafoHeight, trafoWidth, trafoHeight, treeType ) |  |
| } else { |  |
| transform\_unit( x0, y0, tbWidth, tbHeight, treeType ) |  |
| } |  |
| } |  |

#### Transform unit syntax

|  |  |
| --- | --- |
| transform\_unit( x0, y0, tbWidth, tbHeight, treeType ) { | Descriptor |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) |  |
| **tu\_cbf\_luma**[ x0 ][ y0 ] | ae(v) |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA ) { |  |
| **tu\_cbf\_cb**[ x0 ][ y0 ] | ae(v) |
| **tu\_cbf\_cr**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( ( ( ( CuPredMode[ x0 ][ y0 ]  = =  MODE\_INTRA )  &&  sps\_mts\_intra\_enabled\_flag ) | |  ( ( CuPredMode[ x0 ][ y0 ]  = =  MODE\_INTER )  &&  sps\_mts\_inter\_enabled\_flag ) )  &&  tu\_cbf\_luma[ x0 ][ y0 ]  &&  treeType ! = DUAL\_TREE\_CHROMA  &&  ( tbWidth  <=  32 )  &&  ( tbHeight  <=  32 ) ) |  |
| **cu\_mts\_flag**[ x0 ][ y0 ] | ae(v) |
| if( tu\_cbf\_luma[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, log2( tbWidth ), log2( tbHeight ), 0 ) |  |
| if( tu\_cbf\_cb[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, log2( tbWidth / 2 ), log2( tbHeight / 2 ), 1 ) |  |
| if( tu\_cbf\_cr[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, log2( tbWidth / 2 ), log2( tbHeight / 2 ), 2 ) |  |
| } |  |

#### Residual coding syntax

|  |  |  |
| --- | --- | --- |
| residual\_coding( x0, y0, log2TbWidth, log2TbHeight, cIdx ) { | Descriptor | |
| if( transform\_skip\_enabled\_flag && ( cIdx ! = 0  | |  cu\_mts\_flag[ x0 ][ y0 ] = = 0 ) &&   ( log2TbWidth  <=  2 ) && ( log2TbHeight  <=  2 ) ) |  | |
| **transform\_skip\_flag**[ x0 ][ y0 ][ cIdx ] | ae(v) | |
| **last\_sig\_coeff\_x\_prefix** | ae(v) |
| **last\_sig\_coeff\_y\_prefix** | ae(v) |
| if( last\_sig\_coeff\_x\_prefix > 3 ) |  |
| **last\_sig\_coeff\_x\_suffix** | ae(v) |
| if( last\_sig\_coeff\_y\_prefix > 3 ) |  |
| **last\_sig\_coeff\_y\_suffix** | ae(v) |
| log2SbSize = ( Min( log2TbWidth, log2TbHeight ) < 2 ? 1 : 2 ) |  |
| numSbCoeff = 1 << ( log2SbSize << 1 ) |  |
| lastScanPos = numSbCoeff |  |
| lastSubBlock = ( 1  <<  ( log2TbWidth + log2TbHeight − 2 \* log2SbSize ) ) − 1 |  |
| do { |  |
| if( lastScanPos = = 0 ) { |  |
| lastScanPos = numSbCoeff |  |
| lastSubBlock− − |  |
| } |  |
| lastScanPos− − |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbSize ][ log2TbHeight − log2SbSize ]  [ lastSubBlock ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbSize ][ log2TbHeight − log2SbSize ]  [ lastSubBlock ][ 1 ] |  |
| xC = ( xS << log2SbSize ) +   DiagScanOrder[ log2SbSize ][ log2SbSize ][ lastScanPos ][ 0 ] |  |
| yC = ( yS << log2SbSize ) +   DiagScanOrder[ log2SbSize ][ log2SbSize ][ lastScanPos ][ 1 ] |  |
| } while( ( xC != LastSignificantCoeffX ) | | ( yC != LastSignificantCoeffY ) ) |  |
| numSigCoeff = 0 |  |
| QState = 0 |  |
| for( i = lastSubBlock; i >= 0; i− − ) { |  |
| startQStateSb = QState |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbSize ][ log2TbHeight − log2SbSize ]  [ lastSubBlock ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbSize ][ log2TbHeight − log2SbSize ]  [ lastSubBlock ][ 1 ] |  |
| inferSbDcSigCoeffFlag = 0 |  |
| if( ( i < lastSubBlock ) && ( i > 0 ) ) { |  |
| **coded\_sub\_block\_flag**[ xS ][ yS ] | ae(v) |
| inferSbDcSigCoeffFlag = 1 |  |
| } |  |
| firstSigScanPosSb = numSbCoeff |  |
| lastSigScanPosSb = −1 |  |
| for( n = ( i = = lastSubBlock ) ? lastScanPos − 1 : numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( coded\_sub\_block\_flag[ xS ][ yS ] && ( n > 0 | | !inferSbDcSigCoeffFlag ) ) { |  |
| **sig\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| if( sig\_coeff\_flag[ xC ][ yC ] ) |  |
| inferSbDcSigCoeffFlag = 0 |  |
| } |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) { |  | |
| numSigCoeff++ |  | |
| **par\_level\_flag**[ n ] | ae(v) |
| **rem\_abs\_gt1\_flag**[ n ] | ae(v) |
| if( lastSigScanPosSb = = −1 ) |  |
| lastSigScanPosSb = n |  |
| firstSigScanPosSb = n |  |
| } |  |
| AbsLevelPass1[ xC ][ yC ] =   sig\_coeff\_flag[ xC ][ yC ] + par\_level\_flag[ n ] + 2 \* rem\_abs\_gt1\_flag[ n ] |  |
| if( dep\_quant\_enabled\_flag ) |  |
| QState = QStateTransTable[ QState ][ par\_level\_flag[ n ] ] |  |
| } |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| if( rem\_abs\_gt1\_flag[ n ] ) |  |
| **rem\_abs\_gt2\_flag**[ n ] | ae(v) |
| } |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( rem\_abs\_gt2\_flag[ n ] ) |  |
| **abs\_remainder**[ n ] |  |
| AbsLevel[ xC ][ yC ] = AbsLevelPass1[ xC ][ yC ] +  2 \* ( rem\_abs\_gt2\_flag[ n ] + abs\_remainder[ n ] ) |  |
| } |  |
| if( dep\_quant\_enabled\_flag | | !sign\_data\_hiding\_enabled\_flag ) |  |
| signHidden = 0 |  |
| else |  |
| signHidden = ( lastSigScanPosSb − firstSigScanPosSb > 3 ? 1 : 0 ) |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( sig\_coeff\_flag[ xC ][ yC ] &&   ( !signHidden | | ( n != firstSigScanPosSb ) ) ) |  |
| **coeff\_sign\_flag**[ n ] | ae(v) |
| } |  |
| if( dep\_quant\_enabled\_flag ) { |  |
| QState = startQStateSb |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbSize ) +   DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) +   DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) |  |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =  ( 2 \* AbsLevel[ xC ][ yC ] − ( QState > 1 ? 1 : 0 ) ) \*  ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| QState = QStateTransTable[ QState ][ par\_level\_flag[ n ] ] |  |
| } else { |  |
| sumAbsLevel = 0 |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbSize ) +   DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) +   DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) { |  |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   AbsLevel[ xC ][ yC ] \* ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| if( signHidden ) { |  |
| sumAbsLevel += AbsLevel[ xC ][ yC ] |  |
| if( ( n = = firstSigScanPosSb ) && ( sumAbsLevel % 2 ) = = 1 ) ) |  |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   −TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| if(  cu\_mts\_flag[ x0 ][ y0 ]  &&  ( cIdx  = =  0 ) &&  ( ( CuPredMode[ x0 ][ y0 ]  = =  MODE\_INTRA  &&  numSigCoeff  > 2 )  | |  ( CuPredMode[ x0 ][ y0 ]  = =  MODE\_INTER ) )  ) { |  |
| **mts\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  | |

## Semantics

### 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 Specification.

### NAL unit semantics

#### General NAL unit semantics

NumBytesInNalUnit specifies the size of the NAL unit in bytes. This value is required for decoding of the NAL unit. Some form of demarcation of NAL unit boundaries is necessary to enable inference of NumBytesInNalUnit. One such demarcation method is specified in Annex TBD for the byte stream format. Other methods of demarcation may be specified outside of this Specification.

NOTE 1 – The video coding layer (VCL) is specified to efficiently represent the content of the video data. The NAL is specified to format that data and provide header information in a manner appropriate for conveyance on a variety of communication channels or storage media. All data are contained in NAL units, each of which contains an integer number of bytes. A NAL unit specifies a generic format for use in both packet-oriented and bitstream systems. The format of NAL units for both packet-oriented transport and byte stream is identical except that each NAL unit can be preceded by a start code prefix and extra padding bytes in the byte stream format specified in Annex TBD.

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

The RBSP contains an string of data bits **(**SODB) as follows:

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

– Otherwise, the RBSP contains the SODB as follows:

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

2) rbsp\_trailing\_bits( ) are present after the SODB as follows:

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

ii) The next bit consists of a single rbsp\_stop\_one\_bit equal to 1.

iii) When the rbsp\_stop\_one\_bit is not the last bit of a byte-aligned byte, one or more rbsp\_alignment\_zero\_bit is present to result in byte alignment.

3) One or more cabac\_zero\_word 16-bit syntax elements equal to 0x0000 may be present in some RBSPs after the rbsp\_trailing\_bits( ) at the end of the RBSP.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "\_rbsp" suffix. These structures are carried within NAL units as the content of the rbsp\_byte[ i ] data bytes. The association of the RBSP syntax structures to the NAL units is as specified in Table 7‑1.

NOTE 2 – When the boundaries of the RBSP are known, the decoder can extract the SODB from the RBSP by concatenating the bits of the bytes of the RBSP and discarding the rbsp\_stop\_one\_bit, which is the last (least significant, right-most) bit equal to 1, and discarding any following (less significant, farther to the right) bits that follow it, which are equal to 0. The data necessary for the decoding process is contained in the SODB part of the RBSP.

**emulation\_prevention\_three\_byte** is a byte equal to 0x03. When an emulation\_prevention\_three\_byte is present in the NAL unit, it shall be discarded by the decoding process.

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

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

– 0x000000

– 0x000001

– 0x000002

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

– 0x00000300

– 0x00000301

– 0x00000302

– 0x00000303

#### NAL unit header semantics

**forbidden\_zero\_bit** shall be equal to 0.

**nal\_unit\_type** specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7‑1.

Table 7‑1 – NAL unit type codes and NAL unit type classes

|  |  |  |  |
| --- | --- | --- | --- |
| **nal\_unit\_type** | **Name of nal\_unit\_type** | **Content of NAL unit and RBSP syntax structure** | **NAL unit type class** |
| 0 | NON\_IRAP\_NUT | Coded slice segment of a non-IRAP picture  slice\_layer\_rbsp( ) | VCL |
| 1 | IRAP\_NUT | Coded slice of an IRAP picture  slice\_layer\_rbsp( ) | VCL |
| 2-15 | RSV\_VCL\_NUT | Reserved VCL NAL Units | VCL |
| 16 | SPS\_NUT | Sequence parameter set seq\_parameter\_set\_rbsp( ) | non-VCL |
| 17 | PPS\_NUT | Picture parameter set pic\_parameter\_set\_rbsp( ) | non-VCL |
| 18 | EOS\_NUT | End of sequence end\_of\_seq\_rbsp( ) | non-VCL |
| 19 | EOB\_NUT | End of bitstream end\_of\_bitstream\_rbsp( ) | non-VCL |
| 20, 21 | PREFIX\_SEI\_NUT SUFFIX\_SEI\_NUT | Supplemental enhancement information sei\_rbsp( ) | non-VCL |
| 22-26 | RSV\_NVCL | Reserved | non-VCL |
| 27-31 | UNSPEC | Unspecified | non-VCL |

**nuh\_temporal\_id\_plus1** minus 1 specifies a temporal identifier for the NAL unit. The value of nuh\_temporal\_id\_plus1 shall not be equal to 0.

The variable TemporalId is derived as follows:

TemporalId = nuh\_temporal\_id\_plus1 − 1 (7‑1)

When nal\_unit\_type is equal to IRAP\_NUT, the coded slice belongs to an IRAP picture, TemporalId shall be equal to 0.

The value of TemporalId shall be the same for all VCL NAL units of an access unit. The value of TemporalId of a coded picture or an access unit is the value of the TemporalId of the VCL NAL units of the coded picture or the access unit.

The value of TemporalId for non-VCL NAL units is constrained as follows:

– If nal\_unit\_type is equal to SPS\_NUT, TemporalId shall be equal to 0 and the TemporalId of the access unit containing the NAL unit shall be equal to 0.

– Otherwise if nal\_unit\_type is equal to EOS\_NUT or EOB\_NUT, TemporalId shall be equal to 0.

– Otherwise, TemporalId shall be greater than or equal to the TemporalId of the access unit containing the NAL unit.

NOTE – When the NAL unit is a non-VCL NAL unit, the value of TemporalId is equal to the minimum value of the TemporalId values of all access units to which the non-VCL NAL unit applies. When nal\_unit\_type is equal to PPS\_NUT, TemporalId may be greater than or equal to the TemporalId of the containing access unit, as all picture parameter sets (PPSs) may be included in the beginning of a bitstream, wherein the first coded picture has TemporalId equal to 0. When nal\_unit\_type is equal to PREFIX\_SEI\_NUT or SUFFIX\_SEI\_NUT, TemporalId may be greater than or equal to the TemporalId of the containing access unit, as an SEI NAL unit may contain information that applies to a bitstream subset that includes access units for which the TemporalId values are greater than the TemporalId of the access unit containing the SEI NAL unit.

**nuh\_reserved\_zero\_7bits** shall be equal to '0000000'. Other values of nuh\_reserved\_zero\_7bits may be specified in the future by ITU‑T | ISO/IEC. Decoders shall ignore (i.e. remove from the bitstream and discard) NAL units with values of nuh\_reserved\_zero\_7bits not equal to '0000000'.

### Raw byte sequence payloads, trailing bits and byte alignment semantics

#### Sequence parameter set RBSP semantics

**sps\_seq\_parameter\_set\_id** provides an identifier for the SPS for reference by other syntax elements. The value of sps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**chroma\_format\_idc** specifies the chroma sampling relative to the luma sampling as specified in clause 6.2. The value of chroma\_format\_idc shall be in the range of 0 to 3, inclusive.

**separate\_colour\_plane\_flag** equal to 1 specifies that the three colour components of the 4:4:4 chroma format are coded separately. separate\_colour\_plane\_flag equal to 0 specifies that the colour components are not coded separately. When separate\_colour\_plane\_flag is not present, it is inferred to be equal to 0. When separate\_colour\_plane\_flag is equal to 1, the coded picture consists of three separate components, each of which consists of coded samples of one colour plane (Y, Cb, or Cr) and uses the monochrome coding syntax. In this case, each colour plane is associated with a specific colour\_plane\_id value.

NOTE 1 – There is no dependency in decoding processes between the colour planes having different colour\_plane\_id values. For example, the decoding process of a monochrome picture with one value of colour\_plane\_id does not use any data from monochrome pictures having different values of colour\_plane\_id for inter prediction.

Depending on the value of separate\_colour\_plane\_flag, the value of the variable ChromaArrayType is assigned as follows:

– If separate\_colour\_plane\_flag is equal to 0, ChromaArrayType is set equal to chroma\_format\_idc.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), ChromaArrayType is set equal to 0.

**pic\_width\_in\_luma\_samples** specifies the width of each decoded picture in units of luma samples. pic\_width\_in\_luma\_samples shall not be equal to 0 and shall be an integer multiple of MinCbSizeY.

**pic\_height\_in\_luma\_samples** specifies the height of each decoded picture in units of luma samples. pic\_height\_in\_luma\_samples shall not be equal to 0 and shall be an integer multiple of MinCbSizeY.

**bit\_depth\_luma\_minus8** specifies the bit depth of the samples of the luma array BitDepthY and the value of the luma quantization parameter range offset QpBdOffsetY as follows:

BitDepthY = 8 + bit\_depth\_luma\_minus8 (7‑2)

QpBdOffsetY = 6 \* bit\_depth\_luma\_minus8 (7‑3)

bit\_depth\_luma\_minus8 shall be in the range of 0 to 8, inclusive.

**bit\_depth\_chroma\_minus8** specifies the bit depth of the samples of the chroma arrays BitDepthC and the value of the chroma quantization parameter range offset QpBdOffsetC as follows:

BitDepthC = 8 + bit\_depth\_chroma\_minus8 (7‑4)

QpBdOffsetC = 6 \* bit\_depth\_chroma\_minus8 (7‑5)

bit\_depth\_chroma\_minus8 shall be in the range of 0 to 8, inclusive.

**qtbtt\_dual\_tree\_intra\_flag** equal to 1 specifies that for I slices, each CTU is split into coding units with 64x64 luma samples using an implicit quadtree split and that these coding units are the root of two separate coding\_quadtree syntax structure for luma and chroma.

**log2\_ctu\_size\_minus2** plus 2 specifies the luma coding tree block size of each CTU.

The variables CtbLog2SizeY, CtbSizeY, MinCbLog2SizeY, MinCbSizeY, MinTbLog2SizeY, MaxTbLog2SizeY, PicWidthInCtbsY, PicHeightInCtbsY, PicSizeInCtbsY, PicWidthInMinCbsY, PicHeightInMinCbsY, PicSizeInMinCbsY, PicSizeInSamplesY, PicWidthInSamplesC and PicHeightInSamplesC are derived as follows:

CtbLog2SizeY = log2\_ctu\_size\_minus2 + 2 (7‑6)

CtbSizeY = 1  <<  CtbLog2SizeY (7‑7)

MinCbLog2SizeY = 2 (7‑8)

MinCbSizeY = 1  <<  MinCbLog2SizeY (7‑9)

MinTbSizeY = 4 (7‑10)

MaxTbSizeY = 64 (7‑11)

PicWidthInCtbsY = Ceil( pic\_width\_in\_luma\_samples ÷ CtbSizeY ) (7‑12)

PicHeightInCtbsY = Ceil( pic\_height\_in\_luma\_samples ÷ CtbSizeY ) (7‑13)

PicSizeInCtbsY = PicWidthInCtbsY \* PicHeightInCtbsY (7‑14)

PicWidthInMinCbsY = pic\_width\_in\_luma\_samples / MinCbSizeY (7‑15)

PicHeightInMinCbsY = pic\_height\_in\_luma\_samples / MinCbSizeY (7‑16)

PicSizeInMinCbsY = PicWidthInMinCbsY \* PicHeightInMinCbsY (7‑17)

PicSizeInSamplesY = pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples (7‑18)

PicWidthInSamplesC = pic\_width\_in\_luma\_samples / SubWidthC (7‑19)

PicHeightInSamplesC = pic\_height\_in\_luma\_samples / SubHeightC (7‑20)

[Ed. (BB): Currently the minimum CU size is fixed (4x4 luma samples and corresponding chroma samples) as well as the maximum transform size (64x64 luma samples and corresponding chroma sample size) and the minimum transform size (4x4 luma samples and corresponding chroma samples), pending further specification development.]

The variables CtbWidthC and CtbHeightC, which specify the width and height, respectively, of the array for each chroma CTB, are derived as follows:

– If chroma\_format\_idc is equal to 0 (monochrome) or separate\_colour\_plane\_flag is equal to 1, CtbWidthC and CtbHeightC are both equal to 0.

– Otherwise, CtbWidthC and CtbHeightC are derived as follows:

CtbWidthC = CtbSizeY / SubWidthC (7‑21)

CtbHeightC = CtbSizeY / SubHeightC (7‑22)

For log2BlockWidth ranging from 0 to 4 and f or log2BlockHeight ranging from 0 to 4, inclusive, the up-right diagonal scan order array initialization process as specified in clause 6.5.2 is invoked with 1  <<  log2BlockWidth and 1  <<  log2BlockHeight as input, and the output is assigned to DiagScanOrder[ log2BlockWidth ][ log2BlockHeight ].

**log2\_min\_qt\_size\_intra\_slices\_minus2** plus 2 specifies the minimum luma size of a leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 2 (I). The value of log2\_min\_qt\_size\_intra\_slices\_minus2 shall be in the range of 0 to CtbLog2SizeY − 2, inclusive.

MinQtLog2SizeIntraY = log2\_min\_qt\_size\_intra\_slices\_minus2 + 2 (7‑23)

[Ed. (BB): The leaf of a quadtree can either be a coding unit or the root of a nested multi-type tree.]

**log2\_min\_qt\_size\_inter\_slices\_minus2** plus 2 specifies the minimum luma size of a leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 0 (B) or 1 (P). The value of log2\_min\_qt\_size\_inter\_slices\_minus2 shall be in the range of 0 to CtbLog2SizeY − 2, inclusive.

MinQtLog2SizeInterY = log2\_min\_qt\_size\_inter\_slices\_minus2 + 2 (7‑24)

**max\_mtt\_hierarchy\_depth\_inter\_slices** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice\_type equal to 0 (B) or 1 (P). The value of max\_mtt\_hierarchy\_depth\_inter\_slices shall be in the range of 0 to CtbLog2SizeY − MinTbLog2SizeY, inclusive.

**max\_mtt\_hierarchy\_depth\_intra\_slices** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice\_type equal to 2 (I). The value of max\_mtt\_hierarchy\_depth\_intra\_slices shall be in the range of 0 to CtbLog2SizeY − MinTbLog2SizeY, inclusive.

**sps\_cclm\_enabled\_flag** equal to 0 specifies that the cross-component linear model intra prediction from luma component to chroma component is disabled. sps\_cclm\_enabled\_flag equal to 1 specifies that the cross-component linear model intra prediction from luma component to chroma componenent is enabled.

**sps\_alf\_enabled\_flag** equal to 0 specifies that the adaptive loop filter is disabled. sps\_alf\_enabled\_flag equal to 1 specifies that the adaptive loop filter is enabled.

**sps\_temporal\_mvp\_enabled\_flag** equal to 1 specifies that slice\_temporal\_mvp\_enabled\_flag is present in the slice headers of slices with slice\_type not equal to I in the CVS. sps\_temporal\_mvp\_enabled\_flag equal to 0 specifies that slice\_temporal\_mvp\_enabled\_flag is not present in slice headers and that temporal motion vector predictors are not used in the CVS.

**sps\_sbtmvp\_enabled\_flag** equal to 1 specifies that subblock-based temporal motion vector predictors may be used in decoding of pictures with all slices having slice\_type not equal to I in the CVS. sps\_sbtmvp\_enabled\_flag equal to 0 specifies that subblock-based temporal motion vector predictors are not used in the CVS. When sps\_sbtmvp\_enabled\_flag is not present, it is inferred to be equal to 0

**log2\_sbtmvp\_default\_size\_minus2** specifies the inferred value of the syntax element log2\_sbtmvp\_active\_size\_minus2 in the slice headers of slices with slice\_type not equal to I in the CVS when slice\_sbtmvp\_size\_override\_flag is equal to 0.

**sps\_amvr\_enabled\_flag** equal to 1 specifies that adaptive motion vector difference resolution is used in motion vector coding. amvr\_enabled\_flag equal to 0 specifies that adaptive motion vector difference resolution is not used in motion vector coding.

**sps\_affine\_enabled\_flag** specifies whether affine model based motion compensation can be used for inter prediction. If sps\_affine\_enabled\_flag is equal to 0, the syntax shall be constrained such that no affine model based motion compensation is used in the CVS, and merge\_affine\_flag, inter\_affine\_flag and cu\_affine\_type\_flag are not present in coding unit syntax of the CVS. Otherwise (sps\_affine\_enabled\_flag is equal to 1), affine model based motion compensation can be used in the CVS.

**sps\_affine\_type\_flag** specifies whether 6-parameter affine model based motion compensation can be used for inter prediction. If sps\_affine\_type\_flag is equal to 0, the syntax shall be constrained such that no 6-parameter affine model based motion compensation is used in the CVS, and cu\_affine\_type\_flag is not present in coding unit syntax in the CVS. Otherwise (sps\_affine\_type\_flag is equal to 1), 6-parameter affine model based motion compensation can be used in the CVS. When not present, the value of sps\_affine\_type\_flag is inferred to be equal to 0.

**sps\_mts\_intra\_enabled\_flag** equal to 1 specifies that cu\_mts\_flag may be present in the residual coding syntax for intra coding units. sps\_mts\_intra\_enabled\_flag equal to 0 specifies that cu\_mts\_flag is not present in the residual coding syntax for intra coding units.

**sps\_mts\_inter\_enabled\_flag** specifies that cu\_mts\_flag may be present in the residual coding syntax for inter coding units. sps\_mts\_inter\_enabled\_flag equal to 0 specifies that cu\_mts\_flag is not present in the residual coding syntax for inter coding units.

#### Picture parameter set RBSP semantics

**pps\_pic\_parameter\_set\_id** identifies the PPS for reference by other syntax elements. The value of pps\_pic\_parameter\_set\_id shall be in the range of 0 to 63, inclusive.

**pps\_seq\_parameter\_set\_id** specifies the value of sps\_seq\_parameter\_set\_id for the active SPS. The value of pps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**transform\_skip\_enabled\_flag** equal to 1 specifies that transform\_skip\_flag may be present in the residual coding syntax. transform\_skip\_enabled\_flag equal to 0 specifies that transform\_skip\_flag is not present in the residual coding syntax.

#### End of sequence RBSP semantics

When included in a NAL unit with nuh\_layer\_id equal to 0, the end of sequence RBSP specifies that the current access unit is the last access unit in the coded video sequence in decoding order and the next subsequent access unit in the bitstream in decoding order (if any) is an IRAP access unit. The syntax content of the SODB and RBSP for the end of sequence RBSP are empty.

#### End of bitstream RBSP semantics

The end of bitstream RBSP indicates that no additional NAL units are present in the bitstream that are subsequent to the end of bitstream RBSP in decoding order. The syntax content of the SODB and RBSP for the end of bitstream RBSP are empty.

#### Slice layer RBSP semantics

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

#### RBSP slice trailing bits semantics

**cabac\_zero\_word** is a byte-aligned sequence of two bytes equal to 0x0000.

Let NumBytesInVclNalUnits be the sum of the values of NumBytesInNalUnit for all VCL NAL units of a coded picture.

Let BinCountsInNalUnits be the number of times that the parsing process function DecodeBin( ), specified in clause TBD, is invoked to decode the contents of all VCL NAL units of a coded picture.

Let the variable RawMinCuBits be derived as follows:

RawMinCuBits = MinCbSizeY \* MinCbSizeY \*  
 ( BitDepthY + 2 \* BitDepthC / ( SubWidthC \* SubHeightC ) ) (7‑25)

The value of BinCountsInNalUnits shall be less than or equal to ( 32 ÷ 3 ) \* NumBytesInVclNalUnits + ( RawMinCuBits \* PicSizeInMinCbsY ) ÷ 32.

NOTE – The constraint on the maximum number of bins resulting from decoding the contents of the coded slice NAL units can be met by inserting a number of cabac\_zero\_word syntax elements to increase the value of NumBytesInVclNalUnits. Each cabac\_zero\_word is represented in a NAL unit by the three-byte sequence 0x000003 (as a result of the constraints on NAL unit contents that result in requiring inclusion of an emulation\_prevention\_three\_byte for each cabac\_zero\_word).

#### RBSP trailing bits semantics

**rbsp\_stop\_one\_bit** shall be equal to 1.

**rbsp\_alignment\_zero\_bit** shall be equal to 0.

#### Byte alignment semantics

**alignment\_bit\_equal\_to\_one** shall be equal to 1.

**alignment\_bit\_equal\_to\_zero** shall be equal to 0.

### Slice header semantics

#### General slice header semantics

When present, the value of the slice header syntax element slice\_pic\_parameter\_set\_id shall be the same in all slice headers of a coded picture.

**slice\_pic\_parameter\_set\_id** specifies the value of pps\_pic\_parameter\_set\_id for the PPS in use. The value of slice\_pic\_parameter\_set\_id shall be in the range of 0 to 63, inclusive.

**slice\_address** specifies the address of the first CTB in the slice, in CTB raster scan of a picture. The length of the slice\_address syntax element is Ceil( Log2( PicSizeInCtbsY ) ) bits. The value of slice\_address shall be in the range of 0 to PicSizeInCtbsY − 1, inclusive, and the value of slice\_address shall not be equal to the value of slice\_address of any other coded slice NAL unit of the same coded picture.

The variable CtbAddrInRs, specifying a CTB address in CTB raster scan of a picture, is set equal to slice\_address.

**slice\_type** specifies the coding type of the slice according to Table 7‑2.

Table 7‑2 – Name association to slice\_type

|  |  |
| --- | --- |
| slice\_type | Name of slice\_type |
| 0 | B (B slice) |
| 1 | P (P slice) |
| 2 | I (I slice) |

When nal\_unit\_type is equal to IRAP\_NUT, i.e., the picture is an IRAP picture, slice\_type shall be equal to 2.

**log2\_diff\_ctu\_max\_bt\_size** specifies the difference between the luma CTB size and the maximum luma size (width or height) of a coding block that can be split using a binary split. The value of log2\_diff\_ctu\_max\_bt\_size shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

When log2\_diff\_ctu\_max\_bt\_size is not present, the value of log2\_diff\_ctu\_max\_bt\_size is inferred to be equal to 2.

The variables MinQtLog2SizeY, MaxBtLog2SizeY, MinBtLog2SizeY, MaxTtLog2SizeY, MinTtLog2SizeY, MaxBtSizeY, MinBtSizeY, MaxTtSizeY, MinTtSizeY and MaxMttDepth are derived as follows:

MinQtLog2SizeY = ( slice\_type = = I ) ? MinQtLog2SizeIntraY : MinQtLog2SizeInterY (7‑26)

MaxBtLog2SizeY = CtbLog2SizeY − log2\_diff\_ctu\_max\_bt\_size (7‑27)

MinBtLog2SizeY = MinCbLog2SizeY (7‑28)

MaxTtLog2SizeY = ( slice\_type = = I ) ? 5 : 6 (7‑29)

MinTtLog2SizeY = MinCbLog2SizeY (7‑30)

MinQtSizeY = 1  <<  MinQtLog2SizeY (7‑31)

MaxBtSizeY = 1  <<  MaxBtLog2SizeY (7‑32)

MinBtSizeY = 1  <<  MinBtLog2SizeY (7‑33)

MaxTtSizeY = 1  <<  MaxTtLog2SizeY (7‑34)

MinTtSizeY = 1  <<  MinTtLog2SizeY (7‑35)

MaxMttDepth = ( slice\_type = = I ) ? max\_mtt\_hierarchy\_depth\_intra\_slices :   
 max\_mtt\_hierarchy\_depth\_inter\_slices (7‑36)

[Ed. (BB): Currently the maximum TT size is fixed (32x32 luma samples and corresponding chroma samples for I-slices and 64x64 luma samples and corresponding chroma samples for P/B-slices ) as well as the maximum BT size for I-slices (CtbLog2SizeY − 2, e.g. 32x32 luma samples and corresponding chroma samples for a CTU size of 128x128 luma samples).]

**sbtmvp\_size\_override\_flag** equal to 1 specifies that the syntax element log2\_sbtmvp\_active\_size\_minus2 is present for the current slice. sbtmvp\_size\_override\_flag equal to 0 specifies that the syntax element log2\_atmvp\_active\_size\_minus2 is not present and log2\_sbtmvp\_size\_active\_minus2 is inferred to be equal to log2\_sbtmvp\_default\_size\_minus2.

**log2\_sbtmvp\_active\_size\_minus2** plus 2 specifies the value of the subblock size that is used for deriving the motion parameters for the subblock-based TMVP of the current slice. When log2\_sbtmvp\_size\_active\_minus2 is is not present, it is inferred to be equal to log2\_sbtmvp\_default\_size\_minus2. The variable is derived as follows:

Log2SbtmvpSize  =  log2\_sbtmvp\_size\_active\_minus2 + 2 (7‑37)

**slice\_temporal\_mvp\_enabled\_flag** specifies whether temporal motion vector predictors can be used for inter prediction. If slice\_temporal\_mvp\_enabled\_flag is equal to 0, the syntax elements of the current picture shall be constrained such that no temporal motion vector predictor is used in decoding of the current picture. Otherwise (slice\_temporal\_mvp\_enabled\_flag is equal to 1), temporal motion vector predictors may be used in decoding of the current picture. When not present, the value of slice\_temporal\_mvp\_enabled\_flag is inferred to be equal to 0.

**mvd\_l1\_zero\_flag** equal to 1 indicates that the mvd\_coding( x0, y0, 1 ) syntax structure is not parsed and MvdL1[ x0 ][ y0 ][ compIdx ] is set equal to 0 for compIdx = 0..1. mvd\_l1\_zero\_flag equal to 0 indicates that the mvd\_coding( x0, y0, 1 ) syntax structure is parsed.

**collocated\_from\_l0\_flag** equal to 1 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 0. collocated\_from\_l0\_flag equal to 0 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 1. When collocated\_from\_l0\_flag is not present, it is inferred to be equal to 1.

**six\_minus\_max\_num\_merge\_cand** specifies the maximum number of merging motion vector prediction (MVP) candidates supported in the slice subtracted from 6. The maximum number of merging MVP candidates, MaxNumMergeCand is derived as follows:

MaxNumMergeCand = 6 − six\_minus\_max\_num\_merge\_cand (7‑38)

The value of MaxNumMergeCand shall be in the range of 1 to 6, inclusive.

**slice\_alf\_enabled\_flag** equal to 1 specifies that adaptive loop filter is enabled and may be applied to Y, Cb, or Cr colour component in a slice. slice\_alf\_enabled\_flag equal to 0 specifies that adaptive loop filter is disabled for all colour components in a slice.

**dep\_quant\_enabled\_flag** equal to 0 specifies that dependent quantization is disabled. dep\_quant\_enabled\_flag equal to 1 specifies that dependent quantization is enabled.

**sign\_data\_hiding\_enabled\_flag** equal to 0 specifies that sign bit hiding is disabled. sign\_data\_hiding\_enabled\_flag equal to 1 specifies that sign bit hiding is enabled. When sign\_data\_hiding\_enabled\_flag is not present, it is inferred to be equal to 0.

#### Adaptive loop filter data semantics

**alf\_chroma\_idc** equal to 0 specifies that the adaptive loop filter is not applied to Cb or Cr colour component. alf\_chroma\_idc equal to 1 indicates that the adaptive loop filter is applied to Cb. alf\_chroma\_idc equal to 2 indicates that the adaptive loop filter is applied on Cr. alf\_chroma\_idc equal to 3 indicates that the adaptive loop filter is applied for Cb and Cr colour components.

The maximum value maxVal of the truncated unary binarization tu(v) is set equal to 3.

The variable NumAlfFilters specifying the number of different adaptive loop filters is set equal to 25.

**alf\_luma\_num\_filters\_signalled\_minus1** plus 1 specifies the number of adpative loop filter classes for which luma coefficients can be signalled. The value of alf\_luma\_num\_filters\_signalled\_minus1 shall be in the range of 0 to NumAlfFilters − 1, inclusive.

The maximum value maxVal of the truncated binary binarization tb(v) is set equal to NumAlfFilters − 1.

**alf\_luma\_type\_flag** specifies the filter shape of the adaptive loop filter applied to the luma colour component. alf\_luma\_type\_flagequal to 0 specifies a 7x7 filter shape. alf\_\_luma\_type\_flagequal to 1 specifies a 5x5 filter shape.

**alf\_luma\_coeff\_delta\_idx**[ filtIdx ] specifies the indices of the signalled adaptive loop filter luma coefficient deltas for the filter class indicated by filtIdx ranging from 0 to NumAlfFilters − 1. When alf\_luma\_coeff\_delta\_idx[ filtIdx ] is not present, it is inferred to be equal to 0.

The maximum value maxVal of the truncated unary binarization tu(v) is set equal to alf\_luma\_num\_filters\_signalled\_minus1.

**alf\_luma\_coeff\_delta\_flag** equal to 1 indicates that alf\_luma\_coeff\_delta\_prediction\_flag is not signalled. alf\_luma\_coeff\_delta\_flag equal to 0 indicates that alf\_luma\_coeff\_delta\_prediction\_flag may be signalled.

**alf\_luma\_coeff\_delta\_prediction\_flag** equal to 1 specifies that the signalled luma filter coefficient deltas are predicted from the deltas of the previous luma coefficients.alf\_luma\_coeff\_delta\_prediction\_flag equal to 0 specifies that the signalled luma filter coefficient deltas are not predicted from the deltas of the previous luma coefficients. When not present, alf\_luma\_coeff\_delta\_prediction\_flag is inferred to be equal to 0.

**alf\_luma\_min\_eg\_order\_minus1** plus 1 specifies the minimum order of the exp-Golomb code for luma filter coefficient signalling. The value of alf\_luma\_min\_eg\_order\_minus1 shall be in the range of 0 to 6, inclusive.

**alf\_luma\_eg\_order\_increase\_flag**[ i ]equal to 1 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is incremented by 1.

The order expGoOrderY[ i ] of the exp-Golomb code used to decode the values of alf\_luma\_coeff\_delta\_abs[ sigFiltIdx ][ j ] is derived as follows:

expGoOrderY[ i ] = alf\_luma\_min\_eg\_order\_minus1 + 1+ alf\_luma\_eg\_order\_increase\_flag[ i ] (7‑39)

**alf\_luma\_coeff\_flag**[ sigFiltIdx ]equal 1 specifies that the coefficients of the luma filter indicated by sigFiltIdx are signalled. alf\_luma\_coeff\_flag[ sigFiltIdx ] equal to 0 specifies that all filter coefficients of the luma filter indicated by sigFiltIdx are set equal to 0.

**alf\_luma\_coeff\_delta\_abs**[ sigFiltIdx ][ j ]specifies the absolute value of the j-th coefficient delta of the signalled luma filter indicated by sigFiltIdx. When alf\_luma\_coeff\_delta\_abs[ sigFiltIdx ][ j ] is not present, it is inferred to be equal 0. It is a requirement of bitstream conformance that the values of alf\_luma\_coeff\_delta\_abs[ sigFiltIdx ][ j ] shall be in the range of 0 to 211 − 1, inclusive.

The order k of the exp-Golomb binarization uek(v) is derived as follows:

golombOrderIdxY[ ] = { 0, 0, 1, 0, 0, 1, 2, 1, 0, 0, 1, 2 } (7‑40)

k = expGoOrderY[ golombOrderIdxY[ j ] ] ( 7‑41)

**alf\_luma\_coeff\_delta\_sign**[ sigFiltIdx ][ j ]specifies the sign of the j-th luma coefficient of the filter indicated by sigFiltIdx as follows:

* If alf\_luma\_coeff\_delta\_sign[ sigFiltIdx ][ j ] is equal to 0, the corresponding luma filter coefficient has a positive value.
* Otherwise (alf\_luma\_coeff\_delta\_sign[ sigFiltIdx ][ j ]), the corresponding luma filter coefficient has a negative value.

When alf\_luma\_coeff\_delta\_sign[ sigFiltIdx ][ j ] is not present, it is inferred to be equal to 0.

**alf\_chroma\_min\_eg\_order\_minus1** plus 1 specifies the minimum order of the exp-Golomb code for chroma filter coefficient signalling. The value of alf\_chroma\_min\_eg\_order\_minus1 shall be in the range of 0 to 6, inclusive.

**alf\_chroma\_eg\_order\_increase\_flag**[ i ]equal to 1 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is incremented by 1.

The order expGoOrderC[ i ] of the exp-Golomb code used to decode the values of alf\_chroma\_coeff\_abs[ j ] is derived as follows:

expGoOrderC[ i ] = alf\_chroma\_min\_eg\_order\_minus1 + 1+ alf\_chroma\_eg\_order\_increase\_flag[ i ] (7‑42)

**alf\_chroma\_ctb\_present\_flag** equal to 1 specifies that at least one the alf\_ctb\_flag[ cIdx ] syntax elements for colour component index cIdx being equal to 1 or 2 is present in coding tree unit syntax. alf\_chroma\_ctb\_present\_flag equal to 0 specifies that the alf\_ctb\_flag[ cIdx ] syntax elements with cIdx being equal to 1 or 2 are not present in coding tree unit syntax. When alf\_chroma\_ctb\_present\_flag is not present, it is inferred to be equal to 0.

[Ed. (BB): 0 and 1 were swapped during editing -> make sure VTM is aligned.]

**alf\_chroma\_coeff\_abs**[ j ]specifies the absolute value of the j-th chroma filter coefficient. When alf\_chroma\_coeff\_abs[ j ] is not present, it is inferred to be equal 0. It is a requirement of bitstream conformance that the values of alf\_chroma\_coeff\_abs[ j ] shall be in the range of 0 to 210 − 1, inclusive.

The order k of the exp-Golomb binarization uek(v) is derived as follows:

golombOrderIdxC[ ] = { 0, 0, 1, 0, 0, 1 } (7‑43)

k = expGoOrderC[ golombOrderIdxC[ j ] ] (7‑44)

**alf\_chroma\_coeff\_sign**[ j ]specifies the sign of the j-th chroma filter coefficient as follows:

* If alf\_chroma\_coeff\_sign[ j ] is equal to 0, the corresponding chroma filter coefficient has a positive value.
* Otherwise (alf\_chroma\_coeff\_sign[ j ]), the corresponding chroma filter coefficient has a negative value.

When alf\_chroma\_coeff\_sign[ j ] is not present, it is inferred to be equal to 0.

### Slice data semantics

#### General slice data semantics

**end\_of\_slice\_flag** equal to 0 specifies that another CTU is following in the slice. end\_of\_slice\_flag equal to 1 specifies the end of the slice, i.e., that no further CTU follows in the slice.

#### Coding tree unit semantics

The CTU is the root node of the coding quadtree structure.

**alf\_ctb\_flag**[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ]equal to 1 specifies that the adaptive loop filter is applied to the coding tree block of the colour component indicated by cIdx of the coding tree unit at luma location ( xCtb, yCtb ). alf\_ctb\_flag[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] equal to 0 specifies that the adaptive loop filter is not applied to the coding tree block of the colour component indicated by cIdx of the coding tree unit at luma location ( xCtb, yCtb ).

When alf\_ctb\_flag[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is not present, it is inferred as follows

* If all of the following conditions are true, alf\_ctb\_flag[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is inferred to be equal to 1:
* alf\_chroma\_ctb\_present\_flag is equal to 0,
* cIdx is greater than 0,
* alf\_chroma\_idc is equal to cIdx or alf\_chroma\_idc is equal to 3
* Otherwise, alf\_ctb\_flag[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is inferred to be equal to 0.

#### Coding quadtree semantics

**qt\_split\_cu\_flag**[ x0 ][ y0 ] specifies whether a coding unit is split into coding units with half horizontal and vertical size. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When qt\_split\_cu\_flag[ x0 ][ y0 ] is not present, the following applies:

* If one or more of the following conditions are true, the value of qt\_split\_cu\_flag[ x0 ][ y0 ] is inferred to be equal to 1:
* x0 + ( 1  <<  log2CbSize ) is greater than pic\_width\_in\_luma\_samples and ( 1  <<  log2CbSize ) is greater than MaxBtSizeY.
* y0 + ( 1  <<  log2CbSize ) is greater than pic\_height\_in\_luma\_samples and ( 1  <<  log2CbSize ) is greater than MaxBtSizeY.
* If all of the following conditions are true, the value of qt\_split\_cu\_flag[ x0 ][ y0 ] is inferred to be equal to 1:
* x0 + ( 1  <<  log2CbSize ) is greater than pic\_width\_in\_luma\_samples.
* y0 + ( 1  <<  log2CbSize ) is greater than pic\_height\_in\_luma\_samples.
* ( 1  <<  log2CbSize ) is greater than MinQtSizeY.

– Otherwise, the value of qt\_split\_cu\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

#### Multi-type tree semantics

The variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer allowSplitTtHor are derived as follows:

* The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT\_BT\_VER, the coding block width cbWidth, the coding block height cbHeight, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, the current partition index partIdx as input, and the output is assigned to allowSplitBtVer.
* The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT\_BT\_HOR, the coding block height cbHeight, the coding block width cbWidth, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, the current partition index partIdx as input, and the output is assigned to allowSplitBtHor.
* The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT\_TT\_VER, the coding block width cbWidth, the coding block height cbHeight, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, the current partition index partIdx as input, and the output is assigned to allowSplitTtVer.
* The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT\_TT\_HOR, the coding block height cbHeight, the coding block width cbWidth, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, the current partition index partIdx as input, and the output is assigned to allowSplitTtHor.

**mtt\_split\_cu\_flag** equal to 0 specifies that a coding unit is not split. mtt\_split\_cu\_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split or into three coding units using a ternary split as indicated by the syntax element mtt\_split\_cu\_binary\_flag. The binary or ternary split can be either vertical or horizontal as indicated by the syntax element mtt\_split\_cu\_vertical\_flag.

When mtt\_split\_cu\_flag is not present, the value of mtt\_split\_cu\_flag is inferred as follows:

* If one or more of the following conditions are true, the value of mtt\_split\_cu\_flag is inferred to be equal to 1:
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples.
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples.
* Otherwise, the value of mtt\_split\_cu\_flag is inferred to be equal to 0.

**mtt\_split\_cu\_vertical\_flag** equal to 0 specifies that a coding unit is split horizontally. mtt\_split\_cu\_vertical\_flag equal to 1 specifies that a coding unit is split vertically

When mtt\_split\_cu\_vertical\_flag is not present, it is inferred as follows:

* If allowSplitBtHor is equal to TRUE or allowSplitTtHor is equal to TRUE, the value of mtt\_split\_cu\_vertical\_flag is inferred to be equal to 0.
* Otherwise, the value of mtt\_split\_cu\_vertical\_flag is inferred to be equal to 1

**mtt\_split\_cu\_binary\_flag** equal to 0 specifies that a coding unit is split into three coding units using a ternary split. mtt\_split\_cu\_binary\_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split.

When mtt\_split\_cu\_binary\_flag is not present, it is inferred as follows:

* If allowSplitBtVer is equal to FALSE and allowSplitBtHor is equal to FALSE, the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to 0.
* Otherwise if allowSplitTtVer is equal to FALSE and allowSplitTtHor is equal to FALSE, the value of mtt\_split\_cu\_binary\_flag is inferred as to be equal to 1.
* Otherwise if allowSplitBtHor is equal to TRUE and allowSplitTtVer is equal to TRUE, the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to !mtt\_split\_cu\_vertical\_flag.
* Otherwise (allowSplitBtVer is equal to TRUE and allowSplitTtHor is equal to TRUE), the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to mtt\_split\_cu\_vertical\_flag.

The variable MttSplitMode[ x ][ y ][ mttDepth ] is derived from the value of mtt\_split\_cu\_vertical\_flag and from the value of mtt\_split\_cu\_binary\_flag as defined in Table 7‑3 for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.



Figure 7‑1 – Multi-type tree spliting modes indicated by MttSplitMode (informative)

MttSplitMode[ x0 ][ y0 ][ mttDepth ] represents horizontal and vertical binary and ternary splittings of a coding unit within the multi-type tree as illustrated in Figure 7‑1. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7‑3 – Specification of MttSplitMode[ x ][ y ][ mttDepth ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1

|  |  |  |
| --- | --- | --- |
| **MttSplitMode[ x0 ][ y0 ][ mttDepth ]** | **mtt\_split\_cu\_vertical\_flag** | **mtt\_split\_cu\_binary\_flag** |
| SPLIT\_TT\_HOR | 0 | 0 |
| SPLIT\_BT\_HOR | 0 | 1 |
| SPLIT\_TT\_VER | 1 | 0 |
| SPLIT\_BT\_VER | 1 | 1 |

#### Coding unit semantics

The following assignments are made for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

CbPosX[ x ][ y ] = x0 (7‑45)

CbPosY[ x ][ y ] = y0 (7‑46)

CbWidth[ x ][ y ] = cbWidth (7‑47)

CbHeight[ x ][ y ] = cbHeight (7‑48)

**cu\_skip\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, no more syntax elements except the merging candidate index merge\_idx[ x0 ][ y0 ] and the merge affine flag merge\_affine\_flag[ x0 ][ y0 ] are parsed after cu\_skip\_flag[ x0 ][ y0 ]. cu\_skip\_flag[ x0 ][ y0 ] equal to 0 specifies that the coding unit is not skipped. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When cu\_skip\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**pred\_mode\_flag** equal to 0 specifies that the current coding unit is coded in inter prediction mode. pred\_mode\_flag equal to 1 specifies that the current coding unit is coded in intra prediction mode. The variable CuPredMode[ x ][ y ] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* If pred\_mode\_flag is equal to 0, CuPredMode[ x ][ y ] is set equal to MODE\_INTER.
* Otherwise (pred\_mode\_flag is equal to 1), CuPredMode[ x ][ y ] is set equal to MODE\_INTRA.

When pred\_mode\_flag is not present, the variable CuPredMode[ x ][ y ] is inferred to be equal to MODE\_INTRA for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

The syntax elements **intra\_luma\_mpm\_flag**[ x0 ][ y0 ], **intra\_luma\_mpm\_idx**[ x0 ][ y0 ] and **intra\_luma\_mpm\_remainder**[ x0 ][ y0 ] specify the intra prediction mode for luma samples. The array indices x0, y0 specify the location ( x0 , y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When intra\_luma\_mpm\_flag[ x0 ][ y0 ] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted coding unit according to clause 8.2.2.

**intra\_chroma\_pred\_mode**[ x0 ][ y0 ] specifies the intra prediction mode for chroma samples. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

**mvp\_l0\_flag**[ x0 ][ y0 ] specifies the motion vector predictor index of list 0 where x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When mvp\_l0\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**mvp\_l1\_flag**[ x0 ][ y0 ] has the same semantics as mvp\_l0\_flag, with l0 and list 0 replaced by l1 and list 1, respectively.

**merge\_flag**[ x0 ][ y0 ] specifies whether the inter prediction parameters for the current coding unit are inferred from a neighbouring inter-predicted partition. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When merge\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If cu\_skip\_flag[ x0 ][ y0 ] is equal to 1, merge\_flag[ x0 ][ y0 ] is inferred to be equal to 1.
* Otherwise, merge\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**merge\_idx**[ x0 ][ y0 ] specifies the merging candidate index of the merging candidate list where x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When merge\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**inter\_pred\_idc**[ x0 ][ y0 ] specifies whether list0, list1, or bi-prediction is used for the current coding unit according to Table 7‑4. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7‑4 – Name association to inter prediction mode

|  |  |  |
| --- | --- | --- |
| **inter\_pred\_idc** | **Name of inter\_pred\_idc** | |
| ( cbWidth + cbHeight )  !=  12 | ( cbWidth + cbHeight )  = =  12 |
| 0 | PRED\_L0 | PRED\_L0 |
| 1 | PRED\_L1 | PRED\_L1 |
| 2 | PRED\_BI | n.a. |

When inter\_pred\_idc[ x0 ][ y0 ] is not present, it is inferred to be equal to PRED\_L0.

**ref\_idx\_l0**[ x0 ][ y0 ] specifies the list 0 reference picture index for the current coding unit. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When ref\_idx\_l0[ x0 ][ y0 ] is not present it is inferred to be equal to 0.

**ref\_idx\_l1**[ x0 ][ y0 ] has the same semantics as ref\_idx\_l0, with l0 and list 0 replaced by l1 and list 1, respectively.

**merge\_affine\_flag**[ x0 ][ y0 ] specifies whether the affine prediction parameters for the current coding unit are inferred from a neighbouring block with affine model based motion compensation. When merge\_affine\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**inter\_affine\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, affine model based motion compensation is used to generate the prediction samples of the current coding unit. inter\_affine\_flag[ x0 ][ y0 ] equal to 0 specifies that the coding unit is not predicted by affine model based motion compensation. When inter\_affine\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**cu\_affine\_type\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, 6-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit. cu\_affine\_type\_flag[ x0 ][ y0 ] equal to 0 specifies that 4-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit.

MotionModelIdc[ x ][ y ] represents motion model of a coding unit as illustrated in Table 7‑5. The array indices x, y specify the luma sample location ( x, y ) relative to the top-left luma sample of the picture.

The variable MotionModelIdc[ x ][ y ] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* If merge\_flag[ x0 ][ y0 ] is equal to 1, the following applies:

MotionModelIdc[ x ][ y ] = merge\_affine\_flag[ x0 ][ y0 ] (7‑49)

* Otherwise (merge\_flag[ x0 ][ y0 ] is equal to 0), the following applies:

MotionModelIdc[ x ][ y ] = inter\_affine\_flag[ x0 ][ y0 ] + cu\_affine\_type\_flag[ x0 ][ y0 ] (7‑50)

Table 7‑5 – Interpretation of MotionModelIdc[ x0 ][ y0 ]

|  |  |
| --- | --- |
| MotionModelIdc[ x ][ y ] | **Motion model for motion compensation** |
| 0 | Translational motion |
| 1 | 4-parameter affine motion |
| 2 | 6-parameter affine motion |

**amvr\_mode**[ x0 ][ y0 ] specifies the resolution of motion vector difference. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When amvr\_mode[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The variable MvShift is set equal to amvr\_mode[ x0 ][ y0 ] << 1 and the variables MvdL0[ x0 ][ y0 ][ 0 ], MvdL0[ x0 ][ y0 ][ 1 ], MvdL1[ x0 ][ y0 ][ 0 ], MvdL1[ x0 ][ y0 ][ 1 ] are modified as follows:

MvdL0[ x0 ][ y0 ][ 0 ] = MvdL0[ x0 ][ y0 ][ 0 ] << ( MvShift + 2 ) (7‑51)

MvdL0[ x0 ][ y0 ][ 1 ] = MvdL0[ x0 ][ y0 ][ 1 ] << ( MvShift + 2 ) (7‑52)

MvdL1[ x0 ][ y0 ][ 0 ] = MvdL1[ x0 ][ y0 ][ 0 ] << ( MvShift + 2 ) (7‑53)

MvdL1[ x0 ][ y0 ][ 1 ] = MvdL1[ x0 ][ y0 ][ 1 ] << ( MvShift + 2 ) (7‑54)

**cu\_cbf** equal to 1 specifies that the transform\_tree( ) syntax structure is present for the current coding unit. cu\_cbf equal to 0 specifies that the transform\_tree( ) syntax structure is not present for the current coding unit.

When cu\_cbf is not present, it is inferred as follows:

* If cu\_skip\_flag[ x0 ][ y0 ] is equal to 1, cu\_cbf is inferred to be equal to 0.
* Otherwise, cu\_cbf is inferred to be equal to 1.

#### Motion vector difference semantics

**abs\_mvd\_greater0\_flag**[ compIdx ] specifies whether the absolute value of a motion vector component difference is greater than 0.

**abs\_mvd\_greater1\_flag**[ compIdx ] specifies whether the absolute value of a motion vector component difference is greater than 1.

When abs\_mvd\_greater1\_flag[ compIdx ] is not present, it is inferred to be equal to 0.

**abs\_mvd\_minus2**[ compIdx ] plus 2 specifies the absolute value of a motion vector component difference.

When abs\_mvd\_minus2[ compIdx ] is not present, it is inferred to be equal to −1.

**mvd\_sign\_flag**[ compIdx ] specifies the sign of a motion vector component difference as follows:

* If mvd\_sign\_flag[ compIdx ] is equal to 0, the corresponding motion vector component difference has a positive value.
* Otherwise (mvd\_sign\_flag[ compIdx ] is equal to 1), the corresponding motion vector component difference has a negative value.

When mvd\_sign\_flag[ compIdx ] is not present, it is inferred to be equal to 0.

The motion vector difference lMvd[ compIdx ] for compIdx = 0..1 is derived as follows:

lMvd[ compIdx ] = abs\_mvd\_greater0\_flag[ compIdx ] \*  
 ( abs\_mvd\_minus2[ compIdx ] + 2 ) \* ( 1 − 2 \* mvd\_sign\_flag[ compIdx ] ) (7‑55)

The value of lMvd[ compIdx ] shall be in the range of −215 to 215 − 1, inclusive.

Depending in the value of MotionModelIdc[ x ][ y ], motion vector differences are dervied as follows:

* If MotionModelIdc[ x ][ y ] is equal to 0, the variable MvdLX[ x0 ][ y0 ][ compIdx ], with X being 0 or 1, specifies the difference between a list X vector component to be used and its prediction. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.
* If refList is equal to 0, MvdL0[ x0 ][ y0 ][ compIdx ] is set equal to lMvd[ compIdx ] for compIdx = 0..1.
* Otherwise (refList is equal to 1), MvdL1[ x0 ][ y0 ][ compIdx ] is set equal to lMvd[ compIdx ] for compIdx = 0..1.
* Otherwise (MotionModelIdc[ x ][ y ] is not equal to 0), the variable MvdCpLX[ x0 ][ y0 ][ cpIdx ][ compIdx ], with X being 0 or 1, specifies the difference in 1/16 fractional-sample accuracy between a list X vector component to be used and its prediction. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, the array index cpIdx specifies the control point index. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.
* If refList is equal to 0, MvdCpL0[ x0 ][ y0 ][ cpIdx ][ compIdx ] is set equal to ( lMvd[ compIdx ] << 2) for compIdx = 0..1.
* Otherwise (refList is equal to 1), MvdCpL1[ x0 ][ y0 ][ cpIdx ][ compIdx ] is set equal to ( lMvd[ compIdx ] << 2) for compIdx = 0..1.

#### Transform tree semantics

[Ed. (BB): The transform scheme does not have any syntax for spliting a CU into TUs. However, if the height or width of a CU is larger than the current maximum transform length of 64 luma samples or the corresponding chroma sample length, the CU will be implicitly split to divide it into TUs.]

#### Transform unit semantics

The transform coefficient levels are represented by the arrays TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ]. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for Y, 1 for Cb, and 2 for Cr. The array indices xC and yC specify the transform coefficient location ( xC, yC ) within the current transform block. When the value of TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] is not specified in clause 7.3.4.9, it is inferred to be equal to 0.

**tu\_cbf\_luma**[ x0 ][ y0 ] equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

When tu\_cbf\_luma[ x0 ][ y0 ] is not present, its value is inferred to be equal to 0.

**tu\_cbf\_cb**[ x0 ][ y0 ] equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location ( x0, y0 ) of the considered transform block.

When tu\_cbf\_cb[ x0 ][ y0 ] is not present, its value is inferred to be equal to 0.

**tu\_cbf\_cr**[ x0 ][ y0 ] equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location ( x0, y0 ) of the considered transform block.

When tu\_cbf\_cr[ x0 ][ y0 ] is not present, its value is inferred to be equal to 0.

**cu\_mts\_flag**[ x0 ][ y0 ] equal to 1 specifies that multiple transform selection is applied to the residual samples of the associated luma transform block. cu\_mts\_flag[ x0 ][ y0 ] equal to 0 specifies that multiple transform selection is not applied to the residual samples of the associated luma transform block. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

When cu\_mts\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

#### Residual coding semantics

The array AbsLevel[ xC ][ yC ] represents an array of absolute values of transform coefficient levels for the current transform block and the array AbsLevelPass1[ xC ][ yC ] represents an array of partially reconstructed absolute values of transform coefficient levels for the current transform block. The array indices xC and yC specify the transform coefficient location ( xC, yC ) within the current transform block. When the value of AbsLevel[ xC ][ yC ] is not specified in clause 7.3.4.9, it is inferred to be equal to 0. When the value of AbsLevelPass1[ xC ][ yC ] is not specified in clause 7.3.4.9, it is inferred to be equal to 0.

The variables CoeffMin and CoeffMax specifying the minimum and maximum transform coefficient values are derived as follows:

CoeffMin = −( 1 << 15 ) (7‑56)

CoeffMax = ( 1 << 15 ) − 1 (7‑57)

The array QStateTransTable[ ][ ] is specified as follows:

QStateTransTable[ ][ ] = { { 0, 2 }, { 2, 0 }, { 1, 3 }, { 3, 1 } } (7‑58)

**transform\_skip\_flag**[ x0 ][ y0 ][ cIdx ] specifies whether a transform is applied to the associated transform block or not: The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb and equal to 2 for Cr. transform\_skip\_flag[ x0 ][ y0 ][ cIdx ] equal to 1 specifies that no transform is applied to the current transform block. transform\_skip\_flag[ x0 ][ y0 ][ cIdx ] equal to 0 specifies that the decision whether transform is applied to the current transform block or not depends on other syntax elements. When transform\_skip\_flag[ x0 ][ y0 ][ cIdx ] is not present, it is inferred to be equal to 0.

**last\_sig\_coeff\_x\_prefix** specifies the prefix of the column position of the last significant coefficient in scanning order within a transform block. The values of last\_sig\_coeff\_x\_prefix shall be in the range of 0 to ( log2TbWidth  <<  1 ) − 1, inclusive.

**last\_sig\_coeff\_y\_prefix** specifies the prefix of the row position of the last significant coefficient in scanning order within a transform block. The values of last\_sig\_coeff\_y\_prefix shall be in the range of 0 to ( log2TbHeight  <<  1 ) − 1, inclusive.

**last\_sig\_coeff\_x\_suffix** specifies the suffix of the column position of the last significant coefficient in scanning order within a transform block. The values of last\_sig\_coeff\_x\_suffix shall be in the range of 0 to ( 1  <<  ( ( last\_sig\_coeff\_x\_prefix  >>  1 ) − 1 ) ) − 1, inclusive.

The column position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffX is derived as follows:

* If last\_sig\_coeff\_x\_suffix is not present, the following applies:

LastSignificantCoeffX = last\_sig\_coeff\_x\_prefix (7‑59)

* Otherwise (last\_sig\_coeff\_x\_suffix is present), the following applies:

LastSignificantCoeffX = ( 1  <<  ( (last\_sig\_coeff\_x\_prefix  >>  1 ) − 1 ) ) \* (7‑60)  
 ( 2 + (last\_sig\_coeff\_x\_prefix & 1 ) ) + last\_sig\_coeff\_x\_suffix

**last\_sig\_coeff\_y\_suffix** specifies the suffix of the row position of the last significant coefficient in scanning order within a transform block. The values of last\_sig\_coeff\_y\_suffix shall be in the range of 0 to ( 1  <<  ( ( last\_sig\_coeff\_y\_prefix  >>  1 ) − 1 ) ) − 1, inclusive.

The row position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffY is derived as follows:

* If last\_sig\_coeff\_y\_suffix is not present, the following applies:

LastSignificantCoeffY = last\_sig\_coeff\_y\_prefix (7‑61)

* Otherwise (last\_sig\_coeff\_y\_suffix is present), the following applies:

LastSignificantCoeffY = ( 1  <<  ( ( last\_sig\_coeff\_y\_prefix  >>  1 ) − 1 ) ) \* (7‑62)  
 ( 2 + ( last\_sig\_coeff\_y\_prefix & 1 ) ) + last\_sig\_coeff\_y\_suffix

**coded\_sub\_block\_flag**[ xS ][ yS ] specifies the following for the sub-block at location ( xS, yS ) within the current transform block, where a sub-block is a (4x4) array of 16 transform coefficient levels:

* If coded\_sub\_block\_flag[ xS ][ yS ] is equal to 0, the 16 transform coefficient levels of the sub-block at location ( xS, yS ) are inferred to be equal to 0.
* Otherwise (coded\_sub\_block\_flag[ xS ][ yS ] is equal to 1), the following applies:
* If ( xS, yS ) is equal to ( 0, 0 ) and ( LastSignificantCoeffX, LastSignificantCoeffY ) is not equal to ( 0, 0 ), at least one of the 16 sig\_coeff\_flag syntax elements is present for the sub-block at location ( xS, yS ).
* Otherwise, at least one of the 16 transform coefficient levels of the sub-block at location ( xS, yS ) has a non-zero value.

When coded\_sub\_block\_flag[ xS ][ yS ] is not present, it is inferred as follows:

* If one or more of the following conditions are true, coded\_sub\_block\_flag[ xS ][ yS ] is inferred to be equal to 1:
* ( xS, yS ) is equal to ( 0, 0 ).
* ( xS, yS ) is equal to ( LastSignificantCoeffX  >>  2, LastSignificantCoeffY  >>  2 ).
* Otherwise, coded\_sub\_block\_flag[ xS ][ yS ] is inferred to be equal to 0.

**sig\_coeff\_flag**[ xC ][ yC ] specifies for the transform coefficient location ( xC, yC ) within the current transform block whether the corresponding transform coefficient level at the location ( xC, yC ) is non-zero as follows:

* If sig\_coeff\_flag[ xC ][ yC ] is equal to 0, the transform coefficient level at the location ( xC, yC ) is set equal to 0.
* Otherwise (sig\_coeff\_flag[ xC ][ yC ] is equal to 1), the transform coefficient level at the location ( xC, yC ) has a non‑zero value.

When sig\_coeff\_flag[ xC ][ yC ] is not present, it is inferred as follows:

* If ( xC, yC ) is the last significant location ( LastSignificantCoeffX, LastSignificantCoeffY ) in scan order or all of the following conditions are truet[ xC ][ yC ] is inferred to be equal to 1:
* ( xC & 3, yC & 3 ) is equal to ( 0, 0 ).
* inferSbDcSigCoeffFlag is equal to 1.
* coded\_sub\_block\_flag[ xS ][ yS ] is equal to 1.
* Otherwise, sig\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 0.

**par\_level\_flag**[ n ] specifies the parity of the transform coefficient level at scanning position n. When par\_level\_flag[ n ] is not present, it is inferred to be equal to 0.

**rem\_abs\_gt1\_flag**[ n ] specifies whether the syntax element rem\_abs\_gt2\_flag[ n ] is present for the scanning position n. When rem\_abs\_gt1\_flag[ n ] is not present, it is inferred to be equal to 0.

**rem\_abs\_gt2\_flag**[ n ] specifies whether the syntax element abs\_remainder[ n ] is present for the scanning position n. When rem\_abs\_gt2\_flag[ n ] is not present, it is inferred to be equal to 0.

**abs\_remainder**[ n ] is the remaining absolute value of a transform coefficient level that is coded with Golomb-Rice code at the scanning position n. When abs\_remainder[ n ] is not present, it is inferred to be equal to 0.

It is a requirement of bitstream conformance that the value of abs\_remainder[ n ] shall be constrained such that the corresponding value of TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] is in the range of CoeffMin to CoeffMax, inclusive.

**coeff\_sign\_flag**[ n ] specifies the sign of a transform coefficient level for the scanning position n as follows:

* If coeff\_sign\_flag[ n ] is equal to 0, the corresponding transform coefficient level has a positive value.
* Otherwise (coeff\_sign\_flag[ n ] is equal to 1), the corresponding transform coefficient level has a negative value.

When coeff\_sign\_flag[ n ] is not present, it is inferred to be equal to 0.

**mts\_idx**[ x0 ][ y0 ] specifies which transform kernels are applied to the luma residual samples along the horizontal and vertical direction of the current transform block. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

When mts\_idx[ x0 ][ y0 ] is not present, it is inferred as follows:

* If cu\_mts\_flag[ x0 ][ y0 ] is equal to 0 or cIdx is greater than 0, mts\_idx[ x0 ][ y0 ] is inferred to be equal to −1.
* Otherwise, mts\_idx[ x0 ][ y0 ] is inferred to be equal to 0.

# Decoding process

## General decoding process

## Decoding process for coding units coded in intra prediction mode

### General decoding process for coding units coded in intra prediction mode

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a variable treeType specifying whether a single or a dual tree is used and if a dual tree is used, it specifies whether the current tree corresponds to the luma or chroma components.

Output of this process is a modified reconstructed picture before in-loop filtering.

If treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_LUMA, the decoding process for luma samples is specified as follows:

1. The derivation process for the luma intra prediction mode as specified in clause 8.2.2 is invoked with the luma location ( xCb, yCb ), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.
2. The general decoding process for intra blocks as specified in clause 8.2.4.1 is invoked with the luma location ( xCb, yCb ), the tree type treeType, the variable nTbW set equal to cbWidth, the variable nTbH set equal to cbHeight, the variable predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the variable cIdx set equal to 0 as inputs, and the output is a modified reconstructed picture before in-loop filtering.

If treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the decoding process for chroma samples is specified as follows:

1. The derivation process for the chroma intra prediction mode as specified in clause 8.2.3 is invoked with the luma location ( xCb, yCb ) as input.
2. The general decoding process for intra blocks as specified in clause 8.2.4.1 is invoked with the chroma location ( xCb / 2, yCb / 2 ), the tree type treeType, the variable nTbW set equal to ( cbWidth / 2 ), the variable nTbH set equal to ( cbHeight / 2 ), the variable predModeIntra set equal to IntraPredModeC[ xCb ][ yCb ], and the variable cIdx set equal to 1, and the output is a modified reconstructed picture before in-loop filtering.
3. The general decoding process for intra blocks as specified in clause 8.2.4.1 is invoked with the chroma location ( xCb / 2 , yCb / 2 ), the tree type treeType, the variable nTbW set equal to ( cbWidth / 2 ), the variable nTbH set equal to ( cbHeight / 2 ), the variable predModeIntra set equal to IntraPredModeC[ xCb ][ yCb ], and the variable cIdx set equal to 2, and the output is a modified reconstructed picture before in-loop filtering.

### Derivation process for luma intra prediction mode

Input to this process are:

* a luma location ( xCb , yCb ) specifying the top-left sample of the current luma coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

In this process, the luma intra prediction mode IntraPredModeY[ xCb ][ yCb ] is derived.

Table 8‑1 specifies the value for the intra prediction mode IntraPredModeY[ xCb ][ yCb ] and the associated names.

Table 8‑1 – Specification of intra prediction mode and associated names

|  |  |
| --- | --- |
| **Intra prediction mode** | **Associated name** |
| 0 | INTRA\_PLANAR |
| 1 | INTRA\_DC |
| 2..66 | INTRA\_ANGULAR2..INTRA\_ANGULAR66 |
| 77 | INTRA\_CCLM |

NOTE – : The intra prediction mode INTRA\_CCLM is only applicable to chroma components.

IntraPredModeY[ xCb ][ yCb ] is derived by the following ordered steps:

1. The neighbouring locations ( xNbA, yNbA ) and ( xNbB, yNbB ) are set equal to ( xCb − 1, yCb ) and ( xCb, yCb − 1 ), respectively.
2. For X being replaced by either A or B, the variables candIntraPredModeX are derived as follows:

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbX, yNbX ) as inputs, and the output is assigned to availableX.
* The candidate intra prediction mode candIntraPredModeX is derived as follows:
* If one or more of the following conditions are true, candIntraPredModeX is set equal to INTRA\_DC.
* The variable availableX is equal to FALSE.
* CuPredMode[ xNbX ][ yNbX ] is not equal to MODE\_INTRA.
* X is equal to B and yCb − 1 is less than ( ( yCb  >>  CtbLog2SizeY )  <<  CtbLog2SizeY ).
* Otherwise, candIntraPredModeX is set equal to IntraPredModeY[ xNbX ][ yNbX ].

1. The candModeList[ x ] with x = 0..2 is derived as follows:

* If candIntraPredModeB is equal to candIntraPredModeA, the following applies:
  + If candIntraPredModeA is less than 2 (i.e., equal to INTRA\_PLANAR or INTRA\_DC), candModeList[ x ] with x = 0..2 is derived as follows:

candModeList[ 0 ] = INTRA\_PLANAR (8‑1)

candModeList[ 1 ] = INTRA\_DC (8‑2)

candModeList[ 2 ] = INTRA\_ANGULAR50 (8‑3)

* + Otherwise, candModeList[ x ] with x = 0..2 is derived as follows:

candModeList[ 0 ] = candIntraPredModeA (8‑4)

candModeList[ 1 ] = 2 + ( ( candIntraPredModeA + 61 ) % 64 ) (8‑5)

candModeList[ 2 ] = 2 + ( ( candIntraPredModeA − 1 ) % 64 ) (8‑6)

* Otherwise (candIntraPredModeB is not equal to candIntraPredModeA), the following applies:
  + candModeList[ 0 ] and candModeList[ 1 ] are derived as follows:

candModeList[ 0 ] = candIntraPredModeA (8‑7)

candModeList[ 1 ] = candIntraPredModeB (8‑8)

* + If neither of candModeList[ 0 ] and candModeList[ 1 ] is equal to INTRA\_PLANAR, candModeList[ 2 ] is set equal to INTRA\_PLANAR,
  + Otherwise, if neither of candModeList[ 0 ] and candModeList[ 1 ] is equal to INTRA\_DC, candModeList[ 2 ] is set equal to INTRA\_DC,
  + Otherwise, candModeList[ 2 ] is set equal to INTRA\_ANGULAR50.

1. IntraPredModeY[ xCb ][ yCb ] is derived by applying the following procedure:

* If intra\_luma\_mpm\_flag[ xCb ][ yCb ] is equal to 1, the IntraPredModeY[ xCb ][ yCb ] is set equal to candModeList[ intra\_luma\_mpm\_idx[ xCb ][ yCb ] ].
* Otherwise, IntraPredModeY[ xCb ][ yCb ] is derived by applying the following ordered steps:

1. The array candModeList[ x ], x = 0..2 is modified by the following ordered steps:
   1. When candModeList[ 0 ] is greater than candModeList[ 1 ], both values are swapped as follows:

( candModeList[ 0 ], candModeList[ 1 ] ) = Swap( candModeList[ 0 ], candModeList[ 1 ] ) (8‑9)

* 1. When candModeList[ 0 ] is greater than candModeList[ 2 ], both values are swapped as follows:

( candModeList[ 0 ], candModeList[ 2 ] ) = Swap( candModeList[ 0 ], candModeList[ 2 ] ) (8‑10)

* 1. When candModeList[ 1 ] is greater than candModeList[ 2 ], both values are swapped as follows:

( candModeList[ 1 ], candModeList[ 2 ] ) = Swap( candModeList[ 1 ], candModeList[ 2 ] ) (8‑11)

1. IntraPredModeY[ xCb ][ yCb ] is derived by the following ordered steps:
   1. IntraPredModeY[ xCb ][ yCb ] is set equal to intra\_luma\_mpm\_remainder[ xCb ][ yCb ].
   2. For i equal to 0 to 2, inclusive, when IntraPredModeY[ xCb ][ yCb ] is greater than or equal to candModeList[ i ], the value of IntraPredModeY[ xCb ][ yCb ] is incremented by one.

The variable IntraPredModeY[ x ][ y ] with x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1 is set to be equal to IntraPredModeY[ xCb ][ yCb ].

### Derivation process for chroma intra prediction mode

Input to this process is a luma location ( xCb, yCb ) specifying the top-left sample of the current chroma coding block relative to the top‑left luma sample of the current picture.

In this process, the chroma intra prediction mode IntraPredModeC[ xCb ][ yCb ] is derived.

The chroma intra prediction mode IntraPredModeC[ xCb ][ yCb ] is derived using intra\_chroma\_pred\_mode[ xCb ][ yCb ] and IntraPredModeY[ xCb ][ yCb ] as specified in Table 8‑2 – and Table 8‑3.

Table 8‑2 – Specification of IntraPredModeC[ xCb ][ yCb ] depending on intra\_chroma\_pred\_mode[ xCb ][ yCb ] and IntraPredModeY[ xCb ][ yCb ] when sps\_cclm\_enabled\_flag is equal to 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| intra\_chroma\_pred\_mode[ xCb ][ yCb ] | IntraPredModeY[ xCb ][ yCb ] | | | | |
| 0 | 50 | 18 | 1 | X ( 0  <=  X  <=  66 ) |
| 0 | 66 | 0 | 0 | 0 | 0 |
| 1 | 50 | 66 | 50 | 50 | 50 |
| 2 | 18 | 18 | 66 | 18 | 18 |
| 3 | 1 | 1 | 1 | 66 | 1 |
| 4 | 0 | 50 | 18 | 1 | X |

Table 8‑3 – Specification of IntraPredModeC[ xCb ][ yCb ] depending on intra\_chroma\_pred\_mode[ xCb ][ yCb ] and IntraPredModeY[ xCb ][ yCb ] when sps\_cclm\_enabled\_flag is equal to 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| intra\_chroma\_pred\_mode[ xCb ][ yCb ] | IntraPredModeY[ xCb ][ yCb ] | | | | |
| 0 | 50 | 18 | 1 | X ( 0  <=  X  <=  66 ) |
| 0 | 66 | 0 | 0 | 0 | 0 |
| 1 | 50 | 66 | 50 | 50 | 50 |
| 2 | 18 | 18 | 66 | 18 | 18 |
| 3 | 1 | 1 | 1 | 66 | 1 |
| 4 | 77 | 77 | 77 | 77 | 77 |
| 5 | 0 | 50 | 18 | 1 | X |

### Decoding process for intra blocks

#### General decoding process for intra blocks

Inputs to this process are:

* a sample location ( xTb0, yTb0 ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable nTbW specifying the width of the current transform block,
* a variable nTbH specifying the height of the current transform block,
* a variable predModeIntra specifying the intra prediction mode,
* a variable cIdx specifying the colour component of the current block.

Output of this process is a modified reconstructed picture before in-loop filtering.

The maximum transform block size maxTbSize is derived as follows:

maxTbSize = ( cIdx  = =  0 ) ? MaxTbSizeY : MaxTbSizeY / 2 (8‑12)

The luma sample location is derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTb0, yTb0 ) : ( xTb0 \* 2, yTb0 \* 2 ) (8‑13)

Depending on maxTbSize, the following applies:

* If nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables nTbW and nTbH are modified as follows:

nTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑14)

nTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑15)

1. The general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
2. If nTbW is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + nTbW, yTb0 ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
3. If nTbH is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0, yTb0 + nTbH ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
4. If nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + nTbW, yTb0 + nTbH ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

* Otherwise, the following ordered steps apply:

1. The general intra sample prediction process as specified in clause 8.2.4.2.1 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xTb0, yTb0 ), the intra prediction mode predModeIntra, the transform block width nTbW and height nTbH, and the variable cIdx as inputs, and the output is an (nTbW)x(nTbH) array predSamples.
2. The scaling and transformation process as specified in clause 8.4.2 is invoked with the luma location ( xTbY, yTbY ), the variable cIdx, the transform width nTbW and the transform height nTbH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.
3. The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the transform block location ( xTbComp, yTbComp ) set equal to ( xTb0, yTb0 ), the transform block width nTbW, the transform block height nTbH, the variable cIdx, the (nTbW)x(nTbH) array predSamples, and the (nTbW)x(nTbH) array resSamples as inputs, and the output is a modified reconstructed picture before in-loop filtering.

#### Intra sample prediction

##### General intra sample prediction

Inputs to this process are:

* a sample location ( xTbCmp, yTbCmp ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable predModeIntra specifying the intra prediction mode,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable whRatio is set equal to Min( Abs( Log2( nTbW / nTbH ) ), 2 ).

The variables refW and refH are derived as follows:

refW = ( nTbH > nTbW )  ?  ( nTbW + ( nTbH >> whRatio ) + Ceil( nTbH / 32 ) )  :  ( nTbW \* 2 ) (8‑16)

refH = ( nTbW > nTbH )  ?  ( nTbH + ( nTbW >> whRatio ) + Ceil( nTbW / 32 ) )  :  ( nTbH \* 2 ) (8‑17)

For the generation of the reference samples p[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, the following ordered steps apply:

1. The reference sample availability marking process as specified in clause 8.2.4.2.2 is invoked with the sample location ( xTbCmp, yTbCmp ), the reference sample width refW, the reference sample height refH, the colour component index cIdx as inputs, and the reference samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 as output.
2. When at least one sample refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 is marked as "not available for intra prediction", the reference sample substitution process as specified in clause 8.2.4.2.3 is invoked with the reference sample width refW, the reference sample height refH, the reference samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, and the colour component index cIdx as inputs, and the modified reference samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 as output.
3. The reference sample filtering process as specified in clause 8.2.4.2.4 is invoked with the transform block width nTbW and height nTbH, the reference sample width refW, the reference sample height refH, the unfiltered samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, and the colour component index cIdx as inputs, and the reference samples p[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 as output.

The intra sample prediction process according to predModeIntra applies as follows:

* If predModeIntra is equal to INTRA\_PLANAR, the corresponding intra prediction mode process specified in clause 8.2.4.2.5 is invoked with the transform block width nTbW, and the transform block height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.
* Otherwise, if predModeIntra is equal to INTRA\_DC, the corresponding intra prediction mode process specified in clause 8.2.4.2.6 is invoked with the transform block width nTbW, the transform block height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.
* Otherwise, if predModeIntra is equal to INTRA\_CCLM, the corresponding intra prediction mode process specified in clause 8.2.4.2.8 is invoked with the sample location ( xTbC, yTbC ) set equal to ( xTbCmp, yTbCmp ), the transform block width nTbW and height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.
* Otherwise, the corresponding intra prediction mode process specified in clause 8.2.4.2.7 is invoked with the intra prediction mode predModeIntra, the transform block width nTbW, the transform block height nTbH, the reference sample width refW, the reference sample height refH, and the reference sample array p as inputs, and the modified intra prediction mode predModeIntra and the predicted sample array predSamples as outputs.

When one of the following conditions is true, the position-dependent prediction sample filtering process specified in clause 8.2.4.2.9 is invoked with the intra prediction mode predModeIntra, the transform block width nTbW, the transform block height nTbH, the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1, the reference sample width refW, the reference sample height refH, the reference samples p[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, and the colour component index cIdx as inputs, and the output is the modified predicted sample array predSamples:

* predModeIntra is equal to INTRA\_PLANAR
* predModeIntra is equal to INTRA\_DC
* predModeIntra is equal to INTRA\_ANGULAR18
* predModeIntra is equal to INTRA\_ANGULAR50
* predModeIntra is less than or equal to INTRA\_ANGULAR10
* predModeIntra is greater than or equal to INTRA\_ANGULAR58

##### Reference sample availability marking process

Inputs to this process are:

* a sample location ( xTbCmp, yTbCmp ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the reference samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 for intra sample prediction.

The refW + refH + 1 neighbouring samples refUnfilt[ x ][ y ] that are constructed samples prior to the in-loop filter process, with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, are derived as follows:

* The neighbouring location (xNbCmp, yNbCmp ) is specified by:

( xNbCmp, yNbCmp ) = ( xTbCmp + x, yTbCmp + y ) (8‑18)

* The current luma location ( xTbY, yTbY ) and the neighbouring luma location (xNbY, yNbY ) are derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTbCmp, yTbCmp ) : ( xTbCmp << 1, yTbCmp << 1 ) (8‑19)

( xNbY, yNbY ) = ( cIdx  = =  0 ) ? ( xNbCmp, yNbCmp ) : ( xNbCmp << 1, yNbCmp << 1 ) (8‑20)

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the neighbouring luma location ( xNbY, yNbY ) as inputs, and the output is assigned to availableN.

– Each sample refUnfilt[ x ][ y ] is derived as follows:

* If availableN is equal to FALSE, the sample refUnfilt[ x ][ y ] is marked as "not available for intra prediction".
* Otherwise, the sample refUnfilt[ x ][ y ] is marked as "available for intra prediction" and the sample at the location ( xNbCmp, yNbCmp ) is assigned to refUnfilt[ x ][ y ].

##### Reference sample substitution process

Inputs to this process are:

* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* reference samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 for intra sample prediction,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified reference samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 for intra sample prediction.

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to BitDepthY.

– Otherwise, bitDepth is set equal to BitDepthC.

The values of the samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are modified as follows:

* If all samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are marked as "not available for intra prediction", all values of refUnfilt[ x ][ y ] are set equal to 1  <<  ( bitDepth − 1 ).
* Otherwise (at least one but not all samples refUnfilt[ x ][ y ] are marked as "not available for intra prediction"), the following ordered steps apply:

1. When refUnfilt[ −1 ][ refH − 1 ] is marked as "not available for intra prediction", search sequentially starting from x = −1, y = refH − 1 to x = −1, y = −1, then from x = 0, y = −1 to x = refW − 1, y = −1, for a sample refUnfilt[ x ][ y ] that is marked as "available for intra prediction". Once a sample refUnfilt[ x ][ y ] marked as "available for intra prediction" is found, the search is terminated and the value of refUnfilt[ −1 ][ refH − 1 ] is set equal to the value of refUnfilt[ x ][ y ].
2. For x = −1, y = refH − 2..−1, when refUnfilt[ x ][ y ] is marked as "not available for intra prediction", the value of refUnfilt[ x ][ y ] is set equal to the value of refUnfilt[ x ][ y + 1 ].
3. For x = 0..refW − 1, y = −1, when refUnfilt[ x ][ y ] is marked as "not available for intra prediction", the value of refUnfilt[ x ][ y ] is set equal to the value of refUnfilt[ x − 1 ][ y ].

All samples refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are marked as "available for intra prediction".

##### Reference sample filtering process

Inputs to this process are:

* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* the (unfiltered) neighbouring samples refUnfilt[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the reference samples p[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1.

The variable nTbS is set equal to ( Log2 ( nTbW ) + Log2 ( nTbH ) )  >>  1.

The variable whRatio is set equal to Min( Abs( Log2( nTbW / nTbH ) ), 2 ).

The variable wideAngle is derived as follows:

* If all of the following conditions are true, wideAngle is set equal to 1.
* nTbW is greater than nTbH
* predModeIntra is greater than or equal to 2
* predModeIntra is less than ( whRatio > 1 ) ? 12 : 8
* Otherwise, if all of the following conditions are true, wideAngle is set equal to 1.
* nTbH is greater than nTbW
* predModeIntra is less than or equal to 66
* predModeIntra is greater than ( whRatio > 1 ) ? 56 : 60
* Otherwise, wideAngle is set to 0.

The variable filterFlag is derived as follows:

* If one or more of the following conditions is true, filterFlag is set equal to 0:
* predModeIntra is equal to INTRA\_DC,
* predModeIntra is equal to INTRA\_PLANAR and nTbW \* nTbH is less than or equal to 32,
* cIdx is not equal to 0.
* Otherwise, if predModeIntra is equal to INTRA\_PLANAR and nTbW \* nTbH is greater than 32, filterFlag is set equal to 1.
* Otherwise, the following applies:
* The variable minDistVerHor is set equal to Min( Abs( predModeIntra − 50 ), Abs( predModeIntra − 18 ) ).
* The variable intraHorVerDistThres[ nTbS ] is specified in Table 8‑4.
* The variable filterFlag is derived as follows:
* If minDistVerHor is greater than intraHorVerDistThres[ nTbS ] or wideAngle is equal to 1, filterFlag is set equal to 1.
* Otherwise, filterFlag is set equal to 0.

Table 8‑4 – Specification of intraHorVerDistThres[ nTbS ] for various transform block sizes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **nTbS = 2** | **nTbS = 3** | **nTbS = 4** | **nTbS = 5** | **nTbS = 6** | **nTbS = 7** |
| **intraHorVerDistThres[ nTbS ]** | 20 | 14 | 2 | 0 | 20 | 0 |

For the derivation of the reference samples p[ x ][ y ] the following applies:

* If filterFlag is equal to 1, the filtered sample values p[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are derived as follows:

p[ −1 ][ −1 ] = ( refUnfilt[ −1 ][ 0 ] + 2 \* refUnfilt[ −1 ][ −1 ] + refUnfilt[ 0 ][ −1 ] + 2 )  >>  2 (8‑21)

p[ −1 ][ y ] = ( refUnfilt[ −1 ][ y + 1 ] + 2 \* refUnfilt[ −1 ][ y ] + refUnfilt[ −1 ][ y − 1 ] + 2 )  >>  2  
 for y = 0..refH − 2 (8‑22)

p[ −1 ][ refH − 1 ] = refUnfilt[ −1 ][ refH − 1 ] (8‑23)

p[ x ][ −1 ] = ( refUnfilt[ x − 1 ][ −1 ] + 2 \* refUnfilt[ x ][ −1 ] + refUnfilt[ x + 1 ][ −1 ] + 2 )  >>  2  
 for x = 0..refW − 2  (8‑24)

p[ refW − 1 ][ −1 ] = refUnfilt[ refW − 1 ][ −1 ] (8‑25)

* Otherwise, the reference samples values p[ x ][ y ] are set equal to the unfiltered sample values refUnfilt[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1.

##### Specification of INTRA\_PLANAR intra prediction mode

Inputs to this process are:

* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* the neighbouring samples p[ x ][ y ], with x = −1, y = −1..nTbH and x = 0..nTbW, y = −1.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1 and y = 0..nTbH − 1, are derived as follows:

predV[ x ][ y ] = ( ( nTbH − 1 − y ) \* p[ x ][ −1 ] + ( y + 1 ) \* p[ −1 ][ nTbH ] ) << Log2 ( nTbW )  (8‑26)

predH[ x ][ y ] = ( ( nTbW − 1 − x ) \* p[ −1 ][ y ] + ( x + 1 ) \* p[ nTbW ][ −1 ] ) << Log2 ( nTbH )  (8‑27)

predSamples[ x ][ y ] = ( predV[ x ][ y ] + predH[ x ][ y ] + nTbW \* nTbH )  >>    
 (Log2 ( nTbW ) + Log2 ( nTbH ) + 1 ) (8‑28)

##### Specification of INTRA\_DC intra prediction mode

Inputs to this process are:

* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* the neighbouring samples p[ x ][ y ], with x = −1, y = −1..nTbH − 1 and x = 0..nTbW − 1 , y = −1.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1, are derived by the following ordered steps:

1. A variable dcVal is derived as follows:

* When nTbW is equal to nTbH:

dcVal (8‑29)

* When nTbW is greater than nTbH:

dcVal (8‑30)

* When nTbW is less than nTbH:

dcVal (8‑31)

1. The prediction samples predSamples[x][y] are derived as follows:

predSamples[ x ][ y ] = dcVal, with x = 0.. nTbW − 1, y = 0.. nTbH − 1 (8‑32)

##### Specification of INTRA\_ANGULAR2..INTRA\_ANGULAR66 intra prediction modes

Inputs to this process are:

* the intra prediction mode predModeIntra,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* the neighbouring samples p[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1

Outputs of this process are the modified intra prediction mode predModeIntra and the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable whRatio is set equal to Min( Abs( Log2( nTbW / nTbH ) ), 2 ).

For non-square blocks (nTbW is not equal to nTbH), the intra prediction mode predModeIntra is modified as follows:

* If all of the following conditions are true, predModeIntra is set equal to ( predModeIntra + 65 ).
* nTbW is greater than nTbH
* predModeIntra is greater than or equal to 2
* predModeIntra is less than ( whRatio > 1 ) ? 12 : 8
* Otherwise, if all of the following conditions are true, predModeIntra is set equal to ( predModeIntra − 67 ).
* nTbH is greater than nTbW
* predModeIntra is less than or equal to 66
* predModeIntra is greater than ( whRatio > 1 ) ? 56 : 60

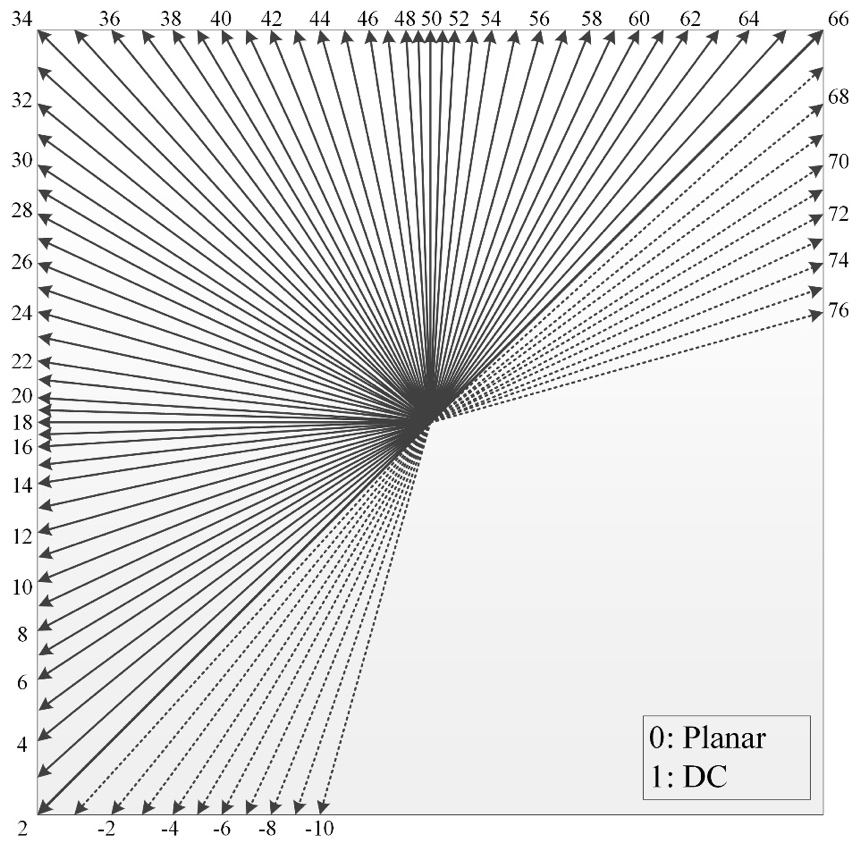


Figure 8‑1 – Intra prediction directions (informative)

Figure 8‑1 illustrates the 85 prediction directions, where the dashed directions are associated with the wide-angle modes that are only applied to non-square blocks.

Table 8‑5 – specifies the mapping table between predModeIntra and the angle parameter intraPredAngle.

Table 8‑5 – Specification of intraPredAngle

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **predModeIntra** | −**10** | −**9** | −**8** | −**7** | −**6** | −**5** | −**4** | −**3** | −**2** | −**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **intraPredAngle** | 114 | 93 | 79 | 68 | 60 | 54 | 49 | 45 | 39 | 35 | 32 | 29 | 26 | 23 | 21 | 19 | 17 |
| **predModeIntra** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** |
| **intraPredAngle** | 15 | 13 | 11 | 9 | 7 | 5 | 3 | 2 | 1 | 0 | −1 | −2 | −3 | −5 | −7 | −9 | −11 |
| **predModeIntra** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** |
| **intraPredAngle** | −13 | −15 | −17 | −19 | −21 | −23 | −26 | −29 | −32 | −29 | −26 | −23 | −21 | −19 | −17 | −15 | −13 |
| **predModeIntra** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** |
| **intraPredAngle** | −11 | −9 | −7 | −5 | −3 | −2 | −1 | 0 | 1 | 2 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| **predModeIntra** | **60** | **61** | **62** | **63** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** | **75** | **76** |
| **intraPredAngle** | 17 | 19 | 21 | 23 | 26 | 29 | 32 | 35 | 39 | 45 | 49 | 54 | 60 | 68 | 79 | 93 | 114 |

The inverse angle parameter invAngle is derived based on intraPredAngle as follows:

invAngle = Round (8‑33)

The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

– If predModeIntra is greater than or equal to 34, the following ordered steps apply:

1. The reference sample array ref[ x ] is specified as follows:

* The following applies:

ref[ x ] = p[ −1 + x ][ −1 ], with x = 0..nTbW (8‑34)

* If intraPredAngle is less than 0, the main reference sample array is extended as follows:
* When ( nTbH \* intraPredAngle )  >>  5 is less than −1,

ref[ x ] = p[ −1 ][ −1 + ( ( x \* invAngle + 128 )  >>  8 ) ],  
 with x = −1..( nTbH \* intraPredAngle )  >>  5 (8‑35)

* Otherwise,

ref[ x ] = p[ −1 + x ][ −1 ], with x = nTbW + 1..refW (8‑36)

1. The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* The index variable iIdx and the multiplication factor iFact are derived as follows:

iIdx = ( ( y + 1 ) \* intraPredAngle )  >>  5 (8‑37)

iFact = ( ( y + 1 ) \* intraPredAngle ) & 31 (8‑38)

* Depending on the value of iFact, the following applies:
* If iFact is not equal to 0, the value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] =   
 ( ( 32 − iFact ) \* ref[ x + iIdx + 1 ] + iFact \* ref[ x + iIdx + 2 ] + 16 )  >>  5 (8‑39)

* Otherwise, the value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] = ref[ x + iIdx + 1 ] (8‑40)

– Otherwise (predModeIntra is less than 34), the following ordered steps apply:

1. The reference sample array ref[ x ] is specified as follows:

* The following applies:

ref[ x ] = p[ −1 ][ −1 + x ], with x = 0..nTbH (8‑41)

* If intraPredAngle is less than 0, the main reference sample array is extended as follows:
* When ( nTbW \* intraPredAngle )  >>  5 is less than −1,

ref[ x ] = p[ −1 + ( ( x \* invAngle + 128 )  >>  8 ) ][ −1 ],  
 with x = −1..( nTbW \* intraPredAngle )  >>  5 (8‑42)

* Otherwise,

ref[ x ] = p[ −1 ][ −1 + x ], with x = nTbH + 1..refH (8‑43)

1. The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* The index variable iIdx and the multiplication factor iFact are derived as follows:

iIdx = ( ( x + 1 ) \* intraPredAngle )  >>  5 (8‑44)

iFact = ( ( x + 1 ) \* intraPredAngle ) & 31 (8‑45)

* Depending on the value of iFact, the following applies:
* If iFact is not equal to 0, the value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] =   
 ( ( 32 − iFact ) \* ref[ y + iIdx + 1 ] + iFact \* ref[ y + iIdx + 2 ] + 16 )  >>  5 (8‑46)

* Otherwise, the value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] = ref[ y + iIdx + 1 ] (8‑47)

##### Specification of INTRA\_CCLM intra prediction mode

Inputs to this process are:

* a sample location ( xTbC, yTbC ) of the top-left sample of the current transform block relative to the top-left sample of the current picture,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* chroma neighbouring samples p[ x ][ y ], with x = −1, y = 0..nTbH − 1 and x = 0..nTbW − 1, y = − 1.

Output of this process are predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The current luma location ( xTbY, yTbY ) is derived as follows:

( xTbY, yTbY )  =  ( xTbC << 1, yTbC << 1 ) (8‑48)

The variables availL, availT and availTL are derived as follows:

* The availability of left neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the neighbouring luma location ( xTbY − 1, yTbY ) as inputs, and the output is assigned to availL.
* The availability of top neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the neighbouring luma location ( xTbY, yTbY − 1 ) as inputs, and the output is assigned to availT.
* The availability of top-left neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the neighbouring luma location ( xTbY − 1, yTbY − 1 ) as inputs, and the output is assigned to availTL.

The prediction samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* If both availL and availT are equal to FALSE, the following applies:

predSamples[ x ][ y ] = 1 << ( BitDepthC − 1 ) (8‑49)

* Otherwise, the following ordered steps apply:
  1. The collocated luma samples pY[ x ][ y ] with x = 0..nTbW \* 2 − 1, y= 0..nTbH \* 2 − 1 are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY + x, yTbY + y ).
  2. The neighbouring luma samples samples pY[ x ][ y ] are derived as follows:
     + When availL is equal to TRUE, the neighbouring left luma samples pY[ x ][ y ] with x = −1..−3, y = 0..2 \* nTbH − 1, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY + x , yTbY +y ).
     + When availT is equal to TRUE, the neighbouring top luma samples pY[ x ][ y ] with x = 0..2 \* nTbW − 1, y = −1, −2, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY+ x, yTbY + y ).
     + When availTL is equal to TRUE, the neighbouring top-left luma samples pY[ x ][ y ] with x = −1, y = −1, −2, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY+ x, yTbY + y ).
  3. The down-sampled collocated luma samples pDsY[ x ][ y ] with x = 0..nTbW − 1,  y = 0..nTbH − 1 are derived as follows:
     + pDsY[ x ][ y ] with x = 1..nTbW − 1, y = 0..nTbH − 1 is derived as follows:

pDsY[ x ][ y ] = ( pY[ 2 \* x − 1 ][ 2 \* y ] +  pY[ 2 \* x − 1 ][ 2 \* y + 1 ] +   
 2\* pY[ 2 \* x ][ 2 \* y ] +  2\*pY[ 2 \* x ][ 2 \* y + 1 ] +  (8‑50)  
 pY[ 2 \* x + 1 ][ 2 \* y ] +  pY[ 2 \* x + 1 ][ 2 \* y + 1 ] + 4 ) >> 3

* + - If availL is equal to TRUE, pDsY[ 0 ][ y ] with y = 0..nTbH − 1 is derived as follows:

pDsY[ 0 ][ y ] = ( pY[ −1 ][ 2 \* y ] +  pY[ −1 ][ 2 \* y + 1 ] +   
 2\* pY[ 0 ][ 2 \* y ] + 2\*pY[ 0 ][ 2\*y + 1 ] +  (8‑51)  
 pY[ 1 ][ 2 \* y ] +  pY[ 1 ][ 2 \* y + 1 ] + 4 ) >> 3

* + - Otherwise, pDsY[ 0 ][ y ] with y = 0..nTbH − 1 is derived as follows:

pDsY[ 0 ][ y ] = ( pY[ 0 ][ 2 \* y ] + pY[ 0 ][ 2 \* y + 1 ] + 1 ) >> 1 (8‑52)

* 1. When availL is equal to TRUE, the down-sampled neighbouring left luma samples pLeftDsY[ y ] with y = 0..nTbH − 1 are derived as follows:

pLeftDsY[ y ] = ( pY[ −1 ][ 2 \* y ] +  pY[ −1 ][ 2 \* y + 1 ] +   
 2\* pY[ −2 ][ 2 \* y ] + 2\*pY[ −2 ][ 2 \* y + 1 ] +  (8‑53)  
 pY[ −3 ][ 2 \* y ] +  pY[ −3 ][ 2 \* y + 1 ] + 4 ) >> 3

* 1. When availT is equal to TRUE, the down-sampled neighbouring top luma samples pTopDsY[ x ] with x = 0..nTbH − 1 are specified as follows:
     + pTopDsY[ x ] with x = 1..nTbW − 1 is derived as follows:

pTopDsY[ x ] = ( pY[ 2 \* x − 1 ][ −2 ] + pY[ 2 \* x − 1 ][ −1 ] +   
 2\* pY[ 2 \* x ][ −2 ] +  2\*pY[ 2 \* x ][ −1 ] +  (8‑54)  
 pY[ 2 \* x + 1 ][ −2 ] + pY[ 2 \* x + 1 ][ −1 ] + 4 ) >> 3

* + - If availTL is equal to TRUE, pTopDsY[ 0 ] is derived as follows:

pTopDsY[ 0 ] = ( pY[ − 1 ][ −2 ] + pY[ − 1 ][ −1 ] +   
 2\* pY[ 0 ][ −2 ] + 2\*pY[ 0 ][ −1 ] +  (8‑55)  
 pY[ 1 ][ −2 ] + pY[ 1 ][ −1 ] + 4 ) >> 3

* + - Otherwise, pTopDsY[ 0 ] is derived as follows:

pTopDsY[ 0 ] = ( pY[ 0 ][ −2 ] + pY[ 0 ][ −1 ] + 1 ) >> 1 (8‑56)

* 1. The variables nS, xS, yS, k0, k1 are derived as follows:

nS = ( ( availL && availT ) ? Min( nTbW, nTbH ) : ( availL ? nTbH : nTbW ) ) (8‑57)

xS = 1 << ( ( ( nTbW > nTbH ) && availL && availT ) ? ( Log2( nTbW) − Log2( nTbH ) ) : 0 ) (8‑58)

yS = 1 << ( ( ( nTbH > nTbW ) && availL && availT ) ? ( Log2( nTbH) − Log2( nTbW ) ) : 0 ) (8‑59)

k1 = ( ( availL && availT ) ? Log2( nS ) + 1 : Log2( nS ) ) (8‑60)

k0 = BitDepthC + k1 − 15 (8‑61)

* 1. Variables l, c, ll, lc and k1 are derived as follows:

l = ( availL ? : 0 ) + ( availT ? : 0 ) (8‑62)

c = ( availL ? : 0 ) + ( availT ? : 0 ) (8‑63)

ll = ( availL ? : 0 ) + ( availT ? : 0 ) (8‑64)

lc = ( availL ? : 0 ) + (8‑65)  
 ( availT ? : 0 )

* 1. When k0 is greater than 0, the variable l, c, ll, lc and k1 are modified as follows

l = ( l + ( l << ( k0 − 1 ) ) ) >> k0  
c = ( c + ( c << ( k0 − 1 ) ) ) >> k0  
ll = ( ll + ( ll << ( k0 − 1 ) ) ) >> k0 (8‑66)  
lc = ( lc + ( lc << ( k0 − 1 ) ) ) >> k0  
k1 = k1 − k0

* 1. The variables a, b, and k are derived as follows:
     + If k1 is equal to 0, the following applies:

k = 0 (8‑67)

a = 0 (8‑68)

b = 1 << ( BitDepthC − 1) (8‑69)

* + - Otherwise, the following applies:

avgY = l >> k1  
errY = l & ( ( 1 << k1 ) − 1 )   
avgC = c >> k1  
errC = c & ( ( 1 << k1 ) − 1   
a1 = lc − ( ( avgY \* avgC ) << k1 + avgY \* errC + avgC \* errY )  
a2 = ll − ( ( avgY2 ) << k1 + 2 \* avgY \* errY )  
k2 = ( a1 = = 0 ) ? 0 : Max( 0, Floor( Log2( Abs( a1 ) ) ) − BitDepthC + 2 )  
k3 = ( a2 = = 0 ) ? 0 : Max( 0, Floor( Log2( Abs( a2 ) ) ) − 5 )  
k4 = k3 − k2 + BitDepthC − 2  
a1s = a1 >> k2  
a2s = a2 >> k3 (8‑70)  
a2t = ( a2s < 32 ) ? 0 : ( ( 1 << ( BitDepthY + 4 ) ) + a2s / 2 ) / a2s  
if( a2s < 32 )  
 a3 = 0  
else if( a2s >= 32 && k4 >= 0 )  
 a3 = ( a1s \* a2t ) >> k4   
else  
 a3 = ( a1s \* a2t ) << ( −k4 )  
a4 = Clip3( − 28, 28 − 1, a3 )  
a5 = a4 << 7  
k5 = ( a5 = = 0 ) ? 0 : Floor( Log2( Abs( a5 ) + ( Sign2( a5 ) − 1 ) / 2 ) ) − 5

k = 13 − k5 (8‑71)

a = a5 >> k5 (8‑72)

b = avgC − ( ( a \* avgY ) >> k ) (8‑73)

* 1. The prediction samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0.. nTbH − 1 are derived as follows:

predSamples[ x ][ y ] = Clip1C( ( ( pDsY[ x ][ y ] \* a ) >> k ) + b ) (8‑74)

##### Position-dependent intra prediction sample filtering process

Inputs to this process are:

* the intra prediction mode predModeIntra,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1,
* the neighbouring samples p[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified predicted samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1.

Depending on the value of cIdx, the function clip1Cmp is set as follows:

* If cIdx is equal to 0, clip1Cmp is set equal to Clip1Y.
* Otherwise, clip1Cmp is set equal to Clip1C.

The variable nScale is set to ( ( Log2( nTbW ) + Log2( nTbH ) − 2 )  >>  2 ).

The reference sample arrays mainRef[ x ] and sideRef[ y ], with x = 0..refW − 1 and y = 0..refH − 1 are derived as follows:

mainRef[ x ] = p[ x ][ −1 ] (8‑75)  
sideRef[ y ] = p[ −1 ][ y ]

The variables refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] with x = 0..nTbW − 1, y =0..nTbH − 1 are derived as follows:

* If predModeIntra is equal to INTRA\_PLANAR, INTRA\_DC, INTRA\_ANGULAR18, or INTRA\_ANGULAR50, the following applies:

refL[ x ][ y ] = p[ −1 ][ y ] (8‑76)

refT[ x ][ y ] = p[ x ][ −1 ] (8‑77)

wT[ y ] = 32  >>  ( ( y  <<  1 )  >>  nScale ) (8‑78)

wL[ x ] = 32  >>  ( ( x  <<  1 )  >>  nScale ) (8‑79)

wTL[ x ][ y ] = ( predModeIntra  = =  INTRA\_DC )  ?  ( ( wL[ x ]>>  4 ) + ( wT[ y ]>>  4 ) )  :  0 (8‑80)

* Otherwise, if predModeIntra is equal to INTRA\_ANGULAR2 or INTRA\_ANGULAR66, the following applies:

refL[ x ][ y ] = p[ −1 ][ x + y + 1 ] (8‑81)

refT[ x ][ y ] = p[ x + y + 1 ][ −1 ] (8‑82)

wT[ y ] = ( 32  >>  1 )  >>  ( ( y  <<  1 )  >>  nScale ) (8‑83)

wL[ x ] = ( 32  >>  1 )  >>  ( ( x  <<  1 )  >>  nScale ) (8‑84)

wTL[ x ][ y ] = 0 (8‑85)

* Otherwise, if predModeIntra is less than or equal to INTRA\_ANGULAR10, the following ordered steps apply:

1. The variables dXPos[ y ], dXFrac[ y ], dXInt[ y ] and dX[ x ][ y ] are derived as follows using invAngle as specified in clause 8.2.4.2.7 depending on intraPredMode:

dXPos[ y ] = ( ( y + 1 ) \* invAngle + 2 )  >>  2  
dXFrac[ y ] = dXPos[ y ] & 63 (8‑86)  
dXInt[ y ] = dXPos [ y ]  >>  6  
dX[ x ][ y ] = x + dXInt[ y ]

1. The variables refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] are derived as follows:

refL[ x ][ y ] = 0 (8‑87)

refT[ x ][ y ] = ( dX[ x ][ y ] < refW − 1 )  ?  ( ( 64 − dXFrac[ y ] ) \* mainRef[ dX[ x ][ y ] ] +   
 dXFrac[ y ] \* mainRef[ dX[ x ][ y ] + 1 ] + 32 )  >>  6 (8‑88)  
   :  0

wT[ y ] = ( dX[ x ][ y ] < refW − 1 )  ?  32  >>  ( ( y  <<  1 )  >>  nScale )  :  0 (8‑89)

wL[ x ] = 0 (8‑90)

wTL[ x ][ y ] = 0 (8‑91)

* Otherwise, if predModeIntra is greater than or equal to INTRA\_ANGULAR58, the following ordered steps apply:

1. The variables dYPos[ x ], dYFrac[ x ], dYInt[ x ] and dY[ x ][ y ] are derived as follows using invAngle as specified in clause 8.2.4.2.7 depending on intraPredMode:

dYPos[ x ] = ( ( x + 1 ) \* invAngle + 2 )  >>  2  
dYFrac[ x ] = dYPos[ x ] & 63 (8‑92)  
dYInt[ x ] = dYPos[ x ]  >>  6  
dY[ x ][ y ] = y + dYInt[ x ]

1. The variables refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] are derived as follows:

refL[ x ][ y ] = ( dY[ x ][ y ] < refH − 1 )  ?  ( ( 64 − dYFrac[ x ] ) \* sideRef[ dY[ x ][ y ] ] +   
 dYFrac[ x ] \* sideRef[ dY[ x ][ y ] + 1 ] + 32 )  >>  6 (8‑93)  
   :  0

refT[ x ][ y ] = 0 (8‑94)

wT[ y ] = 0 (8‑95)

wL[ x ] = ( dY[ x ][ y ] < refH − 1 )  ?  32  >>  ( ( x  <<  1 )  >>  nScale )  :  0 (8‑96)

wTL[ x ][ y ] = 0 (8‑97)

* Otherwise, refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] are all set equal to 0.

The values of the modified predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y =0..nTbH − 1 are derived as follows:

predSamples[ x ][ y ] = clip1Cmp( (  refL[ x ][ y ] \* wL[ x ] + refT[ x ][ y ] \* wT[ y ] −   
 p[ −1 ][ −1 ] \* wTL[ x ][ y ] + (8‑98) ( 64 − wL[ x ] − wT[ y ] + wTL[ x ][ y ] ) \* predSamples[ x ][ y ] + 32  )  
 >> 6 )

## Decoding process for coding units coded in inter prediction mode

### General decoding process for coding units coded in inter prediction mode

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Output of this process is a modified reconstructed picture before in-loop filtering.

The decoding process for coding units coded in inter prediction mode consists of the following ordered steps:

1. The motion vector components and reference indices of the current coding unit are derived as follows:

* If MotionModelIdc[ xCb ][ yCb ] is equal to 0, the following applies:
* The derivation process for motion vector components and reference indices as specified in clause 8.3.2.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ], the chroma motion vectors mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], the reference indices refIdxL0 and refIdxL1 and the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 as outputs.
* Otherwise ( MotionModelIdc[ xCb ][ yCb ] is equal to 1 or 2), the following applies:
* The derivation process for affine motion vector components and reference indices as specified in clause 8.3.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight as inputs, and the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0 and predFlagL1, the luma motion vector array mvLX[ xSbIdx ][ ySbIdx ], and the chroma motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..(cbWidth >> 2) − 1, and ySbIdx = 0..( cbHeight >> 2 ) − 1, with X being 0 or 1 as outputs.

1. The prediction samples of the current coding unit are derived as follows:

* The variables numSbX and numSbY are modified as follows

numSbX  =  ( MotionModelIdc[ xCb ][ yCb ]  =  =  0 )  ?  numSbX  :  ( cbWidth >> 2 ) (8‑99

numSbY  =  ( MotionModelIdc[ xCb ][ yCb ]  =  =  0 )  ?  numSbY  :  ( cbHeight >> 2 ) (8‑100

* The decoding process for inter blocks as specified in clause 8.3.4.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the luma motion vectors mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ], the chroma motion vectors mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array predSamplesL of prediction luma samples and two (cbWidth / 2)x(cbHeight / 2) arrays predSamplesCr and predSamplesCr of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

1. The residual samples of the current coding unit are derived as follows:

* The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location ( xTb0, yTb0 ) set equal to the luma location ( xCb, yCb ), the width nTbW set equal to the luma coding block width cbWidth, the height nTbH set equal to the luma coding block height cbHeight and the variable cIdx as inputs, and the array resSamplesL as output.
* The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / 2, yCb / 2 ), the width nTbW set equal to the chroma coding block width cbWidth / 2, the height nTbH set equal to the chroma coding block height cbHeight / 2 and the variable cIdx as inputs, and the array resSamplesCb as output.
* The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / 2, yCb / 2 ), the width nTbW set equal to the chroma coding block width cbWidth / 2, the height nTbH set equal to the chroma coding block height cbHeight / 2 and the variable cIdx as inputs, and the array resSamplesCr as output.

1. The reconstructed samples of the current coding unit are derived as follows:

* The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location ( xB, yB ) set equal to ( xCb, yCb ), the block width bWidth set equal to cbWidth, the block height bHeight set equal to cbHeight, the variable cIdx, the (cbWidth)x(cbHeight) array predSamples set equal to predSamplesL and the (cbWidth)x(cbHeight) array resSamples set equal to resSamplesL as inputs, and the output is a modified reconstructed picture before in-loop filtering.
* The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location ( xB, yB ) set equal to ( xCb / 2, yCb / 2 ), the block width bWidth set equal to cbWidth / 2, the block height bHeight set equal to cbHeight / 2, the variable cIdx, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to predSamplesCb and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to resSamplesCb as inputs, and the output is a modified reconstructed picture before in-loop filtering.
* The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location ( xB, yB ) set equal to ( xCb / 2, yCb / 2 ), the block width bWidth set equal to cbWidth / 2, the block height bHeight set equal to cbHeight / 2, the variable cIdx, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to predSamplesCb and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to resSamplesCb as inputs, and the output is a modified reconstructed picture before in-loop filtering.

### Derivation process for motion vector components and reference indices

#### General

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,
* the chroma motion vectors in 1/32 fractional-sample accuracy mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

Let the variable LX be RefPicListX, with X being 0 or 1, of the current picture.

For the derivation of the variables mvL0 and mvL1, refIdxL0 and refIdxL1, as well as predFlagL0 and predFlagL1, the following applies:

* If merge\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for luma motion vectors for merge mode as specified in clause 8.3.2.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight inputs, and the output being the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[ xSbIdx ][ ySbIdx ], mvL1[ xSbIdx ][ ySbIdx ], the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.
* Otherwise, for X being replaced by either 0 or 1 in the variables predFlagLX, mvLX and refIdxLX, in PRED\_LX, and in the syntax elements ref\_idx\_lX and MvdLX, the following ordered steps apply:

1. The variables numSbX and numSbY are derived as follows:

numSbX = 1 (8‑101)

numSbY = 1 (8‑102)

1. The variables refIdxLX and predFlagLX are derived as follows:

* If inter\_pred\_idc[ xCb ][ yCb ] is equal to PRED\_LX or PRED\_BI,

refIdxLX = ref\_idx\_lX[ xCb ][ yCb ] (8‑103)

predFlagLX[ 0 ][0 ] = 1 (8‑104)

* Otherwise, the variables refIdxLX and predFlagLX are specified by:

refIdxLX = −1 (8‑105)

predFlagLX[ 0 ][0 ] = 0 (8‑106)

1. The variable mvdLX is derived as follows:

mvdLX[ 0 ] = MvdLX[ xCb ][ yCb ][ 0 ] (8‑107)

mvdLX[ 1 ] = MvdLX[ xCb ][ yCb ][ 1 ] (8‑108)

1. When predFlagLX is equal to 1, the derivation process for luma motion vector prediction in clause 8.3.2.8 is invoked with the luma coding block location ( xCb, yCb ), the coding block width cbWidth, the coding block height cbHeight and the variable refIdxLX as inputs, and the output being mvpLX.
2. When predFlagLX is equal to 1, the luma motion vector mvLX is derived as follows:

uLX[ 0 ] = ( mvpLX[ 0 ] + mvdLX[ 0 ] + 218 ) % 218 (8‑109)

mvLX[ 0 ][0 ][ 0 ] = ( uLX[ 0 ] >= 217 ) ? ( uLX[ 0 ] − 218 ) : uLX[ 0 ] (8‑110)

uLX[ 1 ] = ( mvpLX[ 1 ] + mvdLX[ 1 ] + 218 ) % 218 (8‑111)

mvLX[ 0 ][0 ][ 1 ] = ( uLX[ 1 ] >= 217 ) ? ( uLX[ 1 ] − 218 ) : uLX[ 1 ] (8‑112)

NOTE 1– The resulting values of mvLX[ 0 ] and mvLX[ 1 ] as specified above will always be in the range of −217 to 217 − 1, inclusive.

When predFlagLX, with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in clause 8.3.2.13 is invoked with mvLX[ xSbIdx ][ ySbIdx ] as input, and the output being mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

#### Derivation process for luma motion vectors for merge mode

This process is only invoked when merge\_flag[ xCb ][ yPb ] is equal to 1, where ( xCb, yCb ) specify the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture.

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

The motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1 and the prediction utilization flags predFlagL0 and predFlagL1 are derived by the following ordered steps:

1. The derivation process for merging candidates from neighbouring coding units as specified in clause 8.3.2.3 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and the luma coding block width as inputs, and the output being the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1 and availableFlagB2, the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1 and refIdxLXB2, the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1 and predFlagLXB2, and the motion vectors mvLXA0, mvLXA1, mvLXB0, mvLXB1 and mvLXB2, with X being 0 or 1.
2. The derivation process for subblock-based temporal merging candidates as specified in clause 8.3.2.4 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight , the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1, the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1, the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1 and the motion vectors mvLXA0, mvLXA1, mvLXB0, mvLXB1 as inputs and the output being the availability flag availableFlagSbCol, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the reference indices refIdxLXSbCol, the luma motion vectors mvLXSbCol[ xSbIdx ][ ySbIdx ] and the prediction list utilization flags predFlagLXSbCol[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1.
3. The reference indices for the temporal merging candidate, refIdxLXCol, with X being 0 or 1, are set equal to 0.
4. The derivation process for temporal luma motion vector prediction as specified in in clause 8.3.2.11 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL0Col as inputs, and the output being the availability flag availableFlagL0Col and the temporal motion vector mvL0Col. The variables availableFlagCol, predFlagL0Col and predFlagL1Col are derived as follows:

availableFlagCol = availableFlagL0Col (8‑113)

predFlagL0Col = availableFlagL0Col (8‑114)

predFlagL1Col = 0 (8‑115)

1. When slice\_type is equal to B, the derivation process for temporal luma motion vector prediction as specified in clause 8.3.2.11 is invoked with the luma location ( xCb, yCb ), the the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL1Col as inputs, and the output being the availability flag availableFlagL1Col and the temporal motion vector mvL1Col. The variables availableFlagCol and predFlagL1Col are derived as follows:

availableFlagCol = availableFlagL0Col  | |  availableFlagL1Col (8‑116)

predFlagL1Col = availableFlagL1Col (8‑117)

1. The merging candidate list, mergeCandList, is constructed as follows:

i = 0  
if( availableFlagA1 )  
 mergeCandList[ i++ ] = A1  
if( availableFlagB1 )  
 mergeCandList[ i++ ] = B1if( availableFlagB0 )  
 mergeCandList[ i++ ] = B0 (8‑118)if( availableFlagA0 )  
 mergeCandList[ i++ ] = A0if( availableFlagSbCol )  
 mergeCandList[ i++ ] = SbCol  
if( availableFlagB2 )  
 mergeCandList[ i++ ] = B2if( availableFlagCol )  
 mergeCandList[ i++ ] = Col

1. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the mergeCandList.
2. When slice\_type is equal to B, the derivation process for combined bi-predictive merging candidates specified in clause 8.3.2.6 is invoked with mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList, numCurrMergeCand and numOrigMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices refIdxL0combCandk and refIdxL1combCandk, the prediction list utilization flags predFlagL0combCandk and predFlagL1combCandk and the motion vectors mvL0combCandk and mvL1combCandk of every new candidate combCandk being added into mergeCandList. The number of candidates being added, numCombMergeCand, is set equal to ( numCurrMergeCand − numOrigMergeCand ). When numCombMergeCand is greater than 0, k ranges from 0 to numCombMergeCand − 1, inclusive.
3. The derivation process for zero motion vector merging candidates specified in clause 8.3.2.7 is invoked with the mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList and numCurrMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices refIdxL0zeroCandm and refIdxL1zeroCandm, the prediction list utilization flags predFlagL0zeroCandm and predFlagL1zeroCandm and the motion vectors mvL0zeroCandm and mvL1zeroCandm of every new candidate zeroCandm being added into mergeCandList. The number of candidates being added, numZeroMergeCand, is set equal to ( numCurrMergeCand − numOrigMergeCand − numCombMergeCand ). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand − 1, inclusive.
4. The following applies:

* If mergeCandList[ merge\_idx[ xCb ][ yCb ] ] is equal to SbCol, the following assignments are made for xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1

refIdxLX = refIdxLXSbCol (8‑119)

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLXSbCol[ xSbIdx ][ ySbIdx ] (8‑120)

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = mvLXSbCol[ xSbIdx ][ ySbIdx ][ 0 ] (8‑121)

mvLX[ xSbIdx ][ ySbIdx ][ 1 ] = mvLXSbCol[ xSbIdx ][ ySbIdx ][ 1 ] (8‑122)

* Otherwise, the following assignments are made with N being the candidate at position merge\_idx[ xCb ][ yCb ] in the merging candidate list mergeCandList ( N = mergeCandList[ merge\_idx[ xCb ][ yCb ] ] ) and X being replaced by 0 or 1:

numSbX = 1 (8‑123)

numSbY = 1 (8‑124)

refIdxLX = refIdxLXN (8‑125)

predFlagLX[ 0 ][ 0 ] = predFlagLXN (8‑126)

mvLX[ 0 ][ 0 ][ 0 ] = mvLXN[ 0 ] (8‑127)

mvLX[ 0 ][ 0 ][ 1 ] = mvLXN[ 1 ] (8‑128)

#### Derivation process for spatial merging candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are as follows, with X being 0 or 1:

* the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1 and availableFlagB2 of the neighbouring coding units,
* the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1 and refIdxLXB2 of the neighbouring coding units,
* the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1 and predFlagLXB2 of the neighbouring coding units,
* the motion vectors in 1/16 fractional-sample accuracy mvLXA0, mvLXA1, mvLXB0, mvLXB1 and mvLXB2 of the neighbouring coding units.

For the derivation of availableFlagA1, refIdxLXA1, predFlagLXA1 and mvLXA1 the following applies:

* The luma location ( xNbA1, yNbA1 ) inside the neighbouring luma coding block is set equal to ( xCb − 1,  yCb + cbHeight − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA1, yNbA1 ) as inputs, and the output is assigned to the block availability flag availableA1.
* The variables availableFlagA1, refIdxLXA1, predFlagLXA1 and mvLXA1 are derived as follows:
* If availableA1 is equal to FALSE, availableFlagA1 is set equal to 0, both components of mvLXA1 are set equal to 0, refIdxLXA1 is set equal to −1 and predFlagLXA1 is set equal to 0, with X being 0 or 1.
* Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvLXA1 = MvLX[ xNbA1 ][ yNbA1 ] (8‑129)

refIdxLXA1 = RefIdxLX[ xNbA1 ][ yNbA1 ] (8‑130)

predFlagLXA1 = PredFlagLX[ xNbA1 ][ yNbA1 ] (8‑131)

For the derivation of availableFlagB1, refIdxLXB1, predFlagLXB1 and mvLXB1 the following applies:

* The luma location ( xNbB1, yNbB1 ) inside the neighbouring luma coding block is set equal to ( xCb + cbWidth − 1, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB1, yNbB1 ) as inputs, and the output is assigned to the block availability flag availableB1.
* The variables availableFlagB1, refIdxLXB1, predFlagLXB1 and mvLXB1 are derived as follows:
* If one or more of the following conditions are true, availableFlagB1 is set equal to 0, both components of mvLXB1 are set equal to 0, refIdxLXB1 is set equal to −1 and predFlagLXB1 is set equal to 0, with X being 0 or 1:
  + availableB1 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB1, yNbB1 ) have the same motion vectors and the same reference indices.
* Otherwise, availableFlagB1 is set equal to 1 and the following assignments are made:

mvLXB1 = MvLX[ xNbB1 ][ yNbB1 ] (8‑132)

refIdxLXB1 = RefIdxLX[ xNbB1 ][ yNbB1 ] (8‑133)

predFlagLXB1 = PredFlagLX[ xNbB1 ][ yNbB1 ] (8‑134)

For the derivation of availableFlagB0, refIdxLXB0, predFlagLXB0 and mvLXB0 the following applies:

* The luma location ( xNbB0, yNbB0 ) inside the neighbouring luma coding block is set equal to ( xCb + cbWidth, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB0, yNbB0 ) as inputs, and the output is assigned to the block availability flag availableB0.
* The variables availableFlagB0, refIdxLXB0, predFlagLXB0 and mvLXB0 are derived as follows:
* If one or more of the following conditions are true, availableFlagB0 is set equal to 0, both components of mvLXB0 are set equal to 0, refIdxLXB0 is set equal to −1 and predFlagLXB0 is set equal to 0, with X being 0 or 1:
  + availableB0 is equal to FALSE.
  + availableB1 is equal to TRUE and the luma locations ( xNbB1, yNbB1 ) and ( xNbB0, yNbB0 ) have the same motion vectors and the same reference indices.
* Otherwise, availableFlagB0 is set equal to 1 and the following assignments are made:

mvLXB0 = MvLX[ xNbB0 ][ yNbB0 ] (8‑135)

refIdxLXB0 = RefIdxLX[ xNbB0 ][ yNbB0 ] (8‑136)

predFlagLXB0 = PredFlagLX[ xNbB0 ][ yNbB0 ] (8‑137)

For the derivation of availableFlagA0, refIdxLXA0, predFlagLXA0 and mvLXA0 the following applies:

* The luma location ( xNbA0, yNbA0 ) inside the neighbouring luma coding block is set equal to ( xCb − 1,  yCb + cbWidth ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA0, yNbA0 ) as inputs, and the output is assigned to the block availability flag availableA0.
* The variables availableFlagA0, refIdxLXA0, predFlagLXA0 and mvLXA0 are derived as follows:
* If one or more of the following conditions are true, availableFlagA0 is set equal to 0, both components of mvLXA0 are set equal to 0, refIdxLXA0 is set equal to −1 and predFlagLXA0 is set equal to 0, with X being 0 or 1:
  + availableA0 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbA0, yNbA0 ) have the same motion vectors and the same reference indices.
* Otherwise, availableFlagA0 is set equal to 1 and the following assignments are made:

mvLXA0 = MvLX[ xNbA0 ][ yNbA0 ] (8‑138)

refIdxLXA0 = RefIdxLX[ xNbA0 ][ yNbA0 ] (8‑139)

predFlagLXA0 = PredFlagLX[ xNbA0 ][ yNbA0 ] (8‑140)

For the derivation of availableFlagB2, refIdxLXB2, predFlagLXB2 and mvLXB2 the following applies:

* The luma location ( xNbB2, yNbB2 ) inside the neighbouring luma coding block is set equal to ( xCb − 1, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB2, yNbB2 ) as inputs, and the output is assigned to the block availability flag availableB2.
* The variables availableFlagB2, refIdxLXB2, predFlagLXB2 and mvLXB2 are derived as follows:
* If one or more of the following conditions are true, availableFlagB2 is set equal to 0, both components of mvLXB2 are set equal to 0, refIdxLXB2 is set equal to −1 and predFlagLXB2 is set equal to 0, with X being 0 or 1:
  + availableB2 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices.
  + availableB1 is equal to TRUE and the luma locations ( xNbB1, yNbB1 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices.
  + availableFlagA0 + availableFlagA1 + availableFlagB0 + availableFlagB1 is equal to 4.
* Otherwise, availableFlagB2 is set equal to 1 and the following assignments are made:

mvLXB2 = MvLX[ xNbB2 ][ yNbB2 ] (8‑141)

refIdxLXB2 = RefIdxLX[ xNbB2 ][ yNbB2 ] (8‑142)

predFlagLXB2 = PredFlagLX[ xNbB2 ][ yNbB2 ] (8‑143)

#### Derivation process for subblock-based temporal merging candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.
* the availability flags availableFlagA0, availableFlagA1, availableFlagB0, and availableFlagB1 of the neighbouring coding units,
* the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, and refIdxLXB1 of the neighbouring coding units,
* the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, and predFlagLXB1 of the neighbouring coding units,
* the motion vectors in 1/16 fractional-sample accuracy mvLXA0, mvLXA1, mvLXB0, and mvLXB1 of the neighbouring coding units.

Outputs of this process are:

* the availability flag availableFlagSbCol,
* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the reference indices refIdxL0SbCol and refIdxL1SbCol,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0SbCol[ xSbIdx ][ ySbIdx ] and mvL1SbCol[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,
* the prediction list utilization flags predFlagL0SbCol[ xSbIdx ][ ySbIdx ] and predFlagL1SbCol[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

The availability flag availableFlagSbCol is derived as follows.

* If either slice\_temporal\_mvp\_enable\_flag or sps\_sbtmvp\_flag is equal to 0, availableFlagSbCol is set equal to 0.
* Otherwise, the following ordered steps apply.

1. The location ( xCtb, yCtb ) of the top-left sample of the luma coding tree block that contains the current coding block and the location ( xCtr, yCtr ) of the below-right center sample of the current luma coding block are derived as follows:

xCtb = ( xCb  >>  CtuLog2Size )  <<  CtuLog2Size (8‑144)

yCtb = ( yCb  >>  CtuLog2Size )  <<  CtuLog2Size (8‑145)

xCtr = xCb + ( cbWidth / 2 ) (8‑146)

yCtr = yCb + ( cbHeight / 2 ) (8‑147)

1. The luma location ( xColCtrCb, yColCtrCb ) is set equal to the top-left sample of the collocated luma coding block covering the location given by ( xCtr, yCtr ) inside ColPic relative to the top-left luma sample of the collocated picture specified by ColPic.
2. The derivation process for subblock-based temporal merging base motion data as specified in clause 8.3.2.5 is invoked with the location ( xCtb, yCtb ), the location ( xColCtrCb, yColCtrCb ), the availability flags availableFlagA0, availableFlagA1, availableFlagB0 and availableFlagB1, and the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0 and predFlagLXB1, and the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0 and refIdxLXB1, and the motion vectors mvLXA0, mvLXA1, mvLXB0 and mvLXB1, with X being 0 and 1 as inputs and the motion vectors ctrMvLX, the prediction list utilization flags ctrPredFlagLX and the reference indices ctrRefIdxLX of the collocated block, with X being 0 and 1, and the temporal motion vector tempMV as outputs.
3. The variable availableFlagSbCol is derived as follows:

* If both ctrPredFlagL0 and ctrPredFlagL1 are equal to 0, availableFlagSbCol is set equal to 0.
* Otherwise, availableFlagSbCol is set equal to 1.

When availableFlagSbCol is equal to 1, the following applies:

* The variables numSbX, numSbY, sbWidth, sbHeight and refIdxLXSbCol are derived as follows:

numSbX  =  max( cbWidth >> Log2SbtmvpSize, 1 ) (8‑148)

numSbY  =  max( cbHeight >> Log2SbtmvpSize, 1 ) (8‑149)

sbWidth  =  cbWidth / numSbX (8‑150)

sbHeight  =  cbHeight / numSbY (8‑151)

refIdxLXSbCol  =  0 (8‑152)

* For xSbIdx = 0..numSbX − 1 and ySbIdx = 0 .. numSbY − 1, the motion vectors mvLXSbCol[ xSbIdx ][ ySbIdx ] and prediction list utilization flags predFlagLXSbCol[ xSbIdx ][ ySbIdx ] are derived as follows:
* The luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture is derived as follows:

xSb  =  xCb + xSbIdx \* sbWidth (8‑153)

ySb  =  yCb + ySbIdx \* sbHeight (8‑154)

* The location ( xColSb, yColSb ) of the collocated subblock inside ColPic is derived as follows.

xColSb = Clip3( xCtb,   
 Min( CurPicWidthInSamplesY − 1, xCtb + ( 1  <<  CtbLog2SizeY ) +3 ), (8‑155)  
 xSb + ( tempMv[0]  >>  4 ) )

yColSb = Clip3( yCtb,   
 Min( CurPicHeightInSamplesY − 1, yCtb + ( 1  <<  CtbLog2SizeY ) +3 ), (8‑156)  
 ySb + ( tempMv[1]  >>  4 ) )

* The variable currCb specifies the luma coding block covering the sub-block inside the current picture.
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColSb >> 3 ) << 3, ( yColSb >> 3 ) << 3 ) inside the ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxL0 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL0SbCol[ xSbIdx ][ ySbIdx ] and availableFlagL0SbCol.
* The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxL1 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL1SbCol[ xSbIdx ][ ySbIdx ] and availableFlagL1SbCol.
* When availableFlagL0SbCol and availableFlagL1SbCol are both equal to 0, the following applies:

mvLXSbCol[ xSbIdx ][ ySbIdx ] = ctrMvLX (8‑157)

predFlagLXSbCol[ xSbIdx ][ ySbIdx ] = ctrPredFlagLX (8‑158)

#### Derivation process for subblock-based temporal merging base motion data

Inputs to this process are:

* the location ( xCtb, yCtb ) of the top-left sample of the luma coding tree block that contains the current coding block,
* the location ( xColCtrCb, yColCtrCb ) of the top-left sample of the collocated luma coding block that covers the below-right center sample.
* the availability flags availableFlagA0, availableFlagA1, availableFlagB0, and availableFlagB1 of the neighbouring coding units,
* the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, and refIdxLXB1 of the neighbouring coding units,
* the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, and predFlagLXB1 of the neighbouring coding units,
* the motion vectors in 1/16 fractional-sample accuracy mvLXA0, mvLXA1, mvLXB0, and mvLXB1 of the neighbouring coding units.

Outputs of this process are:

* the motion vectors ctrMvL0 and ctrMvL1,
* the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1,
* the reference indices ctrRefIdxL0 and ctrRefIdxL1,
* the temporal motion vector tempMV.

The variables tempMv, numRefLists and bTerminate are set as follows:

tempMv[ 0 ] = 0 (8‑159)

tempMv[ 0 ] = 0 (8‑160)

numRefLists = (slice\_type = = B) ? 2: 1 (8‑161)

bTerminate = FALSE (8‑162)

The variable currPic specifies the current picture.

For each reference picture list LX with X ranging from 0 to ( numRefList − 1 ), mvTemp is derived as follows:

1. When all of the following conditions are equal to true, X is set equal to !X:
   * DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice,
   * slice\_type is equal to B,
   * collocated\_from\_l0\_flag is equal to 0.
2. When bTerminate is equal to FALSE, and availableFlagA0 and predFlagLXA0 are equal to 1 and DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXA0]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXA0:
3. When bTerminate is equal to FALSE, availableFlagLB0 and predFlagLXB0 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXB0]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXB0.
4. When bTerminate is equal to FALSE, availableFlagB1 and predFlagLXB1 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXB1]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXB1.
5. When bTerminate is equal to FALSE, availableFlagA1 and predFlagLXB1 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXA1]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set to mvLXA1.

The location ( xColCb, yColCb ) of the collocated block inside ColPic is derived as follows.

xColCb = Clip3( xCtb,   
 Min( CurPicWidthInSamplesY − 1, xCtb + ( 1  <<  CtbLog2SizeY ) +3 ), (8‑163)  
 xColCtrCb + ( tempMv[0]  >>  4 ) )

yColCb = Clip3( yCtb,   
 Min( CurPicHeightInSamplesY − 1, yCtb + ( 1  <<  CtbLog2SizeY ) +3 ), (8‑164)  
 yColCtrCb + ( tempMv[1]  >>  4 ) )

The array colPredMode is set equal to the prediction mode array CuPredMode of the collocated picture specified by ColPic.

The motion vectors ctrMvL0 and ctrMvL1, the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1, and the reference indices ctrRefIdxL0 and ctrRefIdxL1 are derived as follows:

* If colPredMode[xColCb][yColCb] is equal to MODE\_INTER, the following applies:
* The variable currCb specifies the luma coding block covering ( xCtrCb ,yCtrCb ) inside the current picture.
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColCb >> 3 ) << 3, ( yColCb >> 3 ) << 3 ) inside the ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
  + The derivation process for temporal motion vector prediction in subclause  8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL0, and sbFlag set equal to 1 as inputs and the output being assigned to centerMvL0 and centerPredFlagL0.
  + The derivation process for temporal motion vector prediction in subclause  8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL1, and sbFlag set equal to 1 as inputs and the output being assigned to centerMvL1 and centerPredFlagL1.
* Otherwise, the following applies:

ctrPredFlagL0 = 0 (8‑165)

ctrPredFlagL1 = 0 (8‑166)

#### Derivation process for combined bi-predictive merging candidates

Inputs to this process are:

* a merging candidate list mergeCandList,
* the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,
* the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,
* the motion vectors in 1/16 fractional-sample accuracy mvL0N and mvL1N of every candidate N in mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the number of elements numOrigMergeCand within the mergeCandList after the spatial and temporal merge candidate derivation process.

Outputs of this process are:

* the merging candidate list mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the reference indices refIdxL0combCandk and refIdxL1combCandk of every new candidate combCandk added into mergeCandList during the invocation of this process,
* the prediction list utilization flags predFlagL0combCandk and predFlagL1combCandk of every new candidate combCandk added into mergeCandList during the invocation of this process,
* the motion vectors in 1/16 fractional-sample accuracy mvL0combCandk and mvL1combCandk of every new candidate combCandk added into mergeCandList during the invocation of this process.

When numOrigMergeCand is greater than 1 and less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable combIdx is set equal to 0, the variable combStop is set equal to FALSE and the following ordered steps are repeated until combStop is equal to TRUE:

1. The variables l0CandIdx and l1CandIdx are derived using combIdx as specified in Table 8‑6.
2. The following assignments are made, with l0Cand being the candidate at position l0CandIdx and l1Cand being the candidate at position l1CandIdx in the merging candidate list mergeCandList:
   * l0Cand = mergeCandList[ l0CandIdx ]
   * l1Cand = mergeCandList[ l1CandIdx ]
3. When all of the following conditions are true:
   * predFlagL0l0Cand = = 1
   * predFlagL1l1Cand = = 1
   * ( DiffPicOrderCnt( RefPicList0[ refIdxL0l0Cand ], RefPicList1[ refIdxL1l1Cand ] ) != 0 ) | |   
     ( mvL0l0Cand != mvL1l1Cand )

the candidate combCandk with k equal to ( numCurrMergeCand − numInputMergeCand ) is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to combCandk, and the reference indices, the prediction list utilization flags and the motion vectors of combCandk are derived as follows and numCurrMergeCand is incremented by 1:

refIdxL0combCandk = refIdxL0l0Cand (8‑167)

refIdxL1combCandk = refIdxL1l1Cand (8‑168)

predFlagL0combCandk = 1 (8‑169)

predFlagL1combCandk = 1 (8‑170)

mvL0combCandk[ 0 ] = mvL0l0Cand[ 0 ] (8‑171)

mvL0combCandk[ 1 ] = mvL0l0Cand[ 1 ] (8‑172)

mvL1combCandk[ 0 ] = mvL1l1Cand[ 0 ] (8‑173)

mvL1combCandk[ 1 ] = mvL1l1Cand[ 1 ] (8‑174)

numCurrMergeCand = numCurrMergeCand + 1 (8‑175)

1. The variable combIdx is incremented by 1.
2. When combIdx is equal to ( numOrigMergeCand \* ( numOrigMergeCand − 1 ) ) or numCurrMergeCand is equal to MaxNumMergeCand, combStop is set equal to TRUE.

Table 8‑6 – Specification of l0CandIdx and l1CandIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **combIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **l0CandIdx** | 0 | 1 | 0 | 2 | 1 | 2 | 0 | 3 | 1 | 3 | 2 | 3 |
| **l1CandIdx** | 1 | 0 | 2 | 0 | 2 | 1 | 3 | 0 | 3 | 1 | 3 | 2 |

#### Derivation process for zero motion vector merging candidates

Inputs to this process are:

* a merging candidate list mergeCandList,
* the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,
* the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,
* the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

* the merging candidate list mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the reference indices refIdxL0zeroCandm and refIdxL1zeroCandm of every new candidate zeroCandm added into mergeCandList during the invocation of this process,
* the prediction list utilization flags predFlagL0zeroCandm and predFlagL1zeroCandm of every new candidate zeroCandm added into mergeCandList during the invocation of this process,
* the motion vectors mvL0zeroCandm and mvL1zeroCandm of every new candidate zeroCandm added into mergeCandList during the invocation of this process.

The variable numRefIdx is derived as follows:

* If slice\_type is equal to P, numRefIdx is set equal to num\_ref\_idx\_l0\_active\_minus1 + 1.
* Otherwise (slice\_type is equal to B), numRefIdx is set equal to Min( num\_ref\_idx\_l0\_active\_minus1 + 1, num\_ref\_idx\_l1\_active\_minus1 + 1 ).

When numCurrMergeCand is less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable zeroIdx is set equal to 0 and the following ordered steps are repeated until numCurrMergeCand is equal to MaxNumMergeCand:

1. For the derivation of the reference indices, the prediction list utilization flags and the motion vectors of the zero motion vector merging candidate, the following applies:
   * If slice\_type is equal to P, the candidate zeroCandm with m equal to ( numCurrMergeCand − numInputMergeCand ) is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to zeroCandm, and the reference indices, the prediction list utilization flags and the motion vectors of zeroCandm are derived as follows and numCurrMergeCand is incremented by 1:

refIdxL0zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑176)

refIdxL1zeroCandm = −1 (8‑177)

predFlagL0zeroCandm = 1 (8‑178)

predFlagL1zeroCandm = 0 (8‑179)

mvL0zeroCandm[ 0 ] = 0 (8‑180)

mvL0zeroCandm[ 1 ] = 0 (8‑181)

mvL1zeroCandm[ 0 ] = 0 (8‑182)

mvL1zeroCandm[ 1 ] = 0 (8‑183)

numCurrMergeCand = numCurrMergeCand + 1 (8‑184)

* + Otherwise (slice\_type is equal to B), the candidate zeroCandm with m equal to ( numCurrMergeCand − numInputMergeCand ) is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to zeroCandm, and the reference indices, the prediction list utilization flags and the motion vectors of zeroCandm are derived as follows and numCurrMergeCand is incremented by 1:

refIdxL0zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑185)

refIdxL1zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑186)

predFlagL0zeroCandm = 1 (8‑187)

predFlagL1zeroCandm = 1 (8‑188)

mvL0zeroCandm[ 0 ] = 0 (8‑189)

mvL0zeroCandm[ 1 ] = 0 (8‑190)

mvL1zeroCandm[ 0 ] = 0 (8‑191)

mvL1zeroCandm[ 1 ] = 0 (8‑192)

numCurrMergeCand = numCurrMergeCand + 1 (8‑193)

1. The variable zeroIdx is incremented by 1.

#### Derivation process for luma motion vector prediction

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the prediction mvpLX in 1/16 fractional-sample accuracy of the motion vector mvLX, with X being 0 or 1.

The motion vector predictor mvpLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for motion vector predictor candidate list as specified in clause 8.3.2.9 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1, as output.
2. The motion vector predictor mvpLX with X being 0 or 1 is derived as follows:

mvpLX = mvpListLX[ mvp\_lX\_flag[ xCb ][ yCb ] ] (8‑194)

#### Derivation process for motion vector predictor candidate list

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is motion vector predictor candidate list mvpListLX in 1/16 fractional-sample accuracy with X being 0 or 1.

The motion vector predictor candidate list mvpListLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for spatial motion vector predictor candidates from neighbouring coding unit partitions as specified in clause 8.3.2.10 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the availability flags availableFlagLXN and the motion vectors mvLXN, with N being replaced by A or B, as output.
2. The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLXN, with N being replaced by A or B, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXN, with N being replaced by A or B, as output.
3. If both availableFlagLXA and availableFlagLXB are equal to 1 and mvLXA is not equal to mvLXB, availableFlagLXCol is set equal to 0.
4. Otherwise, the following applies:

* The derivation process for temporal luma motion vector prediction as specified in clause 8.3.2.11 is with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
* The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLXCol, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXCol as output.

1. The motion vector predictor candidate list, mvpListLX, is constructed as follows:

i = 0  
if( availableFlagLXA ) {  
 mvpListLX[ i++ ] = mvLXA  
 if( availableFlagLXB && ( mvLXA != mvLXB ) )  
 mvpListLX[ i++ ] = mvLXB  
} else if( availableFlagLXB )  
 mvpListLX[ i++ ] = mvLXB (8‑195)  
if( i < 2 && availableFlagLXCol )  
 mvpListLX[ i++ ] = mvLXCol  
while( i < 2 ) {  
 mvpListLX[ i ][ 0 ] = 0  
 mvpListLX[ i ][ 1 ] = 0  
 i++  
}

#### Derivation process for spatial motion vector predictor candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Outputs of this process are (with N being replaced by A or B):

* the motion vectors mvLXN in 1/16 fractional-sample accuracy of the neighbouring coding units,
* the availability flags availableFlagLXN of the neighbouring coding units.

Figure 8‑2 provides an overview of spatial motion vector neighbours.

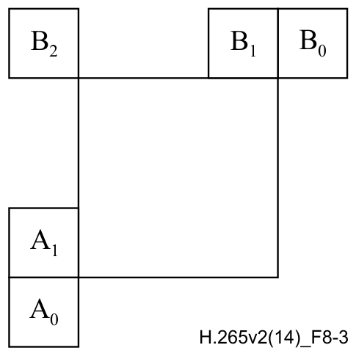


Figure 8‑2 – Spatial motion vector neighbours (informative)

The variable currCb specifies the current luma coding block at luma location ( xCb, yCb ) and the variable currPic specifies the current picture.

The variable isScaledFlagLX, with X being 0 or 1, is set equal to 0.

The motion vector mvLXA and the availability flag availableFlagLXA are derived in the following ordered steps:

1. The sample location ( xNbA0, yNbA0 ) is set equal to ( xCb − 1, yCb + cbHeight ) and the sample location ( xNbA1, yNbA1 ) is set equal to ( xNbA0, yNbA0 − 1 ).
2. The availability flag availableFlagLXA is set equal to 0 and both components of mvLXA are set equal to 0.
3. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA0, yNbA0 ) as inputs, and the output is assigned to the block availability flag availableA0.
4. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA1, yNbA1 ) as inputs, and the output is assigned to the block availability flag availableA1.
5. When availableA0 or availableA1 is equal to TRUE, the variable isScaledFlagLX is set equal to 1.
6. The following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* When availableAk is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
* If PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1 and DiffPicOrderCnt( RefPicListX[ RefIdxLX[ xNbAk ][ yNbAk ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = MvLX[ xNbAk ][ yNbAk ] (8‑196)

* Otherwise, when PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbAk ][ yNbAk ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = MvLY[ xNbAk ][ yNbAk ] (8‑197)

1. When availableFlagLXA is equal to 0, the following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ) or until availableFlagLXA is equal to 1:

* When availableAk is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
* If PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1 , availableFlagLXA is set equal to 1 and the following assignments are made:

mvLXA = MvLX[ xNbAk ][ yNbAk ] (8‑198)

refIdxA = RefIdxLX[ xNbAk ][ yNbAk ] (8‑199)

refPicListA = RefPicListX (8‑200)

* Otherwise, if PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1, availableFlagLXA is set equal to 1 and the following assignments are made:

mvLXA = MvLY[ xNbAk ][ yNbAk ] (8‑201)

refIdxA = RefIdxLY[ xNbAk ][ yNbAk ] (8‑202)

refPicListA = RefPicListY (8‑203)

* When availableFlagLXA is equal to 1, DiffPicOrderCnt( refPicListA[ refIdxA ], RefPicListX[ refIdxLX ] ) is not equal to 0, mvLXA is derived as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑204)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑205)

mvLXA = Clip3( −32768, 32767, Sign( distScaleFactor \* mvLXA ) \*   
 ( ( Abs( distScaleFactor \* mvLXA ) + 127 )  >>  8 ) ) (8‑206)

where td and tb are derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, refPicListA[ refIdxA ] ) ) (8‑207)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicListX[ refIdxLX ] ) ) (8‑208)

The motion vector mvLXB and the availability flag availableFlagLXB are derived in the following ordered steps:

1. The sample locations ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ) and ( xNbB2, yNbB2 ) are set equal to ( xCb + cbWidth, yCb − 1 ), ( xCb + cbWidth − 1, yCb − 1 ) and ( xCb − 1, yCb − 1 ), respectively.
2. The availability flag availableFlagLXB is set equal to 0 and the both components of mvLXB are set equal to 0.
3. The following applies for ( xNbBk, yNbBk ) from ( xNbB0, yNbB0 ) to ( xNbB2, yNbB2 ):

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbBk, yNbBk ) as inputs, and the output is assigned to the block availability flag availableBk.
* When availableBk is equal to TRUE and availableFlagLXB is equal to 0, the following applies:
* If PredFlagLX[ xNbBk ][ yNbBk ] is equal to 1, and DiffPicOrderCnt( RefPicListX[ RefIdxLX[ xNbBk ][ yNbBk ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLX[ xNbBk ][ yNbBk ] (8‑209)

refIdxB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑210)

* Otherwise, when PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbBk ][ yNbBk ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[ xNbBk ][ yNbBk ] (8‑211)

refIdxB = RefIdxLY[ xNbBk ][ yNbBk ] (8‑212)

1. When isScaledFlagLX is equal to 0 and availableFlagLXB is equal to 1, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = mvLXB (8‑213)

1. When isScaledFlagLX is equal to 0, availableFlagLXB is set equal to 0 and the following applies for ( xNbBk, yNbBk ) from ( xNbB0, yNbB0 ) to ( xNbB2, yNbB2 ) or until availableFlagLXB is equal to 1:

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbBk, yNbBk ) as inputs, and the output is assigned to the block availability flag availableBk.
* When availableBk is equal to TRUE and availableFlagLXB is equal to 0, the following applies:
* If PredFlagLX[ xNbBk ][ yNbBk ] is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLX[ xNbBk ][ yNbBk ] (8‑214)

refIdxB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑215)

refPicListB = RefPicListX (8‑216)

* Otherwise, when PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[ xNbBk ][ yNbBk ] (8‑217)

refIdxB = RefIdxLY[ xNbBk ][ yNbBk ] (8‑218)

refPicListB = RefPicListY (8‑219)

* When availableFlagLXB is equal to 1, DiffPicOrderCnt( refPicListB[ refIdxB ], RefPicListX[ refIdxLX ] ) is not equal to 0, mvLXB is derived as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑220)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑221)

mvLXB = Clip3( −32768, 32767, Sign( distScaleFactor \* mvLXB ) \*  
 ( ( Abs( distScaleFactor \* mvLXB ) + 127 )  >>  8 ) ) (8‑222)

where td and tb are derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, refPicListB[ refIdxB ] ) ) (8‑223)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicListX[ refIdxLX ] ) ) (8‑224)

#### Derivation process for temporal luma motion vector prediction

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a reference index refIdxLX, with X being 0 or 1.

Outputs of this process are:

* the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,
* the availability flag availableFlagLXCol.

The variable currCb specifies the current luma coding block at luma location ( xCb, yCb ).

The variables mvLXCol and availableFlagLXCol are derived as follows:

* If slice\_temporal\_mvp\_enabled\_flag is equal to 0, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
* Otherwise (slice\_temporal\_mvp\_enabled\_flag is equal to 1), the following ordered steps apply:

1. The bottom right collocated motion vector is derived as follows:

xColBr = xCb + cbWidth (8‑225)

yColBr = yCb + cbHeight (8‑226)

* If yCb  >>  CtbLog2SizeY is equal to yColBr  >>  CtbLog2SizeY, yColBr is less than pic\_height\_in\_luma\_samples and xColBr is less than pic\_width\_in\_luma\_samples, the following applies:
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColBr  >>  3 )  <<  3, ( yColBr  >>  3 )  <<  3 ) inside the collocated picture specified by ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.
* Otherwise, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

1. When availableFlagLXCol is equal to 0, the central collocated motion vector is derived as follows:

xColCtr = xCb + ( cbWidth  >>  1 ) (8‑227)

yColCtr = yCb + ( cbHeight  >>  1 ) (8‑228)

* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColCtr  >>  3 )  <<  3, ( yColCtr  >>  3 )  <<  3 ) inside the collocated picture specified by ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

#### Derivation process for collocated motion vectors

Inputs to this process are:

* a variable currCb specifying the current coding block,
* a variable colCb specifying the collocated coding block inside the collocated picture specified by ColPic,
* a luma location ( xColCb, yColCb ) specifying the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic,
* a reference index refIdxLX, with X being 0 or 1,
* a flag indicating a subblock temporal merging candidate sbFlag.

Outputs of this process are:

* the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,
* the availability flag availableFlagLXCol.

The variable currPic specifies the current picture.

The arrays predFlagL0Col[ x ][ y ], mvL0Col[ x ][ y ] and refIdxL0Col[ x ][ y ] are set equal to PredFlagL0[ x ][ y ], MvL0[ x ][ y ] and RefIdxL0[ x ][ y ], respectively, of the collocated picture specified by ColPic, and the arrays predFlagL1Col[ x ][ y ], mvL1Col[ x ][ y ] and refIdxL1Col[ x ][ y ] are set equal to PredFlagL1[ x ][ y ], MvL1[ x ][ y ] and RefIdxL1[ x ][ y ], respectively, of the collocated picture specified by ColPic.

[Ed. (BB): Define ColPic NoBackwardPredFlag.]

The variables mvLXCol and availableFlagLXCol are derived as follows:

* If colCb is coded in an intra prediction mode, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
* Otherwise, the motion vector mvCol, the reference index refIdxCol and the reference list identifier listCol are derived as follows:
  + If sbFlag is equal to 0, availableFlagLXCol is set to 1 and the following applies:
  + If predFlagL0Col[ xColCb ][ yColCb ] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL1Col[ xColCb ][ yColCb ], refIdxL1Col[ xColCb ][ yColCb ] and L1, respectively.
  + Otherwise, if predFlagL0Col[ xColCb ][ yColCb ] is equal to 1 and predFlagL1Col[ xColCb ][ yColCb ] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL0Col[ xColCb ][ yColCb ], refIdxL0Col[ xColCb ][ yColCb ] and L0, respectively.
  + Otherwise (predFlagL0Col[ xColCb ][ yColCb ] is equal to 1 and predFlagL1Col[ xColCb ][ yColCb ] is equal to 1), the following assignments are made:
    - * If NoBackwardPredFlag is equal to 1, mvCol, refIdxCol and listCol are set equal to mvLXCol[ xColCb ][ yColCb ], refIdxLXCol[ xColCb ][ yColCb ] and LX, respectively.
      * Otherwise, mvCol, refIdxCol and listCol are set equal to mvLNCol[ xColCb ][ yColCb ], refIdxLNCol[ xColCb ][ yColCb ] and LN, respectively, with N being the value of collocated\_from\_l0\_flag.
  + Otherwise (sbFlag is equal to 1), the following applies:
  + If PredFlagLXCol[ xColCb ][ yColCb ] is equal to 1, mvCol, refIdxCol, and listCol are set equal to mvLXCol[ xColCb ][ yColCb ], refIdxLXCol[ xColCb ][ yColCb ], and LX, respectively, availableFlagLXCol is set to 1.
  + Otherwise (PredFlagLXCol[ xColCb ][ yColCb ] is equal to 0), the following applies:
    - * If DiffPicOrderCnt( aPic, currPic ) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice and PredFlagLYCol[ xColCb ][ yColCb ] is equal to 1, mvCol, refIdxCol, and listCol are set to mvLYCol[ xColCb ][ yColCb ], refIdxLYCol[ xColCb ][ yColCb ] and LY, respectively, with Y being equal to !X where X being the value of X this process is invoked for. availableFlagLXCol is set to 1.
      * Both the components of mvLXCol are set to 0 and availableFlagLXCol is set equal to 0.
  + When availableFlagLXCol is equal to TRUE, mvLXCol and availableFlagLXCol are derived as follows:
  + If LongTermRefPic( currPic, currCb, refIdxLX, LX ) is not equal to LongTermRefPic( ColPic, colCb, refIdxCol, listCol ), both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
  + Otherwise, the variable availableFlagLXCol is set equal to 1, refPicListCol[ refIdxCol ] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the slice containing coding block colCb in the collocated picture specified by ColPic, and the following applies:

colPocDiff = DiffPicOrderCnt( ColPic, refPicListCol[ refIdxCol ] ) (8‑229)

currPocDiff = DiffPicOrderCnt( currPic, RefPicListX[ refIdxLX ] ) (8‑230)

* + - * If RefPicListX[ refIdxLX ] is a long-term reference picture, or colPocDiff is equal to currPocDiff, mvLXCol is derived as follows:

mvLXCol = mvCol (8‑231)

* + - * Otherwise, mvLXCol is derived as a scaled version of the motion vector mvCol as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑232)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑233)

mvLXCol =  Clip3( −32768, 32767, Sign( distScaleFactor \* mvCol ) \*   
 ( ( Abs( distScaleFactor \* mvCol ) + 127 )  >>  8 ) ) (8‑234)

where td and tb are derived as follows:

td = Clip3( −128, 127, colPocDiff ) (8‑235)

tb = Clip3( −128, 127, currPocDiff ) (8‑236)

#### Derivation process for chroma motion vectors

Input to this process is a luma motion vector in 1/16 fractional-sample accuracy mvLX.

Output of this process is a chroma motion vector in 1/32 fractional-sample accuracy mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvCLX, the following applies:

mvCLX[ 0 ] = mvLX[ 0 ] \* 2 / SubWidthC (8‑237)

mvCLX[ 1 ] = mvLX[ 1 ] \* 2 / SubHeightC (8‑238)

#### Rounding process for motion vectors

Inputs to this process are

* the motion vector mvX,
* the right shift parameter rightShift for rounding,
* the left shift parameter leftShift for resolution increase.

Output of this process is the rounded motion vector mvX.

For the rounding of mvX, the following applies:

offset = 1 << ( rightShift − 1 ) (8‑239)

mvX[ 0 ] = ( mvX[ 0 ] >= 0 ? ( mvX[ 0 ] + offset) >> rightShift :    
 − ( (− mvX[ 0 ] + offset ) >>rightShift ) ) << leftShift (8‑240)

mvX[ 1 ] = ( mvX[ 1 ] >= 0 ? ( mvX[ 1 ] + offset) >> rightShift :    
 − ( (− mvX[ 1 ] + offset ) >> rightShift ) ) << leftShift (8‑241)

### Derivation process for affine motion vector components and reference indices

#### General

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0 and predFlagL1,
* the luma subblock motion vector array in 1/16 fractional-sample accuracy mvLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..( cbWidth >> 2 ) − 1, ySbIdx = 0 .. ( cbHeight >> 2 ) − 1, X being 0 and 1,
* the chroma subblock motion vector array in 1/32 fractional-sample accuracy mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..( cbWidth >> 2 ) − 1, ySbIdx = 0 .. ( cbHeight >> 2 ) − 1, X being 0 and 1.

For the derivation of the variables mvLX[ xSbIdx ][ ySbIdx ] and mvCLC[ xSbIdx ][ ySbIdx ], the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1, the following applies:

* For the derivation of the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[ cpIdx ] and cpMvL1[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1, refIdxL0, refIdxL1, predFlagL0 and predFlagL1, the following applies:
* If merge\_affine\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for affine control point motion vectors and reference indices in affine merge mode as specified in 8.3.3.2 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight as input, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[ cpIdx ], cpMvL1[ cpIdx ], with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as output.
* Otherwise (merge\_affine\_flag[ xCb ][ yCb ] is equal to 0), for X being replaced by either 0 or 1 in the variables predFlagLX, cpMvLX, MvdCpLX, and refIdxLX, in PRED\_LX, and in the syntax element ref\_idx\_lX, the following ordered steps apply:

1. The number of control point motion vectors numCpMv is set equal to MotionModelIdc[ xCb ][ yCb ] + 1.
2. The variables refIdxLX and predFlagLX with X being 0 or 1 are derived as follows:

* If inter\_pred\_idc[ xCb ][ yCb ] is equal to PRED\_LX or PRED\_BI,

refIdxLX = ref\_idx\_lX[ xCb ][ yCb ] (8‑242)

predFlagLX = 1 (8‑243)

* Otherwise, the variables refIdxLX and predFlagLX are specified by:

refIdxLX = −1 (8‑244)

predFlagLX = 0 (8‑245)

1. The variable mvdCpLX[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1, is derived as follows:

mvdCpLX[ cpIdx ][ 0 ] = MvdCpLX[ xCb ][ yCb ][ cpIdx ][ 0 ] (8‑246)

mvdCpLX[ cpIdx ][ 1 ] = MvdCpLX[ xCb ][ yCb ][ cpIdx ][ 1 ] (8‑247)

1. When predFlagLX is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in clause 8.3.3.4 is invoked with the luma coding block location ( xCb, yCb ), and the variables cbWidth, cbHeight, refIdxLX, and the number of control point motion vectors numCpMv as inputs, and the output being mvpCpLX[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1.
2. When predFlagLX is equal to 1, the luma motion vectors cpMvLX[ cpIdx ] with cpIdx ranging from 0 to NumCpMv − 1, are derived as follows:

uLX[ cpIdx ][ 0 ] = ( mvpCpLX[ cpIdx ][ 0 ] + mvdCpLX[ cpIdx ][ 0 ] + 218 ) % 218 (8‑248)

cpMvLX[ cpIdx ][ 0 ] = (uLX[ cpIdx ][ 0 ] >= 217 ) ? (uLX[ cpIdx ][ 0 ] − 218 ) :   
 uLX[ cpIdx ][ 0 ] (8‑249)

uLX[ cpIdx ][ 1 ] = ( mvpCpLX[ cpIdx ][ 1 ] + mvdCpLX[ cpIdx ][ 1 ] + 218 ) % 218 (8‑250)

cpMvLX[ cpIdx ][ 1 ] = (uLX[ cpIdx ][ 1 ] >= 217 ) ? (uLX[ cpIdx ][ 1 ] − 218 ) :   
 uLX[ cpIdx ][ 1 ] (8‑251)

* The derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.3.3.7 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma prediction block height cbHeight, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL1[ cpIdx ], cpMvL0[ cpIdx ] with cpIdx being 0..2, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, the luma motion vector array mvLX[ xSbIdx ][ ySbIdx ] and the chroma motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..( cbWidth >> 2 ) − 1, ySbIdx = 0 .. ( cbHeight >> 2 ) − 1, X being 0 and 1 as output.
* For xSbIdx = 0..( cbWidth >> 2 ) − 1 and ySbIdx = 0..( cbHeight >> 2 ) − 1, the motion vectors MvLX with X being 0 and 1 are derived as follows:
* The luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture is derived as follows:

( xSb, ySb )  =  ( xCb + xSbIdx \* 4, yCb + ySbIdx \* 4 ) (8‑252)

* If xSbIdx and ySbIdx are both equal to 0 (top-left subblock), the following applies for x = 0..3 and y = 0..3:

MvL0[ xSb + x ][ ySb + y ] = cpMvL0[ 0 ] (8‑253)

MvL1[ xSb + x ][ ySb + y ] = cpMvL1[ 0 ] (8‑254)

* Otherwise, if xSbIdx is equal to ( cbWidth >> 2 ) − 1 and ySbIdx is equal to 0 (top-right subblock), the following applies for x = 0..3 and y = 0..3:

MvL0[ xSb + x ][ ySb + y ] = cpMvL0[ 1 ] (8‑255)

MvL1[ xSb + x ][ ySb + y ] = cpMvL1[ 1 ] (8‑256)

* Otherwise, if numCpMv is equal to 3 and xSbIdx is equal to 0 and ySbIdx is equal to ( cbHeight >> 2 ) − 1 (below­left subblock), the following applies for x = 0..3 and y = 0..3:

MvL0[ xSb + x ][ ySb + y ] = cpMvL0[ 2 ] (8‑257)

MvL1[ xSb + x ][ ySb + y ] = cpMvL1[ 2 ] (8‑258)

* Otherwise, the following applies for x = 0..3 and y = 0..3:

MvL0[ xSb + x ][ ySb + y ] = mvL0[ xSbIdx ][ ySbIdx ] (8‑259)

MvL1[ xSb + x ][ ySb + y ] = mvL1[ xSbIdx ][ ySbIdx ] (8‑260)

* For x = 0..cbWidth − 1 and y = 0..cbHeight − 1, the reference indices refIdxLX and the prediction list utilization flags PredFlagLX with X being 0 and 1 are derived as follows:

RefIdxL0[ xCb + x ][ yCb + y ] = refIdxL0 (8‑261)

RefIdxL1[ xCb + x ][ yCb + y ] = refIdxL1 (8‑262)

PredFlagL0[ xCb + x ][ yCb + y ] = predFlagL0 (8‑263)

PredFlagL1[ xCb + x ][ yCb + y ] = predFlagL1 (8‑264)

#### Derivation process for luma affine control point motion vectors and reference indices in affine merge mode

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

* the number of control point motion vectors numCpMv,
* the luma affine control point motion vectors cpMvL0[ cpIdx ], cpMvL1[ cpIdx ], with cpIdx ranging from 0 to numCpMv − 1
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0 and predFlagL1.

The derivation process for affine motion information in the neighbouring blocks of the current luma coding block as specified in 8.3.3.5 is invoked with the luma location ( xCb, yCb ), the variables cbWidth, cbHeight as input, the output being availablity flags availableFlagk, luma locations of neighbouring blocks (xNbk, yNbk), the neighbouring luma coding block width nbWk, and the neighbouring luma coding block height nbHk with k ranging from 0 to 4.

The the number of control point motion vectors numCpMv, luma affine control point motion vectors cpMvL0[ cpIdx ] and cpMvL1[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 are derived as follows:

* The variable availableFlag is set equal to 0.
* For (xNbk, yNbk) with k ranging from 0 to 4, the following applies:
* When availableFlag is equal to 0 and availableFlagk is equal to 1 the following applies:
* The reference indices refIdxLX and the prediction list utilization flags predFlagLX with X being 0 and 1, and numCpMv are derived as follows

refIdxLX = RefIdxLX[ xNbk ][ yNbk ] (8‑265)

predFlagLX = PredFlagLX[ xNbk ][ yNbk ] (8‑266)

numCpMv = MotionModelIdc[ xNbk ][ yNbk ] + 1 (8‑267)

* The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[ x ][ y ] = MotionModelIdc[ xNbk ][ yNbk ] (8‑268)

* The derivation process for luma affine control point motion vectors from a neighbouring block in as specified clause 8.3.3.3 is invoked with the coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNb, yNb ) set equal to the neighbouring luma coding block location ( xNbk, yNbk ), the variables nNbW and nNbH set equal to the neighbouring luma coding block width nbWk and the neighbouring luma coding block height nbHk, and the number of control point motion vectors numCpMv as input, and the luma affine control point motion vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 as output.
* The variable availableFlag is set equal to 1.

#### Derivation process for luma affine control point motion vectors from a neighbouring block

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* a luma location ( xNb, yNb ) specifying the top-left sample of the neighbouring luma coding block relative to the top-left luma sample of the current picture,
* two variables nNbW and nNbH specifying the width and the height of the neighbouring luma coding block,
* the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1.

The variables log2NbW and log2NbH are derived as follows:

log2NbW = Log2( nNbW ) (8‑269)

log2NbH = Log2( nNbH ) (8‑270)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

mvScaleHor = MvLX[ xNb ][ yNb ][ 0 ] << 7 (8‑271)

mvScaleVer = MvLX[ xNb ][ yNb ][ 1 ] << 7 (8‑272)

dHorX = ( MvLX[ xNb + nNbW − 1 ][ yNb ][ 0 ] − MvLX[ xNb ][ yNb ][ 0 ] ) << ( 7 − log2NbW ) (8‑273)

dVerX = ( MvLX[ xNb + nNbW − 1 ][ yNb ][ 1 ] − MvLX[ xNb ][ yNb ][ 1 ] )  << ( 7 − log2NbW ) (8‑274)

The variables dHorY and dVerY are derived as follows:

* If MotionModelIdc[ xNb ][ yNb ] is equal to 2, the following applies:

dHorY = ( MvLX[ xNb ][ yNb + nNbH − 1 ][ 0 ] − MvLX[ xNb ][ yNb ][ 0 ] ) << ( 7 − log2NbH ) (8‑275)

dVerY = ( MvLX[ xNb ][ yNb + nNbH − 1 ][ 1 ] − MvLX[ xNb ][ yNb ][ 1 ] ) << ( 7 − log2NbH ) (8‑276)

* Ohterwise (MotionModelIdc[ xNb ][ yNb ] is equal to 1), the following apply,

dHorY = − dVerX (8‑277)

dVerY = dHorX (8‑278)

The luma affine control point motion vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 are derived as follows:

* The first two control point motion vectors cpMvLX[ 0 ] and cpMvLX[ 1 ] are derived as follows:

cpMvLX[ 0 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb − xNb ) + dHorY \* ( yCb − yNb ) ) (8‑279)

cpMvLX[ 0 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb − xNb ) + dVerY \* ( yCb − yNb ) ) 8‑280)

cpMvLX[ 1 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb + cbWidth − xNb ) + dHorY \* ( yCb − yNb ) ) (8‑281)

cpMvLX[ 1 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb + cbWidth − xNb ) + dVerY \* ( yCb − yNb ) ) (8‑282)

* If numCpMv is equal to 3, the third control point vector cpMvLX[ 2 ] is derived as follows:

cpMvLX[ 2 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb − xNb ) + dHorY \* ( yCb + cbHeight − yNb ) ) (8‑283)

cpMvLX[ 2 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb − xNb ) + dVerY \* ( yCb + cbHeight − yNb ) ) (8‑284)

* The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[ cpIdx ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[ cpIdx ] as output, with X being 0 or 1 and cpIdx = 0 .. numCpMv − 1.

#### Derivation process for luma affine control point motion vector predictors

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* the reference index of the current coding unit refIdxLX, with X being 0 or 1,
* the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point motion vector predictors mvpCpLX[ cpIdx ] with X being 0 or 1, and cpIdx = 0 .. numCpMv − 1.

The derivation process for the partitioning information of neighbouring blocks in clause 8.3.3.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the output being the availability flags availableFlagk, the luma locations of neighbouring blocks ( xNbk, yNbk ) and the widths nbWk and the heights nbHk of the neighbouring coding blocks with k = 0..4.

For the derivation of the control point motion vectors predictor candidate list, cpMvpListLX with X being 0 or 1, the following ordered steps apply:

1. The number of control point motion vector predictor candidates in the list numCpMvpCandLX is set equal to 0.
2. The variable availableFlagLX is set equal to 0.
3. For k ranging from 0 to 4, inclusive, the following applies:

* When availableFlagk is equal to 1 and numCpMvpCandLX is less than 2, the following applies:
* When PredFlagLX[ xNbk ][ yNbk ] is equal to 1 and DiffPicOrderCnt( RefPicListX[ RefIdxLX[ xNbk ][ yNbk ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX is set equal to 1, and the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.3.3.3 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNbk, yNbk ), the neighbouring luma coding block width and height (nNbWk, nNbHk), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
* When availableFlagLX is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvpLX[ cpIdx ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvpLX[ cpIdx ] as output, with cpIdx = 0 .. numCpMv − 1.
* If availableFlagLX is equal to 1 and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[ 0 ][ 0 ] = cpMvpLX[ 0 ] (8‑285)

cpMvpListLX[ 0 ][ 1 ] = cpMvpLX[ 1 ] (8‑286)

cpMvpListLX[ 0 ][ 2 ] = cpMvpLX[ 2 ] (8‑287)

numCpMvpCandLX = 1 (8‑288)

* Otherwise, if availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,
* numCpMv is equal to 3 and one of the following conditions is true:
* cpMvpListLX[ 0 ][ 0 ] is not equal to cpMvpLX[ 0 ]
* cpMvpListLX[ 0 ][ 1 ] is not equal to cpMvpLX[ 1 ]
* cpMvpListLX[ 0 ][ 2 ] is not equal to cpMvpLX[ 2 ]
* numCpMv is equal to 2 and one of the following conditions is true:
* cpMvpListLX[ 0 ][ 0 ] is not equal to cpMvpLX[ 0 ]
* cpMvpListLX[ 0 ][ 1 ] is not equal to cpMvpLX[ 1 ]

the following assignments are made:

cpMvpListLX[ 1 ][ 0 ] = cpMvpLX[ 0 ] (8‑289)

cpMvpListLX[ 1 ][ 1 ] = cpMvpLX[ 1 ] (8‑290)

cpMvpListLX[ 1 ][ 2 ] = cpMvpLX[ 2 ] (8‑291)

numCpMvpCandLX = 2 (8‑292)

* The variable availableFlagLX is set equal to 0.

1. For k ranging from 0 to 4, inclusive, the following applies:

* When availableFlagk is equal to 1 and numCpMvpCandLX is less than 2, the following applies:
* When PredFlagLY[ xNbk ][ yNbk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbk ][ yNbk ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX is set equal to 1, and the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.3.3.3 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNbk, yNbk ), the neighbouring luma coding block width and height (nNbWk, nNbHk), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
* When availableFlagLX is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvpLY[ cpIdx ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvpLY[ cpIdx ] as output, with cpIdx = 0 .. numCpMv − 1.
* If availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[ 0 ][ 0 ] = cpMvpLY[ 0 ] (8‑293)

cpMvpListLX[ 0 ][ 1 ] = cpMvpLY[ 1 ] (8‑294)

cpMvpListLX[ 0 ][ 2 ] = cpMvpLY[ 2 ] (8‑295)

numCpMvpCandLX = 1 (8‑296)

* Otherwise, if availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,
* numCpMv is equal to 3 and one of the following conditions is true:
* cpMvpListLX[ 0 ][ 0 ] is not equal to cpMvpLY[ 0 ]
* cpMvpListLX[ 0 ][ 1 ] is not equal to cpMvpLY[ 1 ]
* cpMvpListLX[ 0 ][ 2 ] is not equal to cpMvpLY[ 2 ]
* numCpMv is equal to 2 and one of the following conditions is true:
* cpMvpListLX[ 0 ][ 0 ] is not equal to cpMvpLY[ 0 ]
* cpMvpListLX[ 0 ][ 1 ] is not equal to cpMvpLY[ 1 ]

the following assignments are made:

cpMvpListLX[ 1 ][ 0 ] = cpMvpLY[ 0 ] (8‑297)

cpMvpListLX[ 1 ][ 1 ] = cpMvpLY[ 1 ] (8‑298)

cpMvpListLX[ 1 ][ 2 ] = cpMvpLY[ 2 ] (8‑299)

numCpMvpCandLX = 2 (8‑300)

* The variable availableFlagLX is set equal to 0.

1. When numCpMvpCandLX is less than 2, the following applies

* The derivation process for constructed affine control point motion vector prediction candidate as specified in clause 8.3.3.6 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX, with X being 0 or 1 as inputs, and the availability flag availableConsFlagLX and cpMvpLX[ cpIdx ] with X being 0 or 1, cpIdx = 0..numCpMv − 1 as outputs.
* If availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[ 0 ][ 0 ] = cpMvpLX[ 0 ] (8‑301)

cpMvpListLX[ 0 ][ 1 ] = cpMvpLX[ 1 ] (8‑302)

cpMvpListLX[ 0 ][ 2 ] = cpMvpLX[ 2 ] (8‑303)

numCpMvpCandLX = 1 (8‑304)

* Otherwise, if availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,
* numCpMv is equal to 3 and one of the following conditions is true:
* cpMvpListLX[ 0 ][ 0 ] is not equal to cpMvpLX[ 0 ]
* cpMvpListLX[ 0 ][ 1 ] is not equal to cpMvpLX[ 1 ]
* cpMvpListLX[ 0 ][ 2 ] is not equal to cpMvpLX[ 2 ]
* numCpMv is equal to 2 and one of the following conditions is true:
* cpMvpListLX[ 0 ][ 0 ] is not equal to cpMvpLY[ 0 ]
* cpMvpListLX[ 0 ][ 1 ] is not equal to cpMvpLY[ 1 ]

the following assignments are made:

cpMvpListLX[ 1 ][ 0 ] = cpMvpLX[ 0 ] (8‑305)

cpMvpListLX[ 1 ][ 1 ] = cpMvpLX[ 1 ] (8‑306)

cpMvpListLX[ 1 ][ 2 ] = cpMvpLX[ 2 ] (8‑307)

numCpMvpCandLX = 2 (8‑308)

1. When numCpMvpCandLX is less than 2, the derivation process for luma motion vector predictor candidate list as specified in clause 8.3.2.9 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1 as output. The cpMvpListLX is appended with candidates from mvpListLX as follows:

i = 0  
while ( numCpMvpCandLX < 2 ) {  
 cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvpListLX[ i ]  
 cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvpListLX[ i ] (8‑309)  
 cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvpListLX[ i ]  
 numCpMvpCandLX++  
 i++  
}

The affine control point motion vector predictor cpMvpLX with X being 0 is derived as follows:

cpMvpLX = cpMvpListLX[ mvp\_lX\_flag[ xCb ][ yCb ] ] (8‑310)

#### Derivation process for neighbouring affine blocks partition information

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

* the availability flags availableFlagk with k = 0..4,
* the luma locations (xNbk, yNbk) of the top-left sample of the neighbouring coding blocks relative to the top-left luma sample of the current picture with k = 0..4,
* the widths nbWk and the heights nbHk of the neighbouring coding blocks with k = 0..4.

The neighbouring luma locations (xNk, yNk) with k = 0..4 inside the neighbouring luma coding block are set as follows:

( xN0, yN0 ) = ( xCb − 1, yCb + cbHeight − 1 ) (8‑311)

( xN1, yN1 ) = ( xCb + cbWidth − 1, yCb − 1 ) (8‑312)

( xN2, yN2 ) = ( xCb + cbWidth − 1, yCb ) (8‑313)

( xN3, yN3 ) = ( xCb − 1, yCb + cbHeight ) (8‑314)

( xN4, yN4 ) = ( xCb − 1, yCb − 1 ) (8‑315)

The variables availableFlagk, nbWk and nbHk with k = 0..4 are derived as follows:

* The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, and the luma location ( xNbk, yNbk ) as input, and the output is assigned to the coding block availability flag availableFlagk.
* When availableFlagk is equal to 1, the following applies:
  + When MotionModelIdc[ xNbk ][ yNbk ] is equal to 0, availableFlagk is set equal 0.
  + When availableFlagk is equal to 1, xNbk, yNbk, nbWk and nbHk are set equal to CbPosX[ xNk ][ yNk ], CbPosY[ xNk ][ yNk ], CbWidth[ xNk ][ yNk ] and CbHeight[ xNk ][ yNk ], respectively.

#### Derivation process for constructed affine control point motion vector prediction candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* the reference index of the current prediction unit partition refIdxLX, with X being 0 or 1,

Output of this process are:

* the availability flag of the constructed affine control point motion vector prediction candidiates availableConsFlagLX with X being 0 or 1,
* the constructed affine control point motion vector prediction candidiates cpMvLX[ cpIdx ] with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLX[ 0 ] and the availability flag availableFlagLX[ 0 ] are derived in the following ordered steps:

1. The sample locations ( xNbB2, yNbB2 ), ( xNbB3, yNbB3 ) and ( xNbA2, yNbA2 ) are set equal to ( xCb − 1, yCb − 1 ), ( xCb , yCb − 1 ) and ( xCb − 1, yCb ), respectively.
2. The availability flag availableFlagLX[ 0 ] is set equal to 0 and both components of cpMvLX[ 0 ] are set equal to 0.
3. The following applies for ( xNbTL, yNbTL ) withTL being replaced by B2, B3, and A2:

* The availability derivation process for a coding block as specified in clause  6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNbY, yNbY ) set equal to ( xNbTL, yNbTL ) as inputs, and the output is assigned to the coding block availability flag availableTL.
* When availableTL is equal to TRUE and availableFlagLX[ 0 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbTL ][ yNbTL ] is equal to 1, and DiffPicOrderCnt( RefPicListX[ RefIdxLX[ xNbTL ][ yNbTL ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX[ 0 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 0 ] = MvLX[ xNbTL ][ yNbTL ] (8‑316)

* Otherwise, when PredFlagLY[ xNbTL ][ yNbTL ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbTL ][ yNbTL ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX[ 0 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 0 ] = MvLY[ xNbTL ][ yNbTL ] (8‑317)

* When availableFlagLX[ 0 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[ 0 ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[ 0 ] as output.

The second (top-right) control point motion vector cpMvLX[ 1 ] and the availability flag availableFlagLX[ 1 ] are derived in the following ordered steps:

1. The sample locations ( xNbB1, yNbB1 ) and ( xNbB0, yNbB0 ) are set equal to ( xCb + cbWidth − 1, yCb − 1 ) and ( xCb + cbWidth, yCb − 1 ), respectively.
2. The availability flag availableFlagLX[ 1 ] is set equal to 0 and both components of cpMvLX[ 1 ] are set equal to 0.
3. The following applies for ( xNbTR, yNbTR ) withTR being replaced by B1 and B0:

* The availability derivation process for a coding block as specified in clause  6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNbY, yNbY ) set equal to ( xNbTR, yNbTR ) as inputs, and the output is assigned to the coding block availability flag availableTR.
* When availableTR is equal to TRUE and availableFlagLX[ 0 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbTR ][ yNbTR ] is equal to 1, and DiffPicOrderCnt( RefPicListX[ RefIdxLX[ xNbTR ][ yNbTR ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX[ 1 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 1 ] = MvLX[ xNbTR ][ yNbTR ] (8‑318)

* Otherwise, when PredFlagLY[ xNbTR ][ yNbTR ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbTR ][ yNbTR ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX[ 1 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 1 ] = MvLY[ xNbTR ][ yNbTR ] (8‑319)

* When availableFlagLX[ 1 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[ 1 ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[ 1 ] as output.

The third (bottom-left) control point motion vector cpMvLX[ 2 ] and the availability flag availableFlagLX[ 2 ] are derived in the following ordered steps:

1. The sample locations ( xNbA1, yNbA1 ) and ( xNbA0, yNbA0 ) are set equal to ( xCb − 1, yCb + cbHeight − 1 ) and ( xCb − 1, yCb + cbHeight ), respectively.
2. The availability flag availableFlagLX[ 2 ] is set equal to 0 and both components of cpMvLX[ 2 ] are set equal to 0.
3. The following applies for ( xNbBL, yNbBL ) with BL being replaced by A1 and A0:

* The availability derivation process for a coding block as specified in clause  6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNbY, yNbY ) set equal to ( xNbBL, yNbBL ) as inputs, and the output is assigned to the coding block availability flag availableBL.
* When availableBL is equal to TRUE and availableFlagLX[ 2 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbBL ][ yNbBL ] is equal to 1, and DiffPicOrderCnt( RefPicListX[ RefIdxLX[ xNbBL ][ yNbBL ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX[ 2 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 2 ] = MvLX[ xNbBL ][ yNbBL ] (8‑320)

* Otherwise, when PredFlagLY[ xNbBL ][ yNbBL ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbBL ][ yNbBL ] ], RefPicListX[ refIdxLX ] ) is equal to 0, availableFlagLX[ 2 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 2 ] = MvLY[ xNbBL ][ yNbBL ] (8‑321)

* When availableFlagLX[ 2 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[ 2 ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[ 2 ] as output.

The variables availableConsFlagLX, cpMvLX[ cpIdx ] are derived as follows:

* + The variables log2CbW and log2CbH are derived as follows:

log2CbW = Log2( cbWidth ) (8‑322)

log2CbH = Log2( cbHeight ) (8‑323)

* + If availableFlagLX[ 0 ] is equal to 1 and availableFlagLX[ 1 ] is equal to 1 and availableFlagLX[ 2 ] is equal to 1, availableConsFlagLX is set equal to 1
  + Otherwise, if availableFlagLX[ 0 ] is equal to 1, and availableFlagLX[ 1 ] is equal to 1, and MotionModelIdc[ xCb ][ yCb ] is equal to 2, availableConsFlagLX is set equal to 1 and the following applies:
  + The third control point motion vector cpMvLX[ 2 ] is derived as follows:

cpMvLX[ 2 ][ 0 ] = ( cpMvLX[ 0 ][ 0 ] << 7 ) −  (8‑324)  
 ( ( cpMvLX[ 1 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 + log2CbH − log2CbW ) )

cpMvLX[ 2 ][ 1 ] = ( cpMvLX[ 0 ][ 1 ] << 7 ) +  (8‑325)   
 ( (cpMvLX[ 1 ][ 0 ] − cpMvLX[ 0 ][ 0 ]) << ( 7 + log2CbH − log2CbW ) )

* + The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[ 2 ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[ 2 ] as output, with X being 0 or 1.
  + The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[ 2 ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[ 2 ] as output, with X being 0 or 1.
  + Otherwise, if availableFlagLX[ 0 ] is equal to 1 and availableFlagLX[ 2 ] is equal to 1, availableConsFlagLX is set equal to 1, and the following applies:
  + The second control point motion vector cpMvLX[ 1 ] is derived as follow:

cpMvLX[ 1 ][ 0 ] = ( cpMvLX[ 0 ][ 0 ] << 7 ) +  (8‑326)  
 ( ( cpMvLX[ 2 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 + log2CbH − log2CbW ) )

cpMvLX[ 1 ][ 1 ] = ( cpMvLX[ 0 ][ 1 ] << 7 ) −  (8‑327)   
 ( (cpMvLX[ 2 ][ 0 ] − cpMvLX[ 0 ][ 0 ]) << ( 7 + log2CbH − log2CbW ) )

* + The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[ 1 ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[ 1 ] as output, with X being 0 or 1.
  + The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[ 1 ], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[ 1 ] as output, with X being 0 or 1.
  + Otherwise, availableConsFlagLX is set equal to 0.

#### Derivation process for motion vector arrays from affine control point motion vectors

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the luma coding block,
* the number of control point motion vectors numCpMv,
* the control point motion vectors cpMvL0[ cpIdx ] and cpMvL1[ cpIdx ], with cpIdx = 0..numCpMv − 1,
* the prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are:

* the luma subblock motion vector array mvLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..( cbWidth >> 2 ) − 1, ySbIdx = 0 .. ( cbHeight >> 2 ) − 1, X being 0 and 1,
* the chroma subblock motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..( cbWidth >> 2 ) − 1, ySbIdx = 0 .. ( cbHeight >> 2 ) − 1, X being 0 and 1.

When predFlagLX is equal to 1, the following applies for X being 0 and 1:

* The variables log2CbW and log2CbH are derived as follows:

log2CbW = Log2( cbWidth ) (8‑328)

log2CbH = Log2( cbHeight ) (8‑329)

* The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

mvScaleHor = cpMvLX[ 0 ][ 0 ] << 7 (8‑330)

mvScaleVer = cpMvLX[ 0 ][ 1 ] << 7 (8‑331)

dHorX = ( cpMvLX[ 1 ][ 0 ] − cpMvLX[ 0 ][ 0 ] ) << ( 7 − log2CbW ) (8‑332)

dVerX = ( cpMvLX[ 1 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 − log2CbW ) (8‑333)

* If numCpMv is equal to 3, the variable dHorY and dVerY are derived as follow:

dHorY = ( cpMvLX[ 2 ][ 0 ] − cpMvLX[ 0 ][ 0 ] ) << ( 7 − log2CbH ) (8‑334)

dVerY = ( cpMvLX[ 2 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 − log2CbH ) (8‑335)

* Otherwise ( numCpMv is equal to 2), the variable dHorY and dVerY are derived as follows:

dHorY = − dVerX (8‑336)

dVerY = dHorX (8‑337)

* For xSbIdx = 0..( cbWidth >> 2 ) − 1 and ySbIdx = 0..( cbHeight >> 2 ) − 1, the following applies:
  + The luma motion vector mvLX[ xSbIdx ][ ySbIdx ] is derived as follows :

xPosCb = 2 + ( xSbIdx << 2 ) (8‑338)

yPosCb = 2 + ( ySbIdx << 2 ) (8‑339)

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = ( mvScaleHor + dHorX \* xPosCb + dHorY \* yPosCb ) (8‑340)

mvLX[ xSbIdx ][ ySbIdx ][ 1 ] = ( mvScaleVer + dVerX \* xPosCb + dVerY \* yPosCb ) (8‑341)

* + The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLX[ xSbIdx ][ ySbIdx ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded mvLX[ xSbIdx ][ ySbIdx ] as output.
  + The derivation process for chroma motion vectors in clause 8.3.2.13 is invoked with mvLX[ xSbIdx ][ ySbIdx ] as input, and the chroma motion vector mvCLX[ xSbIdx ][ ySbIdx ] as output.

### Decoding process for inter blocks

#### General

This process is invoked when decoding coding unit whose CuPredMode[ xCb ][ yCb ] is not equal to MODE\_INTRA.

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* the chroma motion vectors mvCL0 and mvCL1,
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are:

* an (cbWidth)x(cbHeight) array predSamplesL of luma prediction samples,
* an (cbWidth / 2)x(cbHeight / 2) array predSamplesCb of chroma prediction samples for the component Cb,
* an (cbWidth / 2)x(cbHeight / 2) array predSamplesCr of chroma prediction samples for the component Cr.

Let predSamplesL0L and predSamplesL1L be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, predSamplesL0Cb, predSamplesL1Cb, predSamplesL0Cr and predSamplesL1Cr be (cbWidth / 2)x(cbHeight / 2) arrays of predicted chroma sample values.

The width and the height of the current luma coding sublock subCbWidth, subCbHeight are derived as follows:

sbWidth  =  cbWidth / numSbX (8‑342)

sbHeight  =  cbHeight / numSbY (8‑343)

For each coding subblock at subblock index ( xSbIdx, ySbIdx ) with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1, the following applies:

* The luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture is derived as follows:

( xSb, ySb )  =  ( xCb + xSbIdx \* sbWidth, yCb + ySbIdx \* sbHeight ) (8‑344)

* For X being each of 0 and 1, when predFlagLX is equal to 1, the following applies:

– The reference picture consisting of an ordered two-dimensional array refPicLXL of luma samples and two ordered two-dimensional arrays refPicLXCb and refPicLXCr of chroma samples is derived by invoking the process specified in clause 8.3.4.2 with refIdxLX as input.

– The array predSamplesLXL and the arrays predSamplesLXCb and predSamplesLXCr are derived by invoking the fractional sample interpolation process specified in clause 8.3.4.3 with the luma location ( xCb, yCb ), the luma coding subblock width sbWidth, the luma coding subblock height sbHeight, the motion vectors mvLX[ xSb ][ ySb ], mvCLX[ xSb ][ ySb ], and the reference arrays refPicLXL, refPicLXCb, and refPicLXCr as inputs.

* The prediction samples inside the current luma coding subblock, predSamplesL[ xL + xSb ][ yL + ySb ] with xL = 0..sbWidth − 1 and yL = 0..sbHeight − 1, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays predSamplesL0L and predSamplesL1L, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx equal to 0 as inputs.
* The prediction samples inside the current chroma component Cb coding block, predSamplesCb[ xC + xCb / 2 ][ yC + yCb / 2 ] with xC = 0..cbWidth / 2 − 1 and yC = 0..cbHeight / 2 − 1, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays predSamplesL0Cb and predSamplesL1Cb, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx equal to 1 as inputs.
* The prediction samples inside the current chroma component Cr coding block, predSamplesCr[ xC + xCb / 2 ][ yC + yCb / 2 ] with xC = 0..cbWidth / 2 − 1 and yC = 0..cbHeight / 2 − 1, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays predSamplesL0Cr and predSamplesL1Cr, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx equal to 2 as inputs.
* When MotionModelIdc[ xCb ][ yCb ] is equal to 0, the following assignments are made for x = 0..sbWidth − 1 and y = 0..sbHeight − 1:

MvL0[ xSb + x ][ ySb + y ] = mvL0[ xSbIdx ][ ySbIdx ] (8‑345)

MvL1[ xSb + x ][ ySb + y ] = mvL1[ xSbIdx ][ ySbIdx ] (8‑346)

RefIdxL0[ xSb + x ][ ySb + y ] = refIdxL0 (8‑347)

RefIdxL1[ xSb + x ][ ySb + y ] = refIdxL1 (8‑348)

PredFlagL0[ xSb + x ][ ySb + y ] = predFlagL0 (8‑349)

PredFlagL1[ xSb + x ][ ySb + y ] = predFlagL1 (8‑350)

#### Reference picture selection process

Input to this process is a reference index refIdxLX.

Output of this process is a reference picture consisting of a two-dimensional array of luma samples refPicLXL and two two-dimensional arrays of chroma samples refPicLXCb and refPicLXCr.

The output reference picture RefPicListX[ refIdxLX ] consists of a pic\_width\_in\_luma\_samples by pic\_height\_in\_luma\_samples array of luma samples refPicLXL and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples refPicLXCb and refPicLXCr.

The reference picture sample arrays refPicLXL, refPicLXCb and refPicLXCr correspond to decoded sample arrays SL, SCb and SCr derived in clause 8.5 for a previously-decoded picture.

#### Fractional sample interpolation process

##### General

Inputs to this process are:

* a luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture,
* a variable sbWidth specifying the width of the current coding subblock in luma samples,
* a variable sbHeight specifying the height of the current coding subblock in luma samples,
* a luma motion vector mvLX given in 1/16-luma-sample units,
* a chroma motion vector mvCLX given in 1/32-chroma-sample units,
* the selected reference picture sample array refPicLXL and the arrays refPicLXCb and refPicLXCr.

Outputs of this process are:

* an (sbWidth)x(sbHeight) array predSamplesLXL of prediction luma sample values,
* two (sbWidth / 2)x(sbHeight / 2) arrays predSamplesLXCb and predSamplesLXCr of prediction chroma sample values.

Let ( xIntL, yIntL ) be a luma location given in full-sample units and ( xFracL, yFracL ) be an offset given in 1/16-sample units. These variables are used only in this clause for specifying fractional-sample locations inside the reference sample arrays refPicLXL, refPicLXCb and refPicLXCr.

For each luma sample location ( xL = 0..sbWidth − 1, yL = 0..sbHeight − 1 ) inside the prediction luma sample array predSamplesLXL, the corresponding prediction luma sample value predSamplesLXL[ xL ][ yL ] is derived as follows:

* The variables xIntL, yIntL, xFracL and yFracL are derived as follows:

xIntL = xSb + ( mvLX[ 0 ]  >>  4 ) + xL (8‑351)

yIntL = ySb + ( mvLX[ 1 ]  >>  4 ) + yL (8‑352)

xFracL = mvLX[ 0 ] & 15 (8‑353)

yFracL = mvLX[ 1 ] & 15 (8‑354)

* The prediction luma sample value predSamplesLXL[ xL ][ yL ] is derived by invoking the process specified in clause 8.3.4.3.2 with ( xIntL, yIntL ), ( xFracL, yFracL ) and refPicLXL as inputs.

Let ( xIntC, yIntC ) be a chroma location given in full-sample units and ( xFracC, yFracC ) be an offset given in 1/32 sample units. These variables are used only in this clause for specifying general fractional-sample locations inside the reference sample arrays refPicLXCb and refPicLXCr.

For each chroma sample location ( xC = 0..sbWidth / 2 − 1, yC = 0.. sbHeight / 2 − 1 ) inside the prediction chroma sample arrays predSamplesLXCb and predSamplesLXCr, the corresponding prediction chroma sample values predSamplesLXCb[ xC ][ yC ] and predSamplesLXCr[ xC ][ yC ] are derived as follows:

* The variables xIntC, yIntC, xFracC and yFracC are derived as follows:

xIntC = ( xSb / SubWidthC ) + ( mvCLX[ 0 ]  >>  5 ) + xC (8‑355)

yIntC = ( ySb / SubHeightC ) + ( mvCLX[ 1 ]  >>  5 ) + yC (8‑356)

xFracC = mvCLX[ 0 ] & 31 (8‑357)

yFracC = mvCLX[ 1 ] & 31 (8‑358)

* The prediction sample value predSamplesLXCb[ xC ][ yC ] is derived by invoking the process specified in clause 8.3.4.3.3 with ( xIntC, yIntC ), ( xFracC, yFracC ) and refPicLXCb as inputs.
* The prediction sample value predSamplesLXCr[ xC ][ yC ] is derived by invoking the process specified in clause 8.3.4.3.3 with ( xIntC, yIntC ), ( xFracC, yFracC ) and refPicLXCr as inputs.

##### Luma sample interpolation process

Inputs to this process are:

– a luma location in full-sample units ( xIntL, yIntL ),

– a luma location in fractional-sample units ( xFracL, yFracL ),

– the luma reference sample array refPicLXL.

Output of this process is a predicted luma sample value predSampleLXL

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min( 4, BitDepthY − 8 ), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max( 2, 14 − BitDepthY ).

– The variable picW is set equal to pic\_width\_in\_luma\_samples and the variable picH is set equal to pic\_height\_in\_luma\_samples.

The luma interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p equal to xFracL or  yFracL are specified in Table 8‑7.

The predicted luma sample value predSampleLXL is derived as follows:

– If both xFracLand yFracL are equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = refPicLXL[ xIntL ][ yIntL ] << shift3 (8‑359)

– Otherwise if xFracL is not equal to 0 and yFracL is equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = ( fL[ xFracL ][ 0 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL − 3 ) ][ yIntL ] +  
  fL[ xFracL ][ 1 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL − 2 ) ][ yIntL ] +  
  fL[ xFracL ][ 2 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL − 1 ) ][ yIntL ] +  
  fL[ xFracL ][ 3 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL ) ][ yIntL ] +  
  fL[ xFracL ][ 4 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 1 ) ][ yIntL ] + (8‑360)  
  fL[ xFracL ][ 5 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 2 ) ][ yIntL ] +  
  fL[ xFracL ][ 6 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 3 ) ][ yIntL ] +  
  fL[ xFracL ][ 7 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 4 ) ][ yIntL ] ) >> shift1

– Otherwise if xFracL is equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = ( fL[ yFracL ][ 0 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL − 3 ) ] +  
  fL[ yFracL ][ 1 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL − 2 ) ] +  
  fL[ yFracL ][ 2 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL − 1 ) ] +  
  fL[ yFracL ][ 3 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL ) ] +  
  fL[ yFracL ][ 4 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL + 1 ) ] + (8‑361)  
  fL[ yFracL ][ 5 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL + 2 ) ] +  
  fL[ yFracL ][ 6 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL + 3 ) ] +  
  fL[ yFracL ][ 7 ] \* refPicLXL[ xIntL ][ Clip3( 0, picH − 1, yIntL + 4 ) ] ) >> shift1

– Otherwise if xFracL is not equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

* The sample array temp[ n ] with n = 0..7, is derived as follows:

yPosL = Clip3( 0, PicH − 1, yIntL + n − 3 ) (8‑362)

temp[ n ] = ( fL[ xFracL ][ 0 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL − 3 ) ][ yPosL ] +  
  fL[ xFracL ][ 1 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL − 2 ) ][ yPosL ] +  
  fL[ xFracL ][ 2 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL − 1 ) ][ yPosL ] +  
  fL[ xFracL ][ 3 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL ) ][ yPosL ] +  
  fL[ xFracL ][ 4 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 1 ) ][ yPosL ] + (8‑363)  
  fL[ xFracL ][ 5 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 2 ) ][ yPosL ] +  
  fL[ xFracL ][ 6 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 3 ) ][ yPosL ] +  
  fL[ xFracL ][ 7 ] \* refPicLXL[ Clip3( 0, picW − 1, xIntL + 4 ) ][ yPosL ] ) >> shift1

* The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL = ( fL[ yFracL ][ 0 ] \* temp[ 0 ] +  
 fL[ yFracL ][ 1 ] \* temp[ 1 ] +  
 fL[ yFracL ][ 2 ] \* temp[ 2 ] +  
 fL[ yFracL ][ 3 ] \* temp[ 3 ] +  
 fL[ yFracL ][ 4 ] \* temp[ 4 ] + (8‑364)  
 fL[ yFracL ][ 5 ] \* temp[ 5 ] +  
 fL[ yFracL ][ 6 ] \* temp[ 6 ] +  
 fL[ yFracL ][ 7 ] \* temp[ 7 ] )  >>  shift2

Table 8‑7 – Specification of the luma interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | | | | | | | |
| **fL[ p ][ 0 ]** | **fL[ p ][ 1 ]** | **fL[ p ][ 2 ]** | **fL[ p ][ 3 ]** | **fL[ p ][ 4 ]** | **fL[ p ][ 5 ]** | **fL[ p ][ 6 ]** | **fL[ p ][ 7 ]** |
| 1 | 0 | 1 | −3 | 63 | 4 | −2 | 1 | 0 |
| 2 | −1 | 2 | −5 | 62 | 8 | −3 | 1 | 0 |
| 3 | −1 | 3 | −8 | 60 | 13 | −4 | 1 | 0 |
| 4 | −1 | 4 | −10 | 58 | 17 | −5 | 1 | 0 |
| 5 | −1 | 4 | −11 | 52 | 26 | −8 | 3 | −1 |
| 6 | −1 | 3 | −9 | 47 | 31 | −10 | 4 | −1 |
| 7 | −1 | 4 | −11 | 45 | 34 | −10 | 4 | −1 |
| 8 | −1 | 4 | −11 | 40 | 40 | −11 | 4 | −1 |
| 9 | −1 | 4 | −10 | 34 | 45 | −11 | 4 | −1 |
| 10 | −1 | 4 | −10 | 31 | 47 | −9 | 3 | −1 |
| 11 | −1 | 3 | −8 | 26 | 52 | −11 | 4 | −1 |
| 12 | 0 | 1 | −5 | 17 | 58 | −10 | 4 | −1 |
| 13 | 0 | 1 | −4 | 13 | 60 | −8 | 3 | −1 |
| 14 | 0 | 1 | −3 | 8 | 62 | −5 | 2 | −1 |
| 15 | 0 | 1 | −2 | 4 | 63 | −3 | 1 | 0 |

##### Chroma sample interpolation process

Inputs to this process are:

– a chroma location in full-sample units ( xIntC, yIntC ),

– a chroma location in 1/32 fractional-sample units ( xFracC, yFracC ),

– the chroma reference sample array refPicLXC.

Output of this process is a predicted chroma sample value predSampleLXC

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min( 4, BitDepthC − 8 ), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max( 2, 14 − BitDepthC ).

– The variable picWC is set equal to pic\_width\_in\_luma\_samples / SubWidthC and the variable picHC is set equal to pic\_height\_in\_luma\_samples / SubHeightC.

The luma interpolation filter coefficients fC[ p ] for each 1/32 fractional sample position p equal to xFracC or  yFracC are specified in Table 8‑8.

The predicted chroma sample value predSampleLXC is derived as follows:

– If both xFracC and yFracC are equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC = refPicLXC[ xIntC ][ yIntC ] << shift3 (8‑365)

– Otherwise if xFracC is not equal to 0 and yFracC is equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC = ( fC[ xFracC ][ 0 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC − 1 ) ][ yIntC ] +  
  fC[ xFracC ][ 1 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC ) ][ yIntC ] +  
  fC[ xFracC ][ 2 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC + 1 ) ][ yIntC ] + (8‑366)  
  fC[ xFracC ][ 3 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC + 2 ) ][ yIntC ] ) >> shift1

– Otherwise if xFracC is equal to 0 and yFracC is not equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC = ( fC[ yFracC ][ 0 ] \* refPicLXC[ xIntC ][ Clip3( 0, picHC − 1, yIntC − 1 ) ] +  
  fC[ yFracC ][ 1 ] \* refPicLXC[ xIntC ][ Clip3( 0, picHC − 1, yIntC ) ] +  
  fC[ yFracC ][ 2 ] \* refPicLXC[ xIntC ][ Clip3( 0, picHC − 1, yIntC + 1 ) ] + (8‑367)  
  fC[ yFracC ][ 3 ] \* refPicLXC[ xIntC ][ Clip3( 0, picHC − 1, yIntC + 2 ) ] ) >> shift1

– Otherwise if xFracC is not equal to 0 and yFracC is not equal to 0, the value of predSampleLXC is derived as follows:

* The sample array temp[ n ] with n = 0..3, is derived as follows:

yPosC = Clip3( 0, PicHC − 1, yIntC + n − 1 ) (8‑368)

temp[ n ] = ( fC[ xFracC ][ 0 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC − 1 ) ][ yPosC ] +  
  fC[ xFracC ][ 1 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC ) ][ yPosC ] +  
  fC[ xFracC ][ 2 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC + 1 ) ][ yPosC ] + (8‑369)  
  fC[ xFracC ][ 3 ] \* refPicLXC[ Clip3( 0, picWC − 1, xIntC + 2 ) ][ yPosC ] ) >> shift1

* The predicted chroma sample value predSampleLXC is derived as follows:

predSampleLXC = ( fC[ yFracC ][ 0 ] \* temp[ 0 ] +  
  fC[ yFracC ][ 1 ] \* temp[ 1 ] +  
  fC[ yFracC ][ 2 ] \* temp[ 2 ] + (8‑370)  
  fC[ yFracC ][ 3 ] \* temp[ 3 ] ) >> shift2

Table 8‑8 – Specification of the chroma interpolation filter coefficients fC[ p ] for each 1/32 fractional sample position p.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | | | |
| **fC[ p ][ 0 ]** | **fC[ p ][ 1 ]** | **fC[ p ][ 2 ]** | **fC[ p ][ 3 ]** |
| 1 | −1 | 63 | 2 | 0 |
| 2 | −2 | 62 | 4 | 0 |
| 3 | -2 | 60 | 7 | −1 |
| 4 | −2 | 58 | 10 | −2 |
| 5 | −3 | 57 | 12 | −2 |
| 6 | −4 | 56 | 14 | −2 |
| 7 | −4 | 55 | 15 | −2 |
| 8 | −4 | 54 | 16 | −2 |
| 9 | −5 | 53 | 18 | −2 |
| 10 | −6 | 52 | 20 | −2 |
| 11 | −6 | 49 | 24 | −3 |
| 12 | −6 | 46 | 28 | −4 |
| 13 | −5 | 44 | 29 | −4 |
| 14 | −4 | 42 | 30 | −4 |
| 15 | −4 | 39 | 33 | −4 |
| 16 | −4 | 36 | 36 | −4 |
| 17 | −4 | 33 | 39 | −4 |
| 18 | −4 | 30 | 42 | −4 |
| 19 | −4 | 29 | 44 | −5 |
| 20 | −4 | 28 | 46 | −6 |
| 21 | −3 | 24 | 49 | −6 |
| 22 | −2 | 20 | 52 | −6 |
| 23 | −2 | 18 | 53 | −5 |
| 24 | −2 | 16 | 54 | −4 |
| 25 | −2 | 15 | 55 | −4 |
| 26 | −2 | 14 | 56 | −4 |
| 27 | −2 | 12 | 57 | −3 |
| 28 | −2 | 10 | 58 | −2 |
| 29 | −1 | 7 | 60 | −2 |
| 30 | 0 | 4 | 62 | −2 |
| 31 | 0 | 2 | 63 | −1 |

#### Weighted sample prediction process

Inputs to this process are:

* two variables nCbW and nCbH specifying the width and the height of the current coding block,
* two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,
* the prediction list utilization flags, predFlagL0 and predFlagL1,
* the reference indices refIdxL0 and refIdxL1,
* a variable cIdx specifying colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to BitDepthY.

– Otherwise, bitDepth is set equal to BitDepthC.

Variables shift1, shift2, offset1 and offset2 are derived as follows:

– The variable shift1 is set equal to Max( 2, 14 − bitDepth ) and the variable shift2 is set equal to Max( 3, 15 − bitDepth ).

– The variable offset1 is set equal to 1  <<  ( shift1 − 1 ).

– The variable offset2 is set equal to 1  <<  ( shift2 − 1 ).

Depending on the values of predFlagL0 and predFlagL1, the prediction samples pbSamples[ x ][ y ] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

– If predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, ( predSamplesL0[ x ][ y ] + offset1 )  >>  shift1 ) (8‑371)

– Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, ( predSamplesL1[ x ][ y ] + offset1 )  >>  shift1 ) (8‑372)

– Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1,  
 ( predSamplesL0[ x ][ y ] + predSamplesL1[ x ][ y ] + offset2 )  >>  shift2 ) (8‑373)

### Decoding process for the residual signal of coding blocks coded in inter prediction mode

Inputs to this process are:

* a sample location ( xTb0, yTb0 ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable nTbW specifying the width of the current transform block,
* a variable nTbH specifying the height of the current transform block,
* a variable cIdx specifying the colour component of the current block.

Output of this process is an (nTbW)x(nTbH) array resSamples.

The maximum transform block size maxTbSize is derived as follows:

maxTbSize = ( cIdx  = =  0 ) ? MaxTbSizeY : MaxTbSizeY / 2 (8‑374)

The luma sample location is derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTb0, yTb0 ) : ( xTb0 / 2, yTb0 / 2 ) (8‑375)

Depending on maxTbSize, the following applies:

* If nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables nTbW and nTbH are modified as follows:

nTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑376)

nTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑377)

1. The decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
2. When nTbW is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + nTbW, yTb0 ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture .
3. When nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0, yTb0 + nTbH ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
4. Wwhen nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + nTbW, yTb0 + nTbH ), the transform block width nTbW and height nTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

* Otherwise, the scaling and transformation process as specified in clause 8.4.2 is invoked with the luma location ( xTbY, yTbY ), the variable cIdx, the transform width nTbW and the transform height nTbH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.

## Scaling, transformation and array construction process

### Derivation process for quantization parameters

In this process, the luma quantization parameter Qp′Y and the chroma quantization parameters Qp′Cb and Qp′Cr are derived.

[Ed. (BB): QP signalling, derivation as well as invokation of this process to be defined.]

### Scaling and transformation process

Inputs to this process are:

* a luma location ( xTbY, yTbY ) specifying the top-left sample of the current luma transform block relative to the top‑left luma sample of the current picture,
* a variable cIdx specifying the colour component of the current block,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height.

Output of this process is the (nTbW)x(nTbH) array of residual samples resSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables bitDepth, bdShift and tsShift are derived as follows:

bitDepth = ( cIdx = = 0 ) ? BitDepthY : BitDepthC (8‑378)

bdShift = Max( 22 − bitDepth, 0 ) (8‑379)

tsShift = 5 + ( ( Log2( nTbW ) + Log2( nTbH ) ) / 2 ) (8‑380)

The (nTbW)x(nTbH) array of residual samples resSamples is derived as follows:

* 1. The scaling process for transform coefficients as specified in clause 8.4.3 is invoked with the transform block location ( xTbY, yTbY ), the transform width nTbW and the transform height nTbH, the colour component variable cIdx and the bit depth of the current colour component bitDepth as inputs, and the output is an (nTbW)x(nTbH) array of scaled transform coefficients d.
  2. The (nTbW)x(nTbH) array of residual samples r is derived as follows:
* If transform\_skip\_flag[ xTbY ][ yTbY ][ cIdx ] is equal to 1, the residual sample array values r[ x ][ y ] with with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

r[ x ][ y ] = d[ x ][ y ]  <<  tsShift (8‑381)

* Otherwise (transform\_skip\_flag[ xTbY ][ yTbY ][ cIdx ] is equal to 0), the transformation process for scaled transform coefficients as specified in clause 8.4.4.1 is invoked with the transform block location ( xTbY, yTbY ), the transform width nTbW and the transform height nTbH, the colour component variable cIdx and the (nTbW)x(nTbH) array of scaled transform coefficients d as inputs, and the output is an (nTbW)x(nTbH) array of residual samples r.
  1. The residual samples resSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

resSamples[ x ][ y ] = ( r[ x ][ y ] + ( 1 << ( bdShift − 1 ) ) ) >> bdShift (8‑382)

### Scaling process for transform coefficients

Inputs to this process are:

* a luma location ( xTbY, yTbY ) specifying the top-left sample of the current luma transform block relative to the top‑left luma sample of the current picture,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable cIdx specifying the colour component of the current block,
* a variable bitDepth specifying the bit depth of the current colour component.

Output of this process is the (nTbW)x(nTbH) array d of scaled transform coefficients with elements d[ x ][ y ].

The quantization parameter qP is derived as follows:

* If cIdx is equal to 0, the following applies:

qP = Qp′Y  (8‑383)

* Otherwise, if cIdx is equal to 1, the following applies:

qP = Qp′Cb (8‑384)

* Otherwise (cIdx is equal to 2), the following applies:

qP = Qp′Cr (8‑385)

The variables bdShift, rectNorm and bdOffset are derived as follows:

bdShift = bitDepth + ( ( ( Log2( nTbW ) + Log2( nTbH ) ) & 1 ) \* 8 + (8‑386)  
 ( Log2( nTbW ) + Log2( nTbH ) ) / 2 ) − 5 + dep\_quant\_enabled\_flag

rectNorm = ( ( Log2( nTbW ) + Log2( nTbH ) ) & 1 )  = =  1 ? 181 : 1 (8‑387)

bdOffset = ( 1 << bdShift ) >> 1 (8‑388)

The list levelScale[ ] is specified as levelScale[ k ] = { 40, 45, 51, 57, 64, 72 } with k = 0..5.

For the derivation of the scaled transform coefficients d[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1, the following applies:

* The intermediate scaling factor m[ x ][ y ] is set equal to 16.
* The scaling factor ls[ x ][ y ] is derived as follows:

– If dep\_quant\_enabled\_flag is equal to 1, the following applies:

ls[ x ][ y ] = ( m[ x ][ y ] \* levelScale[ (qP + 1) % 6 ] ) << ( (qP + 1) / 6 ) (8‑389)

– Otherwise (dep\_quant\_enabled\_flag is equal to 0), the following applies:

ls[ x ][ y ] = ( m[ x ][ y ] \* levelScale[ qP % 6 ] ) << ( qP / 6 ) (8‑390)

* The value dnc[ x ][ y ] is derived as follows:

dnc[ x ][ y ] = (8‑391)  
 ( TransCoeffLevel[ xTbY ][ yTbY ][ cIdx ][ x ][ y ] \* ls[ x ][ y ] \* rectNorm +bdOffset )  >>  bdShift

* The scaled transform coefficient d[ x ][ y ] is derived as follows:

d[ x ][ y ] = Clip3( CoeffMin, CoeffMax, dnc[ x ][ y ] ) (8‑392)

### Transformation process for scaled transform coefficients

#### General

Inputs to this process are:

* a luma location ( xTbY, yTbY ) specifying the top-left sample of the current luma transform block relative to the top‑left luma sample of the current picture,
* a variable nTbW specifying the width of the current transform block,
* a variable nTbH specifying the height of the current transform block,
* a variable cIdx specifying the colour component of the current block,
* an (nTbW)x(nTbH) array d[ x ][ y ] of scaled transform coefficients with x = 0..nTbW − 1, y = 0..nTbH − 1.

Output of this process is the (nTbW)x(nTbH) array r[ x ][ y ] of residual samples with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable trTypeHor specifying the horizontal transform kernel and the variable trTypeVer specifying the vertical transform kernel are derived in Table 8‑9 depending on mts\_idx[ xTbY ][ yTbY ] and CuPredMode[ xTbY ][ yTbY ].

The variables nonZeroW and nonZeroH are derived as follows:

nonZeroW = Min( nTbW, 32 ) (8‑393)

nonZeroH = Min( nTbH, 32 ) (8‑394)

The (nTbW)x(nTbH) array r of residual samples is derived as follows:

1. Each (vertical) column of scaled transform coefficients d[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nonZeroH − 1 is transformed to e[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.4.4.2 for each column x = 0..nonZeroW − 1 with the height of the transform block nTbH, the non-zero height of the scaled transform coefficients nonZeroH, the list d[ x ][ y ] with y = 0..nonZeroH − 1 and the transform type variable trType set equal to trTypeVer as inputs, and the output is the list e[ x ][ y ] with y = 0..nTbH − 1.
2. The intermediate sample values g[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 are derived as follows:

g[ x ][ y ] = Clip3( CoeffMin, CoeffMax, ( e[ x ][ y ] + 256 ) >> 9 ) (8‑395)

1. Each (horizontal) row of the resulting array g[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 is transformed to r[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.4.4.2 for each row y = 0..nTbH − 1 with the width of the transform block nTbW, the non-zero width of the resulting array g[ x ][ y ] nonZeroW, the list g[ x ][ y ] with x = 0..nonZeroW − 1 and the transform type variable trType set equal to trTypeHor as inputs, and the output is the list r[ x ][ y ] with x = 0..nTbW − 1.

Table 8‑9 – Specification of trTypeHor and trTypeVer depending on mts\_idx[ x ][ y ] and CuPredMode[ x ][ y ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| mts\_idx[ xTbY ][ yTbY ] | CuPredMode[ xTbY ][ yTbY ] = =  MODE\_INTRA | | CuPredMode[ xTbY ][ xTbY ] = =  MODE\_INTER | |
| trTypeHor | trTypeVer | trTypeHor | trTypeVer |
| −1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 2 | 2 |
| 1 | 2 | 1 | 1 | 2 |
| 2 | 1 | 2 | 2 | 1 |
| 3 | 2 | 2 | 1 | 1 |

#### Transformation process

Inputs to this process are:

* a variable nTbS specifying the horizontal sample size of transformed samples,
* a variable nonZeroS specifying the horizontal sample size of non-zero scaled transform coefficients,
* a list of scaled transform coefficients x[ j ] with j = 0..nonZeroS − 1,
* a transform kernel type variable trType.

Output of this process is the list of transformed samples y[ i ] with i = 0..nTbS − 1.

The transformation matrix derivation process as specified in clause 8.4.4.3 in invoked with the transform size nTbS and the transform kernel Type trType as input, and the transformation maxtrix transMatrix as output.

Depending on the value of trType, the following applies:, the list of transformed samples y[ i ] with i = 0..nTbS − 1 is derived as follows:

* If trType is equal to 0, the following transform matrix multiplication applies:

y[i]= with i = 0..nTbS − 1 (8‑396)

* Otherwise (trType is equal to 1 or trType is equal to 2), the following transform matrix multiplication applies:

y[i]= with i = 0..nTbS − 1 (8‑397)

#### Transformation matrix derivation process

Inputs to this process are:

* a variable nTbS specifying the horizontal sample size of scaled transform coefficients,
* the transformation kernel type trType.

Output of this process is the transformation matrix transMatrix.

The transformation matrix transMatrix is derived based on trType and nTbs as follows:

* If trType is equal to 0, the following applies:

transMatrix[ m ][ n ] = transMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..63 (8‑398)

transMatrixCol0to15 = (8‑399)

{

{ 256 256 256 256 256 256 256 256 256 256 256 256 256 256 256 256 }

{ 362 361 359 357 353 349 344 338 331 323 315 306 296 285 274 262 }

{ 362 358 351 341 327 311 291 268 243 216 186 155 122 88 53 18 }

{ 361 353 338 315 285 250 208 163 114 62 9 −44 −97 −147 −194 −236 }

{ 360 346 319 280 230 171 105 35 −35 −105 −171 −230 −280 −319 −346 −360 }

{ 359 338 296 236 163 79 −9 −97 −178 −250 −306 −344 −361 −357 −331 −285 }

{ 358 327 268 186 88 −18 −122 −216 −291 −341 −362 −351 −311 −243 −155 −53 }

{ 357 315 236 130 9 −114 −223 −306 −353 −359 −323 −250 −147 −27 97 208 }

{ 355 301 201 71 −71 −201 −301 −355 −355 −301 −201 −71 71 201 301 355 }

{ 353 285 163 9 −147 −274 −349 −357 −296 −178 −27 130 262 344 359 306 }

{ 351 268 122 −53 −216 −327 −362 −311 −186 −18 155 291 358 341 243 88 }

{ 349 250 79 −114 −274 −357 −338 −223 −44 147 296 361 323 194 9 −178 }

{ 346 230 35 −171 −319 −360 −280 −105 105 280 360 319 171 −35 −230 −346 }

{ 344 208 −9 −223 −349 −338 −194 27 236 353 331 178 −44 −250 −357 −323 }

{ 341 186 −53 −268 −362 −291 −88 155 327 351 216 −18 −243 −358 −311 −122 }

{ 338 163 −97 −306 −357 −223 27 262 362 274 44 −208 −353 −315 −114 147 }

{ 334 139 −139 −334 −334 −139 139 334 334 139 −139 −334 −334 −139 139 334 }

{ 331 114 −178 −353 −296 −44 236 362 250 −27 −285 −357 −194 97 323 338 }

{ 327 88 −216 −362 −243 53 311 341 122 −186 −358 −268 18 291 351 155 }

{ 323 62 −250 −359 −178 147 353 274 −27 −306 −338 −97 223 362 208 −114 }

{ 319 35 −280 −346 −105 230 360 171 −171 −360 −230 105 346 280 −35 −319 }

{ 315 9 −306 −323 −27 296 331 44 −285 −338 −62 274 344 79 −262 −349 }

{ 311 −18 −327 −291 53 341 268 −88 −351 −243 122 358 216 −155 −362 −186 }

{ 306 −44 −344 −250 130 361 178 −208 −357 −97 274 331 9 −323 −285 79 }

{ 301 −71 −355 −201 201 355 71 −301 −301 71 355 201 −201 −355 −71 301 }

{ 296 −97 −361 −147 262 323 −44 −353 −194 223 344 9 −338 −236 178 357 }

{ 291 −122 −362 −88 311 268 −155 −358 −53 327 243 −186 −351 −18 341 216 }

{ 285 −147 −357 −27 344 194 −250 −315 97 362 79 −323 −236 208 338 −44 }

{ 280 −171 −346 35 360 105 −319 −230 230 319 −105 −360 −35 346 171 −280 }

{ 274 −194 −331 97 359 9 −357 −114 323 208 −262 −285 178 338 −79 −361 }

{ 268 −216 −311 155 341 −88 −358 18 362 53 −351 −122 327 186 −291 −243 }

{ 262 −236 −285 208 306 −178 −323 147 338 −114 −349 79 357 −44 −361 9 }

{ 256 −256 −256 256 256 −256 −256 256 256 −256 −256 256 256 −256 −256 256 }

{ 250 −274 −223 296 194 −315 −163 331 130 −344 −97 353 62 −359 −27 362 }

{ 243 −291 −186 327 122 −351 −53 362 −18 −358 88 341 −155 −311 216 268 }

{ 236 −306 −147 349 44 −362 62 344 −163 −296 250 223 −315 −130 353 27 }

{ 230 −319 −105 360 −35 −346 171 280 −280 −171 346 35 −360 105 319 −230 }

{ 223 −331 −62 361 −114 −306 262 178 −349 −9 353 −163 −274 296 130 −359 }

{ 216 −341 −18 351 −186 −243 327 53 −358 155 268 −311 −88 362 −122 −291 }

{ 208 −349 27 331 −250 −163 359 −79 −306 285 114 −362 130 274 −315 −62 }

{ 201 −355 71 301 −301 −71 355 −201 −201 355 −71 −301 301 71 −355 201 }

{ 194 −359 114 262 −338 27 315 −296 −62 349 −236 −147 362 −163 −223 353 }

{ 186 −362 155 216 −358 122 243 −351 88 268 −341 53 291 −327 18 311 }

{ 178 −362 194 163 −361 208 147 −359 223 130 −357 236 114 −353 250 97 }

{ 171 −360 230 105 −346 280 35 −319 319 −35 −280 346 −105 −230 360 −171 }

{ 163 −357 262 44 −315 331 −79 −236 361 −194 −130 349 −285 −9 296 −344 }

{ 155 −351 291 −18 −268 358 −186 −122 341 −311 53 243 −362 216 88 −327 }

{ 147 −344 315 −79 −208 359 −274 9 262 −361 223 62 −306 349 −163 −130 }

{ 139 −334 334 −139 −139 334 −334 139 139 −334 334 −139 −139 334 −334 139 }

{ 130 −323 349 −194 −62 285 −361 250 −9 −236 359 −296 79 178 −344 331 }

{ 122 −311 358 −243 18 216 −351 327 −155 −88 291 −362 268 −53 −186 341 }

{ 114 −296 362 −285 97 130 −306 361 −274 79 147 −315 359 −262 62 163 }

{ 105 −280 360 −319 171 35 −230 346 −346 230 −35 −171 319 −360 280 −105 }

{ 97 −262 353 −344 236 −62 −130 285 −359 331 −208 27 163 −306 362 −315 }

{ 88 −243 341 −358 291 −155 −18 186 −311 362 −327 216 −53 −122 268 −351 }

{ 79 −223 323 −362 331 −236 97 62 −208 315 −361 338 −250 114 44 −194 }

{ 71 −201 301 −355 355 −301 201 −71 −71 201 −301 355 −355 301 −201 71 }

{ 62 −178 274 −338 362 −344 285 −194 79 44 −163 262 −331 361 −349 296 }

{ 53 −155 243 −311 351 −362 341 −291 216 −122 18 88 −186 268 −327 358 }

{ 44 −130 208 −274 323 −353 362 −349 315 −262 194 −114 27 62 −147 223 }

{ 35 −105 171 −230 280 −319 346 −360 360 −346 319 −280 230 −171 105 −35 }

{ 27 −79 130 −178 223 −262 296 −323 344 −357 362 −359 349 −331 306 −274 }

{ 18 −53 88 −122 155 −186 216 −243 268 −291 311 −327 341 −351 358 −362 }

{ 9 −27 44 −62 79 −97 114 −130 147 −163 178 −194 208 −223 236 −250 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..63 (8‑400)

transMatrixCol16to31 = (8‑401)

{

{ 256 256 256 256 256 256 256 256 256 256 256 256 256 256 256 256 }

{ 250 236 223 208 194 178 163 147 130 114 97 79 62 44 27 9 }

{ −18 −53 −88 −122 −155 −186 −216 −243 −268 −291 −311 −327 −341 −351 −358 −362 }

{−274 −306 −331 −349 −359 −362 −357 −344 −323 −296 −262 −223 −178 −130 −79 −27 }

{−360 −346 −319 −280 −230 −171 −105 −35 35 105 171 230 280 319 346 360 }

{−223 −147 −62 27 114 194 262 315 349 362 353 323 274 208 130 44 }

{ 53 155 243 311 351 362 341 291 216 122 18 −88 −186 −268 −327 −358 }

{ 296 349 361 331 262 163 44 −79 −194 −285 −344 −362 −338 −274 −178 −62 }

{ 355 301 201 71 −71 −201 −301 −355 −355 −301 −201 −71 71 201 301 355 }

{ 194 44 −114 −250 −338 −361 −315 −208 −62 97 236 331 362 323 223 79 }

{ −88 −243 −341 −358 −291 −155 18 186 311 362 327 216 53 −122 −268 −351 }

{−315 −362 −306 −163 27 208 331 359 285 130 −62 −236 −344 −353 −262 −97 }

{−346 −230 −35 171 319 360 280 105 −105 −280 −360 −319 −171 35 230 346 }

{−163 62 262 359 315 147 −79 −274 −361 −306 −130 97 285 362 296 114 }

{ 122 311 358 243 18 −216 −351 −327 −155 88 291 362 268 53 −186 −341 }

{ 331 344 178 −79 −296 −359 −236 9 250 361 285 62 −194 −349 −323 −130 }

{ 334 139 −139 −334 −334 −139 139 334 334 139 −139 −334 −334 −139 139 334 }

{ 130 −163 −349 −306 −62 223 361 262 −9 −274 −359 −208 79 315 344 147 }

{−155 −351 −291 −18 268 358 186 −122 −341 −311 −53 243 362 216 −88 −327 }

{−344 −296 −9 285 349 130 −194 −361 −236 79 331 315 44 −262 −357 −163 }

{−319 −35 280 346 105 −230 −360 −171 171 360 230 −105 −346 −280 35 319 }

{ −97 250 353 114 −236 −357 −130 223 359 147 −208 −361 −163 194 362 178 }

{ 186 362 155 −216 −358 −122 243 351 88 −268 −341 −53 291 327 18 −311 }

{ 353 223 −163 −362 −147 236 349 62 −296 −315 27 338 262 −114 −359 −194 }

{ 301 −71 −355 −201 201 355 71 −301 −301 71 355 201 −201 −355 −71 301 }

{ 62 −315 −274 130 362 114 −285 −306 79 359 163 −250 −331 27 349 208 }

{−216 −341 18 351 186 −243 −327 53 358 155 −268 −311 88 362 122 −291 }

{−359 −130 296 274 −163 −353 −9 349 178 −262 −306 114 361 62 −331 −223 }

{−280 171 346 −35 −360 −105 319 230 −230 −319 105 360 35 −346 −171 280 }

{ −27 353 130 −315 −223 250 296 −163 −344 62 362 44 −349 −147 306 236 }

{ 243 291 −186 −327 122 351 −53 −362 −18 358 88 −341 −155 311 216 −268 }

{ 362 27 −359 −62 353 97 −344 −130 331 163 −315 −194 296 223 −274 −250 }

{ 256 −256 −256 256 256 −256 −256 256 256 −256 −256 256 256 −256 −256 256 }

{ −9 −361 44 357 −79 −349 114 338 −147 −323 178 306 −208 −285 236 262 }

{−268 −216 311 155 −341 −88 358 18 −362 53 351 −122 −327 186 291 −243 }

{−361 79 338 −178 −285 262 208 −323 −114 357 9 −359 97 331 −194 −274 }

{−230 319 105 −360 35 346 −171 −280 280 171 −346 −35 360 −105 −319 230 }

{ 44 338 −208 −236 323 79 −362 97 315 −250 −194 344 27 −357 147 285 }

{ 291 122 −362 88 311 −268 −155 358 −53 −327 243 186 −351 18 341 −216 }

{ 357 −178 −236 338 9 −344 223 194 −353 44 323 −262 −147 361 −97 −296 }

{ 201 −355 71 301 −301 −71 355 −201 −201 355 −71 −301 301 71 −355 201 }

{ −79 −285 323 9 −331 274 97 −357 208 178 −361 130 250 −344 44 306 }

{−311 −18 327 −291 −53 341 −268 −88 351 −243 −122 358 −216 −155 362 −186 }

{−349 262 79 −344 274 62 −338 285 44 −331 296 27 −323 306 9 −315 }

{−171 360 −230 −105 346 −280 −35 319 −319 35 280 −346 105 230 −360 171 }

{ 114 208 −362 223 97 −338 306 −27 −274 353 −147 −178 359 −250 −62 323 }

{ 327 −88 −216 362 −243 −53 311 −341 122 186 −358 268 18 −291 351 −155 }

{ 338 −323 97 194 −357 285 −27 −250 362 −236 −44 296 −353 178 114 −331 }

{ 139 −334 334 −139 −139 334 −334 139 139 −334 334 −139 −139 334 −334 139 }

{−147 −114 315 −353 208 44 −274 362 −262 27 223 −357 306 −97 −163 338 }

{−341 186 53 −268 362 −291 88 155 −327 351 −216 −18 243 −358 311 −122 }

{−323 357 −250 44 178 −331 353 −236 27 194 −338 349 −223 9 208 −344 }

{−105 280 −360 319 −171 −35 230 −346 346 −230 35 171 −319 360 −280 105 }

{ 178 9 −194 323 −361 296 −147 −44 223 −338 357 −274 114 79 −250 349 }

{ 351 −268 122 53 −216 327 −362 311 −186 18 155 −291 358 −341 243 −88 }

{ 306 −359 344 −262 130 27 −178 296 −357 349 −274 147 9 −163 285 −353 }

{ 71 −201 301 −355 355 −301 201 −71 −71 201 −301 355 −355 301 −201 71 }

{−208 97 27 −147 250 −323 359 −353 306 −223 114 9 −130 236 −315 357 }

{−358 327 −268 186 −88 −18 122 −216 291 −341 362 −351 311 −243 155 −53 }

{−285 331 −357 361 −344 306 −250 178 −97 9 79 −163 236 −296 338 −359 }

{ −35 105 −171 230 −280 319 −346 360 −360 346 −319 280 −230 171 −105 35 }

{ 236 −194 147 −97 44 9 −62 114 −163 208 −250 285 −315 338 −353 361 }

{ 362 −358 351 −341 327 −311 291 −268 243 −216 186 −155 122 −88 53 −18 }

{ 262 −274 285 −296 306 −315 323 −331 338 −344 349 −353 357 −359 361 −362 }

},

transMatrix[ m ][ n ] = ( n & 1 ? −1 : 1 ) \* transMatrixCol16to31[47 −m ][ n ] (8‑402)  
 with m =32..47, n = 0..63

transMatrix[ m ][ n ] = ( n & 1 ? −1 : 1 ) \* transMatrixCol0to15[63 −m ][ n ] (8‑403)  
 with m =48..63, n = 0..63

* Otherwise, if trType is equal to 1 and nTbs is equal to 4, the following applies:

transMatrix[ m ][ n ] = (8‑404)

{

{ 117 219 296 336 }

{ 296 296 0 −296 }

{ 336 −117 −296 219 }

{ 219 −336 296 −117 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 8, the following applies:

transMatrix[ m ][ n ] = (8‑405)

{

{ 65 127 185 237 280 314 338 350 }

{ 185 314 350 280 127 −65 −237 −338 }

{ 280 338 127 −185 −350 −237 65 314 }

{ 338 185 −237 −314 65 350 127 −280 }

{ 350 −65 −338 127 314 −185 −280 237 }

{ 314 −280 −65 338 −237 −127 350 −185 }

{ 237 −350 280 −65 −185 338 −314 127 }

{ 127 −237 314 −350 338 −280 185 −65 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 16, the following applies:

transMatrix[ m ][ n ] = (8‑406)

{

{ 34 67 100 133 163 193 220 246 269 290 309 324 337 346 353 356 }

{ 100 193 269 324 353 353 324 269 193 100 0 −100 −193 −269 −324 −353 }

{ 163 290 353 337 246 100 −67 −220 −324 −356 −309 −193 −34 133 269 346 }

{ 220 346 324 163 −67 −269 −356 −290 −100 133 309 353 246 34 −193 −337 }

{ 269 353 193 −100 −324 −324 −100 193 353 269 0 −269 −353 −193 100 324 }

{ 309 309 0 −309 −309 0 309 309 0 −309 −309 0 309 309 0 −309 }

{ 337 220 −193 −346 −34 324 246 −163 −353 −67 309 269 −133 −356 −100 290 }

{ 353 100 −324 −193 269 269 −193 −324 100 353 0 −353 −100 324 193 −269 }

{ 356 −34 −353 67 346 −100 −337 133 324 −163 −309 193 290 −220 −269 246 }

{ 346 −163 −269 290 133 −353 34 337 −193 −246 309 100 −356 67 324 −220 }

{ 324 −269 −100 353 −193 −193 353 −100 −269 324 0 −324 269 100 −353 193 }

{ 290 −337 100 220 −356 193 133 −346 269 34 −309 324 −67 −246 353 −163 }

{ 246 −356 269 −34 −220 353 −290 67 193 −346 309 −100 −163 337 −324 133 }

{ 193 −324 353 −269 100 100 −269 353 −324 193 0 −193 324 −353 269 −100 }

{ 133 −246 324 −356 337 −269 163 −34 −100 220 −309 353 −346 290 −193 67 }

{ 67 −133 193 −246 290 −324 346 −356 353 −337 309 −269 220 −163 100 −34 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 32, the following applies:

transMatrix[ m ][ n ] = transMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..31 (8‑407)

transMatrixCol0to15 = (8‑408)

{

{ 17 35 52 69 86 103 119 135 151 167 182 197 211 225 238 251 }

{ 52 103 151 197 238 275 305 329 347 357 359 354 342 322 296 263 }

{ 86 167 238 296 336 357 357 336 296 238 167 86 0 −86 −167 −238 }

{ 119 225 305 351 357 322 251 151 35 −86 −197 −285 −342 −359 −336 −275 }

{ 151 275 347 354 296 182 35 −119 −251 −336 −358 −314 −211 −69 86 225 }

{ 182 314 359 305 167 −17 −197 −322 −358 −296 −151 35 211 329 357 285 }

{ 211 342 342 211 0 −211 −342 −342 −211 0 211 342 342 211 0 −211 }

{ 238 357 296 86 −167 −336 −336 −167 86 296 357 238 0 −238 −357 −296 }

{ 263 358 225 −52 −296 −351 −182 103 322 336 135 −151 −342 −314 −86 197 }

{ 285 347 135 −182 −357 −251 52 314 329 86 −225 −359 −211 103 336 305 }

{ 305 322 35 −285 −336 −69 263 347 103 −238 −354 −135 211 358 167 −182 }

{ 322 285 −69 −347 −238 135 358 182 −197 −357 −119 251 342 52 −296 −314 }

{ 336 238 −167 −357 −86 296 296 −86 −357 −167 238 336 0 −336 −238 167 }

{ 347 182 −251 −314 86 359 103 −305 −263 167 351 17 −342 −197 238 322 }

{ 354 119 −314 −225 238 305 −135 −351 17 357 103 −322 −211 251 296 −151 }

{ 358 52 −351 −103 336 151 −314 −197 285 238 −251 −275 211 305 −167 −329 }

{ 359 −17 −358 35 357 −52 −354 69 351 −86 −347 103 342 −119 −336 135 }

{ 357 −86 −336 167 296 −238 −238 296 167 −336 −86 357 0 −357 86 336 }

{ 351 −151 −285 275 167 −347 −17 354 −135 −296 263 182 −342 −35 357 −119 }

{ 342 −211 −211 342 0 −342 211 211 −342 0 342 −211 −211 342 0 −342 }

{ 329 −263 −119 358 −167 −225 347 −52 −305 296 69 −351 211 182 −357 103 }

{ 314 −305 −17 322 −296 −35 329 −285 −52 336 −275 −69 342 −263 −86 347 }

{ 296 −336 86 238 −357 167 167 −357 238 86 −336 296 0 −296 336 −86 }

{ 275 −354 182 119 −336 314 −69 −225 359 −238 −52 305 −342 135 167 −351 }

{ 251 −359 263 −17 −238 358 −275 35 225 −357 285 −52 −211 354 −296 69 }

{ 225 −351 322 −151 −86 285 −359 275 −69 −167 329 −347 211 17 −238 354 }

{ 197 −329 354 −263 86 119 −285 358 −314 167 35 −225 342 −347 238 −52 }

{ 167 −296 357 −336 238 −86 −86 238 −336 357 −296 167 0 −167 296 −357 }

{ 135 −251 329 −359 336 −263 151 −17 −119 238 −322 358 −342 275 −167 35 }

{ 103 −197 275 −329 357 −354 322 −263 182 −86 −17 119 −211 285 −336 358 }

{ 69 −135 197 −251 296 −329 351 −359 354 −336 305 −263 211 −151 86 −17 }

{ 35 −69 103 −135 167 −197 225 −251 275 −296 314 −329 342 −351 357 −359 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..31 (8‑409)

transMatrixCol16to31 = (8‑410)

{

{ 263 275 285 296 305 314 322 329 336 342 347 351 354 357 358 359 }

{ 225 182 135 86 35 −17 −69 −119 −167 −211 −251 −285 −314 −336 −351 −358 }

{−296 −336 −357 −357 −336 −296 −238 −167 −86 0 86 167 238 296 336 357 }

{−182 −69 52 167 263 329 358 347 296 211 103 −17 −135 −238 −314 −354 }

{ 322 359 329 238 103 −52 −197 −305 −357 −342 −263 −135 17 167 285 351 }

{ 135 −52 −225 −336 −354 −275 −119 69 238 342 351 263 103 −86 −251 −347 }

{−342 −342 −211 0 211 342 342 211 0 −211 −342 −342 −211 0 211 342 }

{ −86 167 336 336 167 −86 −296 −357 −238 0 238 357 296 86 −167 −336 }

{ 354 285 35 −238 −359 −251 17 275 357 211 −69 −305 −347 −167 119 329 }

{ 35 −263 −354 −167 151 351 275 −17 −296 −342 −119 197 358 238 −69 −322 }

{−359 −197 151 357 225 −119 −351 −251 86 342 275 −52 −329 −296 17 314 }

{ 17 329 275 −86 −351 −225 151 359 167 −211 −354 −103 263 336 35 −305 }

{ 357 86 −296 −296 86 357 167 −238 −336 0 336 238 −167 −357 −86 296 }

{ −69 −358 −119 296 275 −151 −354 −35 336 211 −225 −329 52 357 135 −285 }

{−347 35 358 86 −329 −197 263 285 −167 −342 52 359 69 −336 −182 275 }

{ 119 347 −69 −357 17 359 35 −354 −86 342 135 −322 −182 296 225 −263 }

{ 329 −151 −322 167 314 −182 −305 197 296 −211 −285 225 275 −238 −263 251 }

{−167 −296 238 238 −296 −167 336 86 −357 0 357 −86 −336 167 296 −238 }

{−305 251 197 −336 −52 358 −103 −314 238 211 −329 −69 359 −86 −322 225 }

{ 211 211 −342 0 342 −211 −211 342 0 −342 211 211 −342 0 342 −211 }

{ 275 −322 −17 336 −251 −135 359 −151 −238 342 −35 −314 285 86 −354 197 }

{−251 −103 351 −238 −119 354 −225 −135 357 −211 −151 358 −197 −167 359 −182 }

{−238 357 −167 −167 357 −238 −86 336 −296 0 296 −336 86 238 −357 167 }

{ 285 −17 −263 357 −197 −103 329 −322 86 211 −358 251 35 −296 347 −151 }

{ 197 −351 305 −86 −182 347 −314 103 167 −342 322 −119 −151 336 −329 135 }

{−314 135 103 −296 358 −263 52 182 −336 342 −197 −35 251 −357 305 −119 }

{−151 305 −359 296 −135 −69 251 −351 336 −211 17 182 −322 357 −275 103 }

{ 336 −238 86 86 −238 336 −357 296 −167 0 167 −296 357 −336 238 −86 }

{ 103 −225 314 −357 347 −285 182 −52 −86 211 −305 354 −351 296 −197 69 }

{−351 314 −251 167 −69 −35 135 −225 296 −342 359 −347 305 −238 151 −52 }

{ −52 119 −182 238 −285 322 −347 358 −357 342 −314 275 −225 167 −103 35 }

{ 358 −354 347 −336 322 −305 285 −263 238 −211 182 −151 119 −86 52 −17 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 4, the following applies:

transMatrix[ m ][ n ] = (8‑411)

{

{ 336 296 219 117 }

{ 296 0 −296 −296 }

{ 219 −296 −117 336 }

{ 117 −296 336 −219 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 8, the following applies:

transMatrix[ m ][ n ] = (8‑412)

{

{ 350 338 314 280 237 185 127 65 }

{ 338 237 65 −127 −280 −350 −314 −185 }

{ 314 65 −237 −350 −185 127 338 280 }

{ 280 −127 −350 −65 314 237 −185 −338 }

{ 237 −280 −185 314 127 −338 −65 350 }

{ 185 −350 127 237 −338 65 280 −314 }

{ 127 −314 338 −185 −65 280 −350 237 }

{ 65 −185 280 −338 350 −314 237 −127 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 16, the following applies:

transMatrix[ m ][ n ] = (8‑413)

{

{ 356 353 346 337 324 309 290 269 246 220 193 163 133 100 67 34 }

{ 353 324 269 193 100 0 −100 −193 −269 −324 −353 −353 −324 −269 −193 −100 }

{ 346 269 133 −34 −193 −309 −356 −324 −220 −67 100 246 337 353 290 163 }

{ 337 193 −34 −246 −353 −309 −133 100 290 356 269 67 −163 −324 −346 −220 }

{ 324 100 −193 −353 −269 0 269 353 193 −100 −324 −324 −100 193 353 269 }

{ 309 0 −309 −309 0 309 309 0 −309 −309 0 309 309 0 −309 −309 }

{ 290 −100 −356 −133 269 309 −67 −353 −163 246 324 −34 −346 −193 220 337 }

{ 269 −193 −324 100 353 0 −353 −100 324 193 −269 −269 193 324 −100 −353 }

{ 246 −269 −220 290 193 −309 −163 324 133 −337 −100 346 67 −353 −34 356 }

{ 220 −324 −67 356 −100 −309 246 193 −337 −34 353 −133 −290 269 163 −346 }

{ 193 −353 100 269 −324 0 324 −269 −100 353 −193 −193 353 −100 −269 324 }

{ 163 −353 246 67 −324 309 −34 −269 346 −133 −193 356 −220 −100 337 −290 }

{ 133 −324 337 −163 −100 309 −346 193 67 −290 353 −220 −34 269 −356 246 }

{ 100 −269 353 −324 193 0 −193 324 −353 269 −100 −100 269 −353 324 −193 }

{ 67 −193 290 −346 353 −309 220 −100 −34 163 −269 337 −356 324 −246 133 }

{ 34 −100 163 −220 269 −309 337 −353 356 −346 324 −290 246 −193 133 −67 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 32, the following applies:

transMatrix[ m ][ n ] = transMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..31 (8‑414)

transMatrixCol0to15 = (8‑415)

{

{ 359 358 357 354 351 347 342 336 329 322 314 305 296 285 275 263 }

{ 358 351 336 314 285 251 211 167 119 69 17 −35 −86 −135 −182 −225 }

{ 357 336 296 238 167 86 0 −86 −167 −238 −296 −336 −357 −357 −336 −296 }

{ 354 314 238 135 17 −103 −211 −296 −347 −358 −329 −263 −167 −52 69 182 }

{ 351 285 167 17 −135 −263 −342 −357 −305 −197 −52 103 238 329 359 322 }

{ 347 251 86 −103 −263 −351 −342 −238 −69 119 275 354 336 225 52 −135 }

{ 342 211 0 −211 −342 −342 −211 0 211 342 342 211 0 −211 −342 −342 }

{ 336 167 −86 −296 −357 −238 0 238 357 296 86 −167 −336 −336 −167 86 }

{ 329 119 −167 −347 −305 −69 211 357 275 17 −251 −359 −238 35 285 354 }

{ 322 69 −238 −358 −197 119 342 296 17 −275 −351 −151 167 354 263 −35 }

{ 314 17 −296 −329 −52 275 342 86 −251 −351 −119 225 357 151 −197 −359 }

{ 305 −35 −336 −263 103 354 211 −167 −359 −151 225 351 86 −275 −329 −17 }

{ 296 −86 −357 −167 238 336 0 −336 −238 167 357 86 −296 −296 86 357 }

{ 285 −135 −357 −52 329 225 −211 −336 35 354 151 −275 −296 119 358 69 }

{ 275 −182 −336 69 359 52 −342 −167 285 263 −197 −329 86 358 35 −347 }

{ 263 −225 −296 182 322 −135 −342 86 354 −35 −359 −17 357 69 −347 −119 }

{ 251 −263 −238 275 225 −285 −211 296 197 −305 −182 314 167 −322 −151 329 }

{ 238 −296 −167 336 86 −357 0 357 −86 −336 167 296 −238 −238 296 167 }

{ 225 −322 −86 359 −69 −329 211 238 −314 −103 358 −52 −336 197 251 −305 }

{ 211 −342 0 342 −211 −211 342 0 −342 211 211 −342 0 342 −211 −211 }

{ 197 −354 86 285 −314 −35 342 −238 −151 359 −135 −251 336 −17 −322 275 }

{ 182 −359 167 197 −358 151 211 −357 135 225 −354 119 238 −351 103 251 }

{ 167 −357 238 86 −336 296 0 −296 336 −86 −238 357 −167 −167 357 −238 }

{ 151 −347 296 −35 −251 358 −211 −86 322 −329 103 197 −357 263 17 −285 }

{ 135 −329 336 −151 −119 322 −342 167 103 −314 347 −182 −86 305 −351 197 }

{ 119 −305 357 −251 35 197 −342 336 −182 −52 263 −358 296 −103 −135 314 }

{ 103 −275 357 −322 182 17 −211 336 −351 251 −69 −135 296 −359 305 −151 }

{ 86 −238 336 −357 296 −167 0 167 −296 357 −336 238 −86 −86 238 −336 }

{ 69 −197 296 −351 354 −305 211 −86 −52 182 −285 347 −357 314 −225 103 }

{ 52 −151 238 −305 347 −359 342 −296 225 −135 35 69 −167 251 −314 351 }

{ 35 −103 167 −225 275 −314 342 −357 358 −347 322 −285 238 −182 119 −52 }

{ 17 −52 86 −119 151 −182 211 −238 263 −285 305 −322 336 −347 354 −358 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..31 (8‑416)

transMatrixCol16to31 = (8‑417)

{

{ 251 238 225 211 197 182 167 151 135 119 103 86 69 52 35 17 }

{−263 −296 −322 −342 −354 −359 −357 −347 −329 −305 −275 −238 −197 −151 −103 −52 }

{−238 −167 −86 0 86 167 238 296 336 357 357 336 296 238 167 86 }

{ 275 336 359 342 285 197 86 −35 −151 −251 −322 −357 −351 −305 −225 −119 }

{ 225 86 −69 −211 −314 −358 −336 −251 −119 35 182 296 354 347 275 151 }

{−285 −357 −329 −211 −35 151 296 358 322 197 17 −167 −305 −359 −314 −182 }

{−211 0 211 342 342 211 0 −211 −342 −342 −211 0 211 342 342 211 }

{ 296 357 238 0 −238 −357 −296 −86 167 336 336 167 −86 −296 −357 −238 }

{ 197 −86 −314 −342 −151 135 336 322 103 −182 −351 −296 −52 225 358 263 }

{−305 −336 −103 211 359 225 −86 −329 −314 −52 251 357 182 −135 −347 −285 }

{−182 167 358 211 −135 −354 −238 103 347 263 −69 −336 −285 35 322 305 }

{ 314 296 −52 −342 −251 119 357 197 −182 −358 −135 238 347 69 −285 −322 }

{ 167 −238 −336 0 336 238 −167 −357 −86 296 296 −86 −357 −167 238 336 }

{−322 −238 197 342 −17 −351 −167 263 305 −103 −359 −86 314 251 −182 −347 }

{−151 296 251 −211 −322 103 357 17 −351 −135 305 238 −225 −314 119 354 }

{ 329 167 −305 −211 275 251 −238 −285 197 314 −151 −336 103 351 −52 −358 }

{ 135 −336 −119 342 103 −347 −86 351 69 −354 −52 357 35 −358 −17 359 }

{−336 −86 357 0 −357 86 336 −167 −296 238 238 −296 −167 336 86 −357 }

{−119 357 −35 −342 182 263 −296 −135 354 −17 −347 167 275 −285 −151 351 }

{ 342 0 −342 211 211 −342 0 342 −211 −211 342 0 −342 211 211 −342 }

{ 103 −357 182 211 −351 69 296 −305 −52 347 −225 −167 358 −119 −263 329 }

{−347 86 263 −342 69 275 −336 52 285 −329 35 296 −322 17 305 −314 }

{ −86 336 −296 0 296 −336 86 238 −357 167 167 −357 238 86 −336 296 }

{ 351 −167 −135 342 −305 52 238 −359 225 69 −314 336 −119 −182 354 −275 }

{ 69 −296 354 −211 −52 285 −357 225 35 −275 358 −238 −17 263 −359 251 }

{−354 238 −17 −211 347 −329 167 69 −275 359 −285 86 151 −322 351 −225 }

{ −52 238 −347 342 −225 35 167 −314 358 −285 119 86 −263 354 −329 197 }

{ 357 −296 167 0 −167 296 −357 336 −238 86 86 −238 336 −357 296 −167 }

{ 35 −167 275 −342 358 −322 238 −119 −17 151 −263 336 −359 329 −251 135 }

{−358 336 −285 211 −119 17 86 −182 263 −322 354 −357 329 −275 197 −103 }

{ −17 86 −151 211 −263 305 −336 354 −359 351 −329 296 −251 197 −135 69 }

{ 359 −357 351 −342 329 −314 296 −275 251 −225 197 −167 135 −103 69 −35 }

},

### Picture reconstruction process

[Ed. (BB): tbd]

## In-loop Filter Process

### General

[Ed. (SL): Place holder, assuming we will have more than one in-loop filter and one of them is deblocking filter.]

The two in-loop filters, namely deblocking filter and adaptive loop filter, are applied as specified by the following ordered steps:

1. For the deblocking filter, the following applies:

– The deblocking filter process as specified in clause 8.5.2.1 is invoked with the reconstructed picture sample arrays SL, SCb and SCr as inputs, and the modified reconstructed picture sample arrays S′L, S′Cb and S′Cr after deblocking as outputs.

– The array S′L, S′Cb and S′Cr are assigned to the arrays SL, SCb and SCr (which represent the decoded picture), respectively.

1. When sps\_alf\_enabled\_flag is equal to 1, the following applies:

– The adaptive loop filter process as specified in clause 8.5.3.1 is invoked with the reconstructed picture sample arrays SL, SCb and SCr as inputs, and the modified reconstructed picture sample arrays S′L, S′Cb and S′Cr after sample adaptive offset as outputs.

– The arrays S′L, S′Cb and S′Cr are assigned to the arrays SL, SCb and SCr (which represent the decoded picture), respectively.

### Deblocking filter process

#### General

[Ed. (SL): Place holder, assuming we will have a deblocking filter.]

### Adaptive loop filter process

#### General

Inputs of this process are the reconstructed picture sample arrays prior to adaptive loop filter recPictureL, recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture sample arrays after adaptive loop filter alfPictureL, alfPictureCb and alfPictureCr.

The sample values in the modified reconstructed picture sample arrays after adaptive loop filter alfPictureL, alfPictureCb and alfPictureCr are initially set equal to the sample values in the reconstructed picture sample arrays prior to adaptive loop filter recPictureL, recPictureCb and recPictureCr.

When slice\_alf\_enabled\_flag is equal to 1, the following applies:

* + The derivation process for ALF luma coefficients as specified in clause 8.5.3.2 is invoked and the output is assigned to the array of luma coefficients cL.
  + If alf\_chroma\_idc is not equal to 0, the derivation process for ALF chroma coefficients as specified in clause 8.5.3.3 is invoked and the output is assigned to the array of chroma coefficients cC.
  + For every coding tree unit with luma coding tree block location ( rx, ry ), where rx = 0..PicWidthInCtbs − 1 and ry = 0..PicHeightInCtbs − 1, the following applies:
* When alf\_ctb\_flag[ 0 ][ rx ][ ry ] is equal to 1, the coding tree block filtering process for luma samples as specified in clause 8.5.3.4 is invoked with recPictureL, alfPictureL, the luma coding tree block location ( xCtb, yCtb ) set equal to ( rx  <<  CtbLog2SizeY, ry  <<  CtbLog2SizeY ), and the array of luma coefficients cL as inputs, and the output is the modified filtered picture alfPictureL.
* When alf\_ctb\_flag[ 1 ][ rx ][ ry ] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.5.3.6 is invoked with recPicture set equal to recPictureCb, alfPicture set equal to alfPictureCb, the chroma coding tree block location ( xCtbC, yCtbC ) set equal to ( rx  <<  ( CtbLog2SizeY − 1 ), ry  <<  ( CtbLog2SizeY − 1 ) ) and the array of chroma coefficients cC as inputs, and the output is the modified filtered picture alfPictureCb.
* When alf\_ctb\_flag[ 2 ][ rx ][ ry ] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.5.3.6 is invoked with recPicture set equal to recPictureCr, alfPicture set equal to alfPictureCr, the chroma coding tree block location ( xCtbC, yCtbC ) set equal to ( rx  <<  ( CtbLog2SizeY − 1 ), ry  <<  ( CtbLog2SizeY − 1 ) ) and the array of chroma coefficients cC as inputs, and the output is the modified filtered picture alfPictureCr.

#### Derivation process for ALF luma coefficients

Outputs of this process is a luma filter coefficient array cL.

The variable filterCoefficients[ sigFiltIdx ][ j ] with sigFiltIdx = 0..alf\_luma\_num\_filters\_signalled\_minus1, j = 0..11 is initialized as follows:

filterCoefficients[ sigFiltIdx ][ j ] = alf\_luma\_coeff\_delta\_abs[ sigFiltIdx ][ j ] \* (8‑418)  
 ( 1 − 2 \* alf\_luma\_coeff\_delta\_sign[ sigFiltIdx ][ j ] )

When alf\_luma\_coeff\_delta\_prediction\_flag is equal 1, filterCoefficients[ sigFiltIdx ][ j ] with sigFiltIdx = 1..alf\_luma\_num\_filters\_signalled\_minus1 and j = 0..11 are modified as follows:

filterCoefficients[ sigFiltIdx ][ j ] += filterCoefficients[ sigFiltIdx − 1 ][ j ]  (8‑419)

The binary array filter\_pattern[ j ] with j=0..11 determines which coefficients are available based on 7x7 filter pattern and is defined as follows:

filterPattern[ ] = { 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1 } (8‑420)

The variable k is set equal to 0, and the luma filter coefficients cL with elements cL[ filtIdx ][ j ], with filtIdx = 0..NumAlfFilters − 1 and j = 0..11 are derived as follows

* + If filterPattern[ j ] is equal 1 or alf\_luma\_type\_flag is equal 0, the following applies:

cL[ filtIdx ][ j ] = filterCoefficients[ alf\_luma\_coeff\_delta\_idx[ filtIdx ] ][ k ] (8‑421)  
k++

* + Otherwise, the following applies:

cL[ filtIdx ][ j ] = 0 (8‑422)

The last filter coefficients cL[ filtIdx ][ j ] for j = 12 and filtIdx = 0..NumAlfFilters − 1 are derived as follows:

cL[ filtIdx ][ 12 ] = 512 − Σk ( cL[ filtIdx ][ k ]  <<  1 ), with k = 0..11 (8‑423)

#### Derivation process for ALF chroma coefficients

Outputs of this process are the 7 chroma filter coefficients cc.

The chroma filter coefficients cC with elements cC[ j ], j = 0..5 are derived as follows:

cC[ j ] = alf\_chroma\_coeff\_abs[ j ] \* ( 1 − 2 \* alf\_chroma\_coeff\_sign[ j ] ) (8‑424)

The last filter coefficient for j = 6 is derived as follows:

cC[ 6 ] = 512 − Σk ( cC[ k ]  <<  1 ), with k = 0..5 (8‑425)

#### Coding tree block filtering process for luma samples

Inputs of this process are:

* a reconstructed luma picture sample array recPictureL prior to the adaptive loop filtering process,
* a filtered reconstructed luma picture sample array alfPictureL,
* a luma location ( xCtb, yCtb ) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture,
* an array of luma filter coefficients cL[ filtIdx ][ j ] with filtIdx = 0..NumAlfFilters − 1 and j = 0..12.

Output of this process is the modified filtered reconstructed luma picture sample array alfPictureL.

The derivation process for filter index clause 8.5.3.5 is invoked with the location ( xCtb, yCtb ) and the reconstructed luma picture sample array recPictureL as inputs, and filtIdx[ x ][ y ] and transposeIdx[ x ][ y ] with x, y = 0..CtbSizeY − 1 as outputs.

For the derivation of the filtered reconstructed luma samples alfPictureL[ x ][ y ], each reconstructed luma sample inside the current luma coding tree block recPictureL[ x ][ y ] is filtered as follows with x, y = 0..CtbSizeY − 1:

* + The array of luma filter coefficients f[ j ] corresponding to the filter specified by filtIdx[ x ][ y ] is derived as follows with j = 0..12:

f[ j ] = cL[ filtIdx[ x ][ y ] ][ j ] (8‑426)

* + The luma filter coefficients filterCoeff are derived depending on transposeIdx[ x ][ y ] as follows:
  + If transposeIndex[ x ][ y ] = = 1,

filterCoeff[ ] = { f[9], f[4], f[10], f[8], f[1], f[5], f[11], f[7], f[3], f[0], f[2], f[6], f[12] } (8‑427)

* + Otherwise, if transposeIndex[ x ][ y ] = = 2,

filterCoeff[ ] = { f[0], f[3], f[2], f[1], f[8], f[7], f[6], f[5], f[4], f[9], f[10], f[11], f[12] } (8‑428)

* + Otherwise, if transposeIndex[ x ][ y ] = = 3,

filterCoeff[ ] = { f[9], f[8], f[10], f[4], f[3], f[7], f[11], f[5], f[1], f[0], f[2], f[6], f[12] } (8‑429)

* + Otherwise,

filterCoeff[ ] = { f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10], f[11], f[12] } (8‑430)

* + The locations ( hx, vy ) for each of the corresponding luma samples ( x, y ) inside the given array recPicture of luma samples are derived as follows:

hx = Clip3( 0, pic\_width\_in\_luma\_samples − 1, xCtb + x ) (8‑431)

vy = Clip3( 0, pic\_height\_in\_luma\_samples − 1, yCtb + y ) (8‑432)

* + The variable sum is set to 0 and modifed using the following ordered steps:

1. When filterType is equal 0, the following applies:

sum = filterCoeff[ 0 ] \* ( recPictureL[ hx, vy + 3 ] + recPictureL[ hx, vy − 3 ] ) +   
 filterCoeff[ 1 ] \* ( recPictureL[ hx + 1, vy + 2 ] + recPictureL[ hx − 1, vy − 2 ] ) +   
 filterCoeff[ 3 ] \* ( recPictureL[ hx − 1, vy + 2 ] + recPictureL[ hx + 1, vy − 2 ] ) + (8‑433)  
 filterCoeff[ 4 ] \* ( recPictureL[ hx + 2, vy + 1 ] + recPictureL[ hx − 2, vy − 1 ] ) +   
 filterCoeff[ 8 ] \* ( recPictureL[ hx − 2, vy + 1 ] + recPictureL[ hx + 2, vy − 1 ] ) +   
 filterCoeff[ 9 ] \* ( recPictureL[ hx + 3, vy ] + recPictureL[ hx − 3, vy ] )

1. The following operations apply:

sum += filterCoeff[ 2 ]    \* ( recPictureL[ hx, vy + 2 ] + recPictureL[ hx, vy − 2 ] ) +   
 filterCoeff[ 5 ]   \* ( recPictureL[ hx + 1, vy + 1 ] + recPictureL[ hx − 1, vy − 1 ] ) +   
 filterCoeff[ 6 ]   \* ( recPictureL[ hx, vy + 1 ] + recPictureL[ hx, vy − 1 ] ) + (8‑434)  
 filterCoeff[ 7 ]   \* ( recPictureL[ hx − 1, vy + 1 ] + recPictureL[ hx + 1, vy − 1 ] ) +   
 filterCoeff[ 10 ] \* ( recPictureL[ hx + 2, vy ] + recPictureL[ hx − 2, vy ] ) +   
 filterCoeff[ 11 ] \* ( recPictureL[ hx + 1, vy ] + recPictureL[ hx − 1, vy ] ) +   
 filterCoeff[ 12 ] \*   recPictureL[ hx, vy ]

sum = ( sum + 256 ) >> 9 (8‑435)

* + The modified filtered reconstructed luma picture sample alfPictureL[ xCtb + x ][ yCtb + y ] is derived as follows:

alfPictureL[ xCtb + x ][ yCtb + y ] = Clip3( 0, ( 1 << BitDepthY ) − 1, sum ) (8‑436)

#### Derivation process for ALF transpose and filter index for luma samples

Inputs of this process are:

* a luma location ( xCtb, yCtb ) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture,
* a reconstructed luma picture sample array recPictureL prior to the adaptive loop filtering process.

Outputs of this process are

* the classification filter index array filtIdx[ x ][ y ] with x, y = 0..CtbSizeY − 1,
* the transpose index array transposeIdx[ x ][ y ] with x, y = 0..CtbSizeY − 1.

The locations ( hx, vy ) for each of the corresponding luma samples ( x, y ) inside the given array recPicture of luma samples are derived as follows:

hx = Clip3( 0, pic\_width\_in\_luma\_samples − 1, x ) (8‑437)

vy = Clip3( 0, pic\_height\_in\_luma\_samples − 1, y ) (8‑438)

The classification filter index array filtIdx and the transpose index array transposeIdx are derived by the following ordered steps:

1. The variables varTempH[ x ][ y ], varTempV[ x ][ y ], varTempD0[ x ][ y ] and varTempD1[ x ][ y ] with x, y = − 2..CtbSizeY + 1 are derived as follows:

varTempH[ x ][ y ] = Abs(  ( recPicture[ hxC+x, vyC+y ] << 1 ) − recPicture[ hxC+x−1, vyC+y ] −  (8‑439)  
  recPicture[ hxC+x+1, vyC+y ]  )

varTempV[ x ][ y ] = Abs(  ( recPicture[ hxC+x, vyC+y ] << 1 ) − recPicture[ hxC+x, vyC+y−1 ] −  (8‑440)  
  recPicture[ hxC+x, vyC+y+1 ]  )

varTempD0[ x ][ y ] = Abs(  ( recPicture[ hxC+x, vyC+y ] << 1 ) − recPicture[ hxC+x−1, vyC+y−1 ] −  (8‑441)  
 recPicture[ hxC+x+1, vyC+y+1 ]  )

varTempD1[ x ][ y ] = Abs(  ( recPicture[ hxC+x, vyC+y ] << 1 ) − recPicture[ hxC+x+1, vyC+y−1 ] −  (8‑442)  
 recPicture[ hxC+x−1, vyC+y+1 ]  )

1. The variables varTempH1[ x ][ y ], varTempV1[ x ][ y ], varTempD01[ x ][ y ], varTempD11[ x ][ y ] and varTemp[ x ][ y ] with x, y = 0..( CtbSizeY − 1 ) >> 2 are derived as follows:

varTempH1[ x ][ y ] = ΣiΣj varTempH[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑443)

varTempV1[ x ][ y ] = ΣiΣj varTempV[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑444)

varTempD01[ x ][ y ] = ΣiΣj varTempD0[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑445)

varTempD11[ x ][ y ] = ΣiΣj varTempD1[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑446)

varTemp[ x ][ y ] = varTempH1[ x ][ y ] + varTempV1[ x ][ y ] (8‑447)

1. The variables mainDirection, secondaryDirection and directionStrength with x, y = 0..CtbSizeY)
2. The variables mainDirection, secondaryDirection and directionStrength with x, y = 0..CtbSizeY − 1 are derived as follows:

* The variables hv1, hv0 and directionHV are derived as follows:
* If varTempV1[ x >> 2 ][ y >> 2 ] is greater than varTempH1[ x >> 2 ][ y >> 2 ], the following applies:

hv1 = varTempV1[ x >> 2 ][ y >> 2 ] (8‑448)

hv0 = varTempH1[ x >> 2 ][ y >> 2 ]  (8‑449)

directionHV = 1 (8‑450)

* Otherwise, the following applies:

hv1 = varTempH1[ x >> 2 ][ y >> 2 ] (8‑451)

hv0 = varTempV1[ x >> 2 ][ y >> 2 ]  (8‑452)

directionHV = 3 (8‑453)

* The variables d1, d0 and directionD are derived as follows:
* If varTempD0[ x >> 2 ][ y >> 2 ] is greater than varTempD1[ x >> 2 ][ y >> 2 ], the following applies:

d1 = varTempD0[ x >> 2 ][ y >> 2 ] (8‑454)

d0 = varTempD1[ x >> 2 ][ y >> 2 ]  (8‑455)

directionD = 0 (8‑456)

* Otherwise, the following applies:

d1 = varTempD1[ x >> 2 ][ y >> 2 ] (8‑457)

d0 = varTempD0[ x >> 2 ][ y >> 2 ]  (8‑458)

directionD = 2 (8‑459)

* The variables hvd1, hvd0, are derived as follows:

hvd1 = ( d1 \* hv0 > hv1 \* d0 )  ?  d1  :  hv1 (8‑460)

hvd0 = ( d1 \* hv0 > hv1 \* d0 )  ?  d0  :  hv0 (8‑461)

* The variables directionStrength, mainDirection and secondaryDirectionare derived as follows:

mainDirection = ( d1 \* hv0 > hv1 \* d0 )  ?  directionD  :  directionHV (8‑462)

secondaryDirection = ( d1 \* hv0 > hv1 \* d0 )  ?  direction  :  HVdirectionD (8‑463)

directionStrength = ( hvd1 > 2 \* hvd0 )  ?  1  :  ( ( hvd1 \* 2 > 9 \* hvd0 )  ?  2  :  0 ) (8‑464)

1. The variable avgVar is derived as follows:

varTab[ ] = { 0, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4 } (8‑465)

avgVar = varTab[ Clip3( 0, 15, ( varTemp[ x >> 2 ][ y >> 2 ] \* 32 ) >> ( 3 + BitDepthY ) ) ] (8‑466)

1. The classification filter index array filtIdx[ x ][ y ] and the transpose index array transposeIdx[ x ][ y ] with x = y = 0..CtbSizeY − 1 are derived as follows:

transposeTable[ ] = { 0, 1, 0, 2, 2, 3, 1, 3 }

transposeIdx[ x ][ y ] = transposeTable[ mainDirection \* 2 + ( secondaryDirection >> 1 ) ]

filtIdx[ x ][ y ] = avgVar

When directionStrength is not equal 0, filtIdx[ x ][ y ] is modified as follows:

filtIdx[ x ][ y ] += ( ( ( mainDirection & 0x1 ) << 1 ) + directionStrength ) \* 5 (8‑467)

#### Coding tree block filtering process for chroma samples

Inputs of this process are:

* a reconstructed chroma picture sample array recPicture prior to the adaptive loop filtering process,
* a filtered reconstructed chroma picture sample array alfPicture,
* a chroma location ( xCtbC, yCtbC ) specifying the top-left sample of the current chroma coding tree block relative to the top left sample of the current picture,
* an array of chroma filter coefficients cC[ j ] with j = 0..6.

Output of this process is the modified filtered reconstructed chroma picture sample array alfPicture.

The size of the current chroma coding tree block ctbSizeC is derived as follows:

ctbSizeC = CtbSizeY / SubWidthC (8‑468)

For the derivation of the filtered reconstructed chroma samples alfPicture[ x ][ y ], each reconstructed chroma sample inside the current chroma coding tree block recPicture[ x ][ y ] is filtered as follows with x, y = 0..ctbSizeC − 1:

* + The locations ( hx, vy ) for each of the corresponding chroma samples ( x, y ) inside the given array recPicture of chroma samples are derived as follows:

hx = Clip3( 0, pic\_width\_in\_luma\_samples / SubWidthC − 1, xCtbC + x ) (8‑469)

vy = Clip3( 0, pic\_height\_in\_luma\_samples / SubHeightC − 1, yCtbC + y ) (8‑470)

* + The variable sum is derived as follows:

sum = cC[ 0 ] \* ( recPicture[ hx, vy + 2 ] + recPicture[ hx, vy− 2 ] ) +   
 cC[ 1 ] \* ( recPicture[ hx + 1, vy + 1 ] + recPicture[ hx − 1, vy − 1 ] ) +   
 cC[ 2 ] \* ( recPicture[ hx, vy + 1 ] + recPicture[ hx, vy − 1 ] ) + (8‑471)  
 cC[ 3 ] \* ( recPicture[ hx − 1, vy + 1 ] + recPicture[ hx + 1, vy − 1 ] ) +   
 cC[ 4 ] \* ( recPicture[ hx + 2, vy ] + recPicture[ hx − 2, vy ] ) +   
 cC[ 5 ] \* ( recPicture[ hx + 1, vy ] + recPicture[ hx − 1, vy ] ) +   
 cC[ 6 ] \* recPictureC[ hx, vy ]

sum = ( sum + 256 ) >> 9 (8‑472)

* + The modified filtered reconstructed chroma picture sample alfPicture[ xCtbC + x ][ yCtbC + y ] is derived as follows:

alfPicture[ xCtbC + x ][ yCtbC + y ] = Clip3( 0, ( 1 << BitDepthC ) − 1, sum ) (8‑473)

# Parsing process

## General

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), se(v), uek(v) (see clause 9.2), tu(v) (see clause 9.3), tb(v) (see clause 9.4), or ae(v) (see clause 9.5).

## Parsing process for k-th order Exp-Golomb codes

### General

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), uek(v) or se(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v) or se(v) are Exp-Golomb-coded with order k equal to 0 and syntax elements coded as uek(v) are Exp-Golomb-coded with order k. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

leadingZeroBits = −1  
for( b = 0; !b; leadingZeroBits++ ) (9‑1)  
 b = read\_bits( 1 )

The variable codeNum is then assigned as follows:

codeNum = ( 2leadingZeroBits − 1 ) \* 2k + read\_bits( leadingZeroBits + k ) (9‑2)

where the value returned from read\_bits( leadingZeroBits ) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

Table 9‑1 illustrates the structure of the 0-th order Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed as specified above for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 9‑1. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as xi in Table 9‑1, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each xi is equal to either 0 or 1.

Table 9‑1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative)

|  |  |
| --- | --- |
| **Bit string form** | **Range of codeNum** |
| 1 | 0 |
| 0 1 x0 | 1..2 |
| 0 0 1 x1 x0 | 3..6 |
| 0 0 0 1 x2 x1 x0 | 7..14 |
| 0 0 0 0 1 x3 x2 x1 x0 | 15..30 |
| 0 0 0 0 0 1 x4 x3 x2 x1 x0 | 31..62 |
| ... | ... |

Table 9‑2 illustrates explicitly the assignment of bit strings to codeNum values.

Table 9‑2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative)

|  |  |
| --- | --- |
| **Bit string** | **codeNum** |
| 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 2 |
| 0 0 1 0 0 | 3 |
| 0 0 1 0 1 | 4 |
| 0 0 1 1 0 | 5 |
| 0 0 1 1 1 | 6 |
| 0 0 0 1 0 0 0 | 7 |
| 0 0 0 1 0 0 1 | 8 |
| 0 0 0 1 0 1 0 | 9 |
| ... | ... |

Depending on the descriptor, the value of a syntax element is derived as follows:

* If the syntax element is coded as ue(v), the value of the syntax element is equal to codeNum.
* Otherwise (the syntax element is coded as se(v)), the value of the syntax element is derived by invoking the mapping process for signed Exp-Golomb codes as specified in clause 9.2.2 with codeNum as input.

### Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in clause 9.2.

Output of this process is a value of a syntax element coded as se(v).

The syntax element is assigned to the codeNum by ordering the syntax element by its absolute value in increasing order and representing the positive value for a given absolute value with the lower codeNum. Table 9‑3 provides the assignment rule.

Table 9‑3 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v)

|  |  |
| --- | --- |
| **codeNum** | **syntax element value** |
| 0 | 0 |
| 1 | 1 |
| 2 | −1 |
| 3 | 2 |
| 4 | −2 |
| 5 | 3 |
| 6 | −3 |
| k | (−1)k + 1 Ceil( k ÷ 2 ) |

## Parsing process for truncated unary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tu(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tu(v) are truncated unary coded. The range of possible values for the syntax element is determined first. The range of this syntax element may be between 0 and maxVal, with maxVal being greater than or equal to 1. codeNum which is equal to the value of the syntax element is given by a process specified as follows:

codeNum = 0  
keepGoing = 1  
for(i = 0; i < maxVal && keepGoing; i++){  
 keepGoing = read\_bits( 1 ) (9‑3)  
 if( keepGoing )  
 codeNum ++  
}

## Parsing process for truncated binary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tb(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tb(v) are truncated binary coded. The range of possible values for the syntax element is determined first. The range of this syntax element may be between 0 and maxVal, with maxVal being greater than or equal to 1. synVal which is equal to the value of the syntax element is given by a process specified as follows:

thVal = 1 << 8  
th = 8  
while( thVal <= maxVal ) {  
 th++  
 thVal <<= 1  
}  
th− −  
val = 1 << th (9‑4)  
b = n − val  
synVal = read\_bits( 1 )  
if( synVal >= val - b ) {  
 synVal <<= 1  
 synVal += read\_bits( 1 )  
 synVal −= val − b  
}

## CABAC parsing process for slice segment data

### General

### Initialization process

### Binarization process

#### General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 9‑4 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the k-th order Exp-Golomb (EGk) binarization process and the fixed-length (FL) binarization process are given in clauses 9.5.3.3 through 9.5.3.5, respectively.

| Table 9‑4 – Syntax elements and associated binarizations | | | |
| --- | --- | --- | --- |
| **Syntax structure** | **Syntax element** | **Binarization** | |
| **Process** | **Input parameters** |
| coding\_quadtree( ) | qt\_split\_cu\_flag[ ][ ] | FL | cMax = 1 |
| multi\_type\_tree( ) | mtt\_split\_cu\_flag | FL | cMax = 1 |
| mtt\_split\_cu\_vertical\_flag | FL | cMax = 1 |
| mtt\_split\_cu\_binary\_flag | FL | cMax = 1 |
| coding\_tree\_unit( ) | alf\_ctb\_flag[ ][ ][ ] | FL | cMax = 1 |
| coding\_unit( ) | cu\_skip\_flag[ ][ ] | FL | cMax = 1 |
| pred\_mode\_flag | FL | cMax = 1 |
| intra\_luma\_mpm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_luma\_mpm\_idx[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| intra\_luma\_mpm\_remainder[ ][ ] | FL | cMax = 63 |
| intra\_chroma\_pred\_mode[ ][ ] | 9.5.3.6 | - |
| merge\_affine\_flag[ ][ ] | FL | cMax = 1 |
| merge\_flag[ ][ ] | FL | cMax = 1 |
| merge\_idx[ ][ ] | TR | cMax = MaxNumMergeCand − 1, cRiceParam = 0 |
| inter\_pred\_idc[ x0 ][ y0 ] | 9.5.3.7 | - |
| inter\_affine\_flag[ ][ ] | FL | cMax = 1 |
| cu\_affine\_type\_flag[ ][ ] | FL | cMax = 1 |
| ref\_idx\_l0[ ][ ] | TR | cMax = num\_ref\_idx\_l0\_active\_minus1, cRiceParam = 0 |
| mvp\_l0\_flag[ ][ ] | FL | cMax = 1 |
| ref\_idx\_l1[ ][ ] | TR | cMax = num\_ref\_idx\_l1\_active\_minus1, cRiceParam = 0 |
| mvp\_l1\_flag[ ][ ] | FL | cMax = 1 |
| amvr\_mode[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| cu\_cbf | FL | cMax = 1 |
| mvd\_coding( ) | abs\_mvd\_greater0\_flag[ ] | FL | cMax = 1 |
| abs\_mvd\_greater1\_flag[ ] | FL | cMax = 1 |
| abs\_mvd\_minus2[ ] | EG1 | - |
| mvd\_sign\_flag[ ] | FL | cMax = 1 |
| transform\_unit( ) | tu\_cbf\_luma[ ][ ][ ] | FL | cMax = 1 |
| tu\_cbf\_cb[ ][ ][ ] | FL | cMax = 1 |
| tu\_cbf\_cr[ ][ ][ ] | FL | cMax = 1 |
| cu\_mts\_flag[ ][ ] | FL | cMax = 1 |
| residual\_coding( ) | transform\_skip\_flag[ ][ ][ ] | FL | cMax = 1 |
| last\_sig\_coeff\_x\_prefix | TR | cMax = ( log2TrafoSize << 1 ) − 1, cRiceParam = 0 |
| last\_sig\_coeff\_y\_prefix | TR | cMax = ( log2TrafoSize << 1 ) − 1, cRiceParam = 0 |
| last\_sig\_coeff\_x\_suffix | FL | cMax = ( 1  <<  ( ( last\_sig\_coeff\_x\_prefix  >>  1 ) − 1 ) − 1 ) |
| last\_sig\_coeff\_y\_suffix | FL | cMax = ( 1  <<  ( ( last\_sig\_coeff\_y\_prefix  >>  1 ) − 1 ) − 1 ) |
| coded\_sub\_block\_flag[ ][ ] | FL | cMax = 1 |
| sig\_coeff\_flag[ ][ ] | FL | cMax = 1 |
| par\_level\_flag[ ] | FL | cMax = 1 |
| rem\_abs\_gt1\_flag[ ] | FL | cMax = 1 |
| rem\_abs\_gt2\_flag[ ] | FL | cMax = 1 |
| abs\_remainder[ ] | 9.5.3.8 | cIdx, x0, y0, xC, yC, log2TbWidth, log2TbHeight |
| coeff\_sign\_flag[ ] | FL | cMax = 1 |
| mts\_idx[ ][ ] | FL | cMax = 3 |

#### Rice parameter derivation process

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the Rice parameter cRiceParam.

Given the syntax elements sig\_coeff\_flag[ x ][ y ] and the array AbsLevel[ x ][ C ] for the transform block with component index cIdx and the top-left luma location ( x0, y0 ), the variable locSumAbs is derived as specified by the following pseudo code:

locSumAbs = 0  
if( xC < (1 << log2TbWidth) − 1 ) {  
 locSumAbs += AbsLevel[ xC + 1 ][ yC ] − sig\_coeff\_flag[ xC + 1 ][ yC ]  
 if( xC < (1 << log2TbWidth) − 2 )  
 locSumAbs += AbsLevel[ xC + 2 ][ yC ] − sig\_coeff\_flag[ xC + 2 ][ yC ]  
 if( yC < (1 << log2TbHeight) − 1 )  
 locSumAbs += AbsLevel[ xC + 1 ][ yC + 1 ] − sig\_coeff\_flag[ xC + 1 ][ yC + 1 ] (9‑5)  
}  
if( yC < (1 << log2TbHeight) − 1 ) {  
 locSumAbs += AbsLevel[ xC ][ yC + 1 ] − sig\_coeff\_flag[ xC ][ yC + 1 ]  
 if( yC < (1 << log2TbHeight) − 2 )  
 locSumAbs += AbsLevelPass1 [ xC ][ yC + 2 ] − sig\_coeff\_flag[ xC ][ yC + 2 ]  
}

The Rice parameter cRiceParam is derived as follows:

* If locSumAbs is less than 12, cRiceParam is set equal to 0;
* Otherwise, if locSumAbs is less than 25, cRiceParam is set equal to 1;
* Otherwise (locSumAbs is greater than or equal to 25), cRiceParam is set equal to 2.

#### Truncated Rice binarization process

Input to this process is a request for a truncated Rice (TR) binarization, cMax and cRiceParam.

Output of this process is the TR binarization associating each value symbolVal with a corresponding bin string.

A TR bin string is a concatenation of a prefix bin string and, when present, a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of symbolVal, prefixVal, is derived as follows:

prefixVal = symbolVal  >>  cRiceParam (9‑6)

* The prefix of the TR bin string is specified as follows:
* If prefixVal is less than cMax  >>  cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0. Table 9‑5 illustrates the bin strings of this unary binarization for prefixVal.
* Otherwise, the bin string is a bit string of length cMax  >>  cRiceParam with all bins being equal to 1.

Table 9‑5 – Bin string of the unary binarization (informative)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **prefixVal** | **Bin string** | | | | | |
| 0 | 0 |  |  |  |  |  |
| 1 | 1 | 0 |  |  |  |  |
| 2 | 1 | 1 | 0 |  |  |  |
| 3 | 1 | 1 | 1 | 0 |  |  |
| 4 | 1 | 1 | 1 | 1 | 0 |  |
| 5 | 1 | 1 | 1 | 1 | 1 | 0 |
| ... |  |  |  |  |  |  |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 |

When cMax is greater than symbolVal and cRiceParam is greater than 0, the suffix of the TR bin string is present and it is derived as follows:

* The suffix value suffixVal is derived as follows:

suffixVal = symbolVal − ( ( prefixVal )  <<  cRiceParam ) (9‑7)

* The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified in clause 9.5.3.5 for suffixVal with a cMax value equal to ( 1  <<  cRiceParam ) − 1.

NOTE – For the input parameter cRiceParam = 0, the TR binarization is exactly a truncated unary binarization and it is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

#### k-th order Exp-Golomb binarization process

Inputs to this process is a request for a k-th order Exp-Golomb (EGk) binarization.

Output of this process is the EGk binarization associating each value symbolVal with a corresponding bin string.

The bin string of the EGk binarization process for each value symbolVal is specified as follows, where each call of the function put( X ), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

absV = Abs( symbolVal )  
stopLoop = 0  
do  
 if( absV >= ( 1 << k ) ) {  
 put( 1 )  
 absV = absV − ( 1 << k )  
 k++  
 } else {  
 put( 0 ) (9‑8)  
 while( k− − )  
 put( ( absV >> k ) & 1 )  
 stopLoop = 1  
 }  
while( !stopLoop )

NOTE – The specification for the k-th order Exp-Golomb (EGk) code uses 1's and 0's in reverse meaning for the unary part of the Exp-Golomb code of k-th order as specified in clause 9.2.

#### Fixed-length binarization process

Inputs to this process are a request for a fixed-length (FL) binarization and cMax.

Output of this process is the FL binarization associating each value symbolVal with a corresponding bin string.

FL binarization is constructed by using the fixedLength‑bit unsigned integer bin string of the symbol value symbolVal, where fixedLength = Ceil( Log2( cMax + 1 ) ). The indexing of bins for the FL binarization is such that the binIdx = 0 relates to the most significant bit with increasing values of binIdx towards the least significant bit.

#### Binarization process for intra\_chroma\_pred\_mode

Input to this process is a request for a binarization for the syntax element intra\_chroma\_pred\_mode.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element intra\_chroma\_pred\_mode is specified in Table 9‑6 and Table 9‑7.

Table 9‑6 – Binarization for intra\_chroma\_pred\_mode when sps\_cclm\_enabled\_flag is equal to 0

|  |  |
| --- | --- |
| **Value of intra\_chroma\_pred\_mode** | **Bin string** |
| 4 | 0 |
| 0 | 100 |
| 1 | 101 |
| 2 | 110 |
| 3 | 111 |

Table 9‑7 – Binarization for intra\_chroma\_pred\_mode when sps\_cclm\_enabled\_flag is equal to 1

|  |  |
| --- | --- |
| **Value of intra\_chroma\_pred\_mode** | **Bin string** |
| 5 | 0 |
| 4 | 10 |
| 0 | 1100 |
| 1 | 1101 |
| 2 | 1110 |
| 3 | 1111 |

#### Binarization process for inter\_pred\_idc

Input to this process is a request for a binarization for the syntax element inter\_pred\_idc.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element inter\_pred\_idc is specified in Table 9‑8.

Table 9‑8 – Binarization for inter\_pred\_idc

|  |  |  |
| --- | --- | --- |
| **Value of inter\_pred\_idc** | **Name of inter\_pred\_idc** | **Bin string** |
| 0 | PRED\_L0 | 00 |
| 1 | PRED\_L1 | 01 |
| 2 | PRED\_BI | 1 |

#### Binarization process for abs\_remainder[ ]

Input to this process is a request for a binarization for the syntax element abs\_remainder[ n ], the colour component cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the picture), the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the binarization of the syntax element.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process as specified in clause 9.5.3.2 with the colour component index cIdx, the luma location ( x0, y0 ), the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as inputs.

The variable cMax is derived from cRiceParam as:

cMax = ( cRiceParam  = =  1 ? 6 : 7 )  <<  cRiceParam (9‑9)

The binarization of the syntax element abs\_remainder[ n ] is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of abs\_remainder[ n ], prefixVal, is derived as follows:

prefixVal = Min( cMax, abs\_remainder[ n ] ) (9‑10)

* The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with the variables cMax and cRiceParam as inputs.

When the prefix bin string is equal to the bit string of length 4 with all bits equal to 1, the suffix bin string is present and it is derived as follows:

* The suffix value of abs\_remainder[ n ], suffixVal, is derived as follows:

suffixVal = abs\_remainder[ n ] − cMax (9‑11)

* The suffix bin string is specified by invoking the k-th order EGk binarization process as specified in clause 9.5.3.4 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1.

### Decoding process flow

#### General

#### Derivation process for ctxTable, ctxIdx and bypassFlag

##### General

##### Derivation process of ctxInc for the syntax element tu\_cbf\_cr[ ][ ]

Inputs to this process is the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture.

Outputs of this process is the variable ctxInc.

The variable ctxInc is set equal to tu\_cbf\_cb[ x0 ][ y0 ].

##### Derivation process for the variables locNumSig, locSumAbsPass1

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Outputs of this process are the variables locNumSig and locSumAbsPass1.

Given the syntax elements sig\_coeff\_flag[ x ][ y ] and the array AbsLevelPass1[ x ][ C ] for the transform block with component index cIdx and the top-left luma location ( x0, y0 ), the variables locNumSig and locSumAbsPass1 are derived as specified by the following pseudo code:

locNumSig = 0  
locSumAbsPass1 = 0  
if( xC < (1 << log2TbWidth) − 1 ) {  
 locNumSig += sig\_coeff\_flag[ xC + 1 ][ yC ]  
 locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC ]  
 if( xC < (1 << log2TbWidth) − 2 ) {  
 locNumSig += sig\_coeff\_flag[ xC + 2 ][ yC ]   
 locSumAbsPass1 += AbsLevelPass1[ xC + 2 ][ yC ]  
 }  
 if( yC < (1 << log2TbHeight) − 1 ) {  
 locNumSig += sig\_coeff\_flag[ xC + 1 ][ yC + 1 ] (9‑12)  
 locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC + 1 ]  
 }  
}  
if( yC < (1 << log2TbHeight) − 1 ) {  
 locNumSig += sig\_coeff\_flag[ xC ][ yC + 1 ]   
 locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 1 ]  
 if( yC < (1 << log2TbHeight) − 2 ) {  
 locNumSig += sig\_coeff\_flag[ xC ][ yC + 2 ]   
 locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 2 ]  
 }  
}

##### Derivation process of ctxInc for the syntax element sig\_coeff\_flag

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.3 with colour component index cIdx, the luma location ( x0, y0), the current coefficient scan location (xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

* If cIdx is equal to 0, ctxInc is derived as follows:

ctxInc = 18 \* Max( 0, QState − 1) + Min( locSumAbsPass1, 5 ) + ( d < 2  ?  12  :  ( d < 5  ?  6  :  0 ) ) (9‑13)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 54 + 12 \* Max( 0, QState − 1) + Min( locSumAbsPass1, 5 ) + ( d < 2  ?  6  :  0 ) (9‑14)

##### Derivation process of ctxInc for the syntax elements par\_level\_flag, rem\_abs\_gt1\_flag, and rem\_abs\_level\_gt2\_flag

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variablea locNumSig and locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.3 with colour component index cIdx, the luma location ( x0, y0), the current coefficient scan location (xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable ctxOffset is set equal to Min( locSumAbsPass1 − locNumSig, 4 ).

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

* If xC is equal to LastSignificantCoeffX and yC is equal to LastSignificantCoeffY, ctxInc is derived as follows:

ctxInc = ( cIdx  = =  0  ?  0  :  21 ) (9‑15)

* Otherwise, if cIdx is equal to 0, ctxInc is derived as follows:

ctxInc = 1 + ctxOffset + ( d  = =  0  ?  15  :  ( d < 3  ?  10  :  ( d < 10  ?  5  :  0 ) ) ) (9‑16)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 22 + ctxOffset + ( d  = =  0  ?  5  :  0 ) (9‑17)