**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

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

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11 N18274**

**January 2019, Marrakech, MA**

**Source: JVET**

**Title: Working Draft 4 of Versatile Video Coding**

**Status: Approved**

|  |  |
| --- | --- |
| **Joint Video Experts Team (JVET)**  **of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11**  13th Meeting: Marrakech, MA, 9–18 Jan. 2019 | Document: JVET-M1001-v7 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **Versatile Video Coding (Draft 4)** | | |
| *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 tile group 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 tile group 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 tile groups (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 tile group 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.

Draft 3 of Versatile Video Coding.

Ed. Notes:

* Fixed bug [#92](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/92) missing MV clipping in interplolation process.
* Incorporated JVET-L0664: disabling 5x5 ALF for luma component.
* Incorporated JVET-L0082/JVET-L0083: reduction of bits for ALF coefficients.
* Incorporated JVET-L0147: subsampling of ALF classifiers.
* Incorporated SAO as found in HEVC.
* Incorporated deblocking filter as found in HEVC with the following modifications:
  + adapt processing to dual tree CTU partitioning,
  + replace RQT based transfrom block processing with implicit TU split for large blocks (JVET-K0307, JVET-K0237, JVET-K0369, JVET-K0232, JVET-K0315),
  + replace prediction blocks with coding subblocks (JVET-L0074),
  + only apply deblocking on transfrom block / coding subblock edges if CU is aligned on 8x8 grid in the respecitive direction.
* Incorporated JVET-L0410: tc table fix
* Incorporated JVET-L0414: luma adaptive deblocking filter.
* Fixed bug [#97](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/97) mismatch between agreed ALF text JVET-K0564 and draft by removing alf\_chroma\_ctb\_present\_flag.
* Incorporated JVET-L0285: 8-bit transform matrix.
* Incorporated JVET-L0118: MTS index fix.
* Incorporated JVET-L0059: remove dependency on number non-zero coeff. for MTS index signalling.
* Incorporated JVET-L0362: QP signalling.
* Incorporated JVET-L0428: delta QP and chroma QP offset for dual tree.
* Incorporated JVET-K0251: increase maximum QP value from 51 to 63 (including fix from JVET-L0553).
* Incorporated JVET-L0217: fix relation between QT/BT/TT syntax elements.
* Incorporated JVET-L0678: QT/BT/TT syntax overriding in tile group header.
* Incorporated JVET-L0081: BT/TT constraint.
* Incorporated JVET-L0191: cclm simplification using min and max values.
* Incorporated JVET-L0136: cclm with line buffer restriction.
* Incorporated JVET-L0340: multi-directional cclm.
* Incorporated JVET-L0053/JVET-L0272: modified chroma direct mode.
* Incorporated JVET-L0628: intra 4-tap interpolation filter.
* Incorporated JVET-L0165: intra 6 MPMs.
* Incorporated JVET-L0283: multi-line intra prediction.
* Incorporated JVET-L0279: unification of intra angular prediction.
* Incorporated PCM mode from HEVC and JVET-L0209: PCM mode with dual tree partition.
* Incorporated JVET-L0694: combination of affine mode and subblock temporal merging candidate modifications including:
  + JVET-L0142/JVET-L0366/JVET-L0632: affine merge candidate list (CE4 4.2.6.d as modifed in L0632).
  + JVET-L0369: moving subblock temporal merging candidate into affine merge candidate list (CE4 4.2.8).
  + JVET-L0045: line buffer reduction for affine inherited candidates, location 1 (CE4 4.1.11.a).
  + JVET-L0047: modification of affine control point motion vector storage (method 1).
  + JVET-L0271: Simplification of affine motion vector predictor candidate list construction (CE4 4.1.6.a).
* Incorporated JVET-L0265: change chroma subblock size to 4x4 instead of 2x2 for affine motion compensation.
* Incorporated JVET-L0198: fix subblock size to 8x8 for subblock TMVP (JVET-L0468/JVET-L0104).
* Incorporated JVET-L0198: use first spatial neighbouring MV for collocated subblock TMVP position
* Incorporated JVET-L0055: restrict subblock TMVP to CUs with width >= 8 and height >=8.
* Fixed bug [#120](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/120) clipping in vertical subblock TMVP location goes beyond the current CTU row.
* Incorporated JVET-L0274: reduce number of context coded bins (CE7 7.1.3.b).
* Incorporated missing context derivation for all CABAC-coded syntax elements.
* Incorporated JVET-L0361: context modeling of CU split modes.
* Incorporated JVET-L0194: use one context for the first bin of merge\_idx and bypass coding for the others.
* Incorporated JVET-L0266: History-based motion vector prediction.
* Incorporated JVET-L0158: Reset the HMVP FIFO list for each CTU row.
* Incorporated JVET-L0104: Prohibit 4x4 bi-prediction for inter CU.
* Incorporated JVET-L0054: Merge with MVD (MMVD) (CE4 4.5.4.b).
* Incorporated JVET-L0100: Combined inter merge / intra prediction (CIIP).
* Incorporated JVET-L0090: Pairwise average merging candidates.
* Incorporated JVET-L0646: Bi-prediction with CU weights.
* Incorporated JVET-L0256: Bidirectional optical flow (BDOF).
* Incorporated JVET-L0231: Horizontal MV wrap-around.
* Fixes and cleanups:
  + fixed minor typos,
  + resolved open issues from editors notes,
  + reordered SPS flags to be more aligned with VTM,
  + put all skip/merge related CU syntax in a merge\_data( ) syntax structure (editorial),
  + fixed order of merge syntax by moving merge\_idx and after combined merge/intra syntax,
  + conditioned combined merge/intra syntax on sps\_ciip\_enabled\_flag.
* Incorporated JVET-L0124: Triangular inter-picture prediction mode.
* Incorporated JVET-L0293: Current picture referencing (CPR).
* Fixed bug [#142](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/142) corrected order of derivation process for subblock-based temporal merging base motion data.
* Fixed bug [#137](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/137) wrong CCLM parameter derivation condition.
* Fixed issues related to triangular partitions.
* Incorporated JVET-L0064: Agreement in principle on the need of CRA and its signalling in NAL unit header
* Incorporated JVET-L0248: A restriction on the TemporalId values of the current picture and the active PPS
* Incorporated JVET-L0249: POC signalling and derivation
* Incorporated JVET-L0449: POC LSB in the tile group headers for all picture types
* Incorporated JVET-L0686: Tiling and tiling grouping; removed traditional slices
* Incorporated JVET-L0696: Interoperability point and constraint flags

Draft 4 of Versatile Video Coding.

Ed. Notes:

* Incorporated JVET-M0102: Intra subpartitions (ISP).
* Incorporated JVET-M0142: chroma format dependent CCLM downsampling filter.
* Incorporated JVET-M0064: table reduction in CCLM model parameter calculation.
* Incorporated JVET-M0092: intra reference sample filtering cleanup.
* Incorporated JVET-M0238: PDPC linear interpolation on the secondary boundary for adjacent angular modes is changed to nearest neighbour.
* Incorporated JVET-M0407: CPR search range.
* Fixed bug [#154](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/154) Availability check for CPR/IBC chroma CU reference block is missing.
* Aligned PDPC filtering for INTRA\_ANGULAR18 and INTRA\_ANGULAR50 to VTM.
* Fixed bug [#167](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/167) on PDPC size condition.
* Incorporated JVET-M0497: Fast DST-7/DCT-8.
* Incorporated JVET-M0303: Shape adaptive transform selection.
* Incorporated JVET-M0464/JVET-M0201: Combined transform skip(TS) and MTS syntax plus extended TS sizes.
* Incorporated JVET-M0297: Zero-out of last 16 samples for 32 samples DST-7/DCT-8.
* Incorporated JVET-M0140: Subblock transform for inter CUs.
* Incorporated JVET-M0251/M0257: Zero-out of last 32 samples for 64 samples DCT-2 fix.
* Incorporated JVET-M0273/M0240/M0116/M0338/M0204: only using left neighbor for SbTMVP fetching
* Incorporated JVET-M0246: AMVR for affine
* Incorporated JVET-M0145: affine sub-block MV clipping
* Incorporated JVET-M0166/M0228/M0477: remove MV comparison for constructed merge candidates
* Incorporated JVET-M0170: Parallel processing for merge mode
* Incorporated JVET-M0147: Decoder side motion vector refinement
* Incorporated JVET-M0361: fix of cu\_cbf for merge mode
* Incorporated JVET-M0487: using integer samples instead of bilinear interpolation for extended region of BDOF.
* Incorporated JVET-M0483: IBC signalled as a separate CU prediction mode.
* Incorporated JVET-M0063: Generalization of BDOF bit-depth.
* Incorporated triangular modifications including:
  + JVET-M0118/M0185/M0190/M207(test 1)/M0216(the first aspect)/M0234 (change corresponding to the result table 7 and 8)/M0317(section 2.2)/M0328: Do not signal the triangular prediction mode flag in cases where the combination is not allowed (MMVD, CIIP),
  + JVET-M0328: always use second weight group in triangular prediction.
* Incorporated JVET-M0883: signaling change of triangular merging candidate which does not need LUT.
* Incorporated JVET-M0193: pairwise average merging candidate reduction.
* Incorporated HMVP modifications including:
  + JVET-M0436 reduce HMVP number from 6 to 5,
  + JVET-M0300 HMVP initialization for parallel processing with tiles,
  + JVET-M0264 GBI weight is also stored in HMVP,
  + JVET-M0126 reduced HMVP candidate pruning.
* Incorporated JVET-M0255: MMVD mode without fractional sample offsets for screen content coding.
* Incorporated JVET-M0171/M0068: remove redundant MV scaling in MMVD.
* Incorporated JVET-M0444: symmetrical MVD coding for L0 to L1.
* Incorporated JVET-M0479: MV clipping to 18 bits.
* Incorporated JVET-M0512: TMVP storage reduction.
* Incorporated JVET-M0192: subblock chroma MV derivation for affine from two luma MVs.
* Incorporated JVET-M0111: weighted prediction (WP) and disable GBI signalling if WP is enabled.
* Incorporated JVET-M0281/M0117: modified AMVP pruning with rounding.
* Fixed bug [#175](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/175) incorrect derivation of CCLM parameter b.
* Incorporated JVET-M0128: Reference picture management
* Incorporated JVET-M0132: Adaptation parameter set (APS)
* Incorporated JVET-M0853: Adding the support of rectangular tile groups in addition to the existing raster-scan tile groups, and enabling extraction of MCTSs without changing VLC NAL units
* Incorporated JVET-M0160: Adding loop\_filter\_across\_tile\_group\_enabled\_flag to the PPS
* Incorporated JVET-M0101:
  + Replace the existing IRAP\_NUT with 3 new NAL unit types: IDR\_W\_RADL, IDR\_N\_LP, CRA\_NUT (from JVET-M0101).
  + Add external means flag HandleCraAsCvsStartFlag, with similar text as in HEVC. Text provided in a v3 of JVET-M0101.
  + Add a NUT value for step-wise temporal access STSA (from JVET-M0101).
  + Add a NUT value for AUD (from JVET-M0101).
  + Add sps\_max\_sub\_layers\_minus1 syntax element to SPS, and decoding process in 8.1.1, 8.1.2 and 8.1.3 of JVET-M0101.
  + Add text of sections 7.4.2.4 to 7.4.2.4.5 on NAL unit order and AU boundary detection from JVET-M0101, which is primarily editorial, but has some technical aspects.
  + Add profile\_tier\_level( ) syntax structure which includes sub-layer level idc (similar to HEVC but without sub-layer-specific profiles).
  + Add general\_non\_packed\_constraint\_flag with semantics as in JVET-M0101 (rename the flag to display\_suitability\_flag? – that's editorial).
  + Add the temporal scalability sub-bitstream extraction process in JVET-M0101.
  + Add RASL and RADL NUTs
* Incorporated JVET-M0451: Add new constraint flags corresponding to VVC WD 3 tools.
* Incorporated JVET-M0415: Change the sps\_ref\_wraparound\_offset to sps\_ref\_wraparound\_offset\_minus1 and changing the units to be MinCbSizeY as in option 1 (minor cleanup).
* Incorporated JVET-M0381: Reduce merge idx ctx coded bins (test 2.2.2a).
* Incorporated JVET-M0502: Add one context for pred\_mode\_flag (method 2).
* Incorporated JVET-M0453: Modified CABAC probability estimation (5.1.13\* + init from 5.1.2).
* Fixed bug [#147](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/147) on coefficient coding.
* Incorporated JVET-M0470: Limited EGk for abs\_rem/ dec\_abs\_level.
* Incorporated JVET-M0173: Move rem\_abs\_gt3\_flag into first coding pass.
* Incorporated JVET-M0119: Modified dequantization scaling for TS.
* Incorporated JVET-M0685: QP prediction fix for parallel encoding.
* Incorporated JVET-M0113/M0188: Bug fix for quantization group QP signalling.
* Incorporated JVET-M0421: Split-first signalling for partitioning.
* Incorporated JVET-M0446/M0888/M0905: Inferred QT split to avoid 32x128/128x32 partitions at picture boundaries.
* Incorporated JVET-M0427: Picture reconstruction with luma mapping and chroma scaling (LMCS).

**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 7

5 Conventions 9

5.1 General 9

5.2 Arithmetic operators 9

5.3 Logical operators 9

5.4 Relational operators 9

5.5 Bit-wise operators 9

5.6 Assignment operators 10

5.7 Range notation 10

5.8 Mathematical functions 10

5.9 Order of operation precedence 11

5.10 Variables, syntax elements and tables 12

5.11 Text description of logical operations 13

5.12 Processes 14

6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 15

6.1 Bitstream formats 15

6.2 Source, decoded and output picture formats 15

6.3 Partitioning of pictures, tile groups, tiles, and CTUs 17

6.3.1 Partitioning of pictures into tile groups and tiles 17

6.3.2 Block, quadtree and multi-type tree structures 18

6.3.3 Spatial or component-wise partitionings 19

6.4 Availability processes 19

6.4.1 Allowed quad split process 19

6.4.2 Allowed binary split process 20

6.4.3 Allowed ternary split process 21

6.4.4 Derivation process for neighbouring block availability 22

6.5 Scanning processes 22

6.5.1 CTB raster and tile scanning process 22

6.5.2 Up-right diagonal scan order array initialization process 24

7 Syntax and semantics 25

7.1 Method of specifying syntax in tabular form 25

7.2 Specification of syntax functions and descriptors 26

7.3 Syntax in tabular form 27

7.3.1 NAL unit syntax 27

7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 28

7.3.3 Profile, tier, and level syntax 33

7.3.4 Tile group header syntax 34

7.3.5 Reference picture list structure syntax 39

7.3.6 Tile group data syntax 39

7.4 Semantics 54

7.4.1 General 54

7.4.2 NAL unit semantics 54

7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 58

7.4.4 Profile, tier, and level semantics 69

7.4.5 Tile group header semantics 71

7.4.6 Reference picture list structure semantics 82

7.4.7 Tile group data semantics 83

8 Decoding process 98

8.1 General decoding process 98

8.2 NAL unit decoding process 99

8.3 Tile group decoding process 99

8.3.1 Decoding process for picture order count 99

8.3.2 Decoding process for reference picture lists construction 100

8.3.3 Decoding process for reference picture marking 102

8.3.4 Decoding process for symmetric motion vector difference reference indices 102

8.4 Decoding process for coding units coded in intra prediction mode 103

8.4.1 General decoding process for coding units coded in intra prediction mode 103

8.4.2 Derivation process for luma intra prediction mode 104

8.4.3 Derivation process for chroma intra prediction mode 109

8.4.4 Decoding process for intra blocks 110

8.5 Decoding process for coding units coded in inter prediction mode 129

8.5.1 General decoding process for coding units coded in inter prediction mode 129

8.5.2 Derivation process for motion vector components and reference indices 132

8.5.3 Decoder side motion vector refinement process 152

8.5.4 Derivation process for triangle motion vector components and reference indices 157

8.5.5 Derivation process for subblock motion vector components and reference indices 165

8.5.6 Derivation process for intra prediction mode in combined merge and intra prediction 186

8.5.7 Decoding process for inter blocks 188

8.5.8 Decoding process for triangle inter blocks 202

8.5.9 Decoding process for the residual signal of coding blocks coded in inter prediction mode 208

8.6 Decoding process for coding units coded in IBC prediction mode 209

8.6.1 General decoding process for coding units coded in IBC prediction mode 209

8.6.2 Derivation process for motion vector components for IBC blocks 211

8.6.3 Decoding process for ibc blocks 219

8.7 Scaling, transformation and array construction process 221

8.7.1 Derivation process for quantization parameters 221

8.7.2 Scaling and transformation process 222

8.7.3 Scaling process for transform coefficients 223

8.7.4 Transformation process for scaled transform coefficients 224

8.7.5 Picture reconstruction process 232

8.8 In-loop Filter Process 235

8.8.1 General 235

8.8.2 Deblocking filter process 236

8.8.3 Sample adaptive offset process 252

8.8.4 Adaptive loop filter process 254

9 Parsing process 258

9.1 General 258

9.2 Parsing process for k-th order Exp-Golomb codes 258

9.2.1 General 258

9.2.2 Mapping process for signed Exp-Golomb codes 259

9.3 Parsing process for truncated unary codes 260

9.4 Parsing process for truncated binary codes 260

9.5 CABAC parsing process for tile group data 261

9.5.1 General 261

9.5.2 Initialization process 262

9.5.3 Binarization process 268

9.5.4 Decoding process flow 276

10 Sub-bitstream extraction process 289

# 

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. **associated IRAP picture**: The previous *IRAP picture* in *decoding order* (when present).
  4. **associated non-VCL NAL unit**: A *non-VCL NAL unit* (when present) for a *VCL NAL unit* where the *VCL NAL unit* is the *associated VCL NAL unit* of the *non-VCL NAL unit*.
  5. **associated VCL NAL unit**: The preceding *VCL NAL unit* in *decoding order* for a *non-VCL NAL unit* with nal\_unit\_type equal to EOS\_NUT, EOB\_NUT, SUFFIX\_SEI\_NUT, or in the range of RSV\_NVCL26..RSV\_NVCL27 or UNSPEC30..UNSPEC31; or otherwise the next *VCL NAL unit* in *decoding order*.
  6. **bin**: One bit of a *bin string*.
  7. **binarization**: A set of *bin strings* for all possible values of a *syntax element*.
  8. **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.
  9. **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*.
  10. **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.
  11. **bi-predictive (B) tile group**: A *tile group* 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*.
  12. **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)*.
  13. **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.
  14. **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.
  15. **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.
  16. **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex TBD.
  17. **can**: A term used to refer to behaviour that is allowed, but not necessarily required*.*
  18. **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. **clean random access (CRA) access unit**: An *access unit* in which the *coded picture* is a *CRA picture*.
  2. **clean random access (CRA) picture**: An *IRAP picture* for which each *VCL NAL unit* has nal\_unit\_type equal to CRA\_NUT.

NOTE – A CRA picture does not refer to any pictures other than itself for inter prediction in its decoding process, and may be the first picture in the bitstream in decoding order, or may appear later in the bitstream. A CRA picture may have associated RADL or RASL pictures. When a CRA picture has NoRaslOutputFlag equal to 1, the associated RASL pictures are not output by the decoder, because they may not be decodable, as they may contain references to pictures that are not present in the bitstream.

* 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* with NoRaslOutputFlag equal to 1, followed by zero or more *access* *units* that are not *IRAP access units* with NoRaslOutputFlag equal to 1, including all subsequent *access units* up to but not including any subsequent *access unit* that is an *IRAP access unit* with NoRaslOutputFlag equal to 1.

NOTE – An IRAP access unit may be an IDR access unit or a CRA access unit. The value of NoRaslOutputFlag is equal to 1 for each IDR access unit and each CRA access unit that is the first access unit in the bitstream in decoding order, is the first access unit that follows an end of sequence NAL unit in decoding order, or has HandleCraAsCvsStartFlag equal to 1.

* 1. **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*.
  2. **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*.
  3. **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.
  4. **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.
  5. **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.
  6. **context variable**: A variable specified for the *adaptive binary arithmetic decoding* *process* of a *bin* by an equation containing recently decoded *bins*.
  7. **decoded picture**: A *decoded picture* is derived by decoding a *coded picture*.
  8. **decoder**: An embodiment of a *decoding process*.
  9. **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.
  10. **decoding process**: The process specified in this Specification that reads a *bitstream* and derives *decoded* *pictures* from it.
  11. **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*.
  12. **encoder**: An embodiment of an *encoding process*.
  13. **encoding process**: A process not specified in this Specification that produces a *bitstream* conforming to this Specification.
  14. **flag**: A variable or single-bit *syntax element* that can take one of the two possible values: 0 and 1.
  15. **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.*
  16. **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.
  17. **instantaneous decoding refresh (IDR) access unit**: An *access unit* in which the *coded picture* is an *IDR picture*.
  18. **instantaneous decoding refresh (IDR) picture**: An *IRAP* *picture* for which each *VCL NAL unit* has nal\_unit\_type equal to IDR\_W\_RADL or IDR\_N\_LP.

NOTE – An IDR picture does not refer to any pictures other than itself for inter prediction in its decoding process, and may be the first picture in the bitstream in decoding order, or may appear later in the bitstream. Each IDR picture is the first picture of a CVS in decoding order. When an IDR picture for which each VCL NAL unit has nal\_unit\_type equal to IDR\_W\_RADL, it may have associated RADL pictures. When an IDR picture for which each VCL NAL unit has nal\_unit\_type equal to IDR\_N\_LP, it does not have any associated leading pictures. An IDR picture does not have associated RASL pictures.

* 1. **inter coding**: Coding of a *coding block*, *tile group*, or *picture* that uses *inter prediction*.
  2. **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 block copy (IBC) prediction**: A *prediction* derived in a manner that is dependent on data elements (e.g., sample values or block vectors) of the same decoded *tile group* without referring to a *reference picture*.
  2. **intra coding**: Coding of a *coding block, tile group*, or *picture* that uses *intra prediction*.
  3. **intra prediction**: A *prediction* derived from only data elements (e.g., sample values) of the same decoded *tile group* without referring to a *reference picture*.
  4. **intra random access point (IRAP) access unit**: An *access unit* in which the *coded picture* is an *IRAP picture*.
  5. **intra random access point (IRAP) picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type in the range of IDR\_W\_RADL to RSV\_IRAP\_VCL13, inclusive.

NOTE – An IRAP picture does not refer to any pictures other than itself for inter prediction in its decoding process, and may be a CRA picture or an IDR picture. The first picture in the bitstream in decoding order must be an IRAP picture. Provided the necessary parameter sets are available when they need to be activated, the IRAP picture and all subsequent non-RASL pictures in decoding order can be correctly decoded without performing the decoding process of any pictures that precede the IRAP picture in decoding order.

* 1. **intra (I) tile group**: A *tile group* that is decoded using *intra prediction* only.
  2. **leading picture**: A *picture* that precedes the *associated* *IRAP picture* in *output order*.
  3. **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.
  4. **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 *tile group* using a *reference index* pointing into *reference picture list 0* (*list 1*).
  3. **long-term reference picture (LTRP)**: A *picture* that is marked as "used for long-term reference".
  4. **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-IDR picture**: A *coded picture* that is not an *IDR picture*.
  7. **non-IRAP picture**: A *coded picture* that is not an *IRAP picture*.
  8. **non-VCL NAL unit**: A *NAL unit* that is not a *VCL NAL unit*.
  9. **note**: A term that is used to prefix *informative* remarks (used exclusively in an *informative* context).
  10. **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*).
  11. **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*.
  12. **partitioning**: The division of a set into subsets such that each element of the set is in exactly one of the subsets.
  13. **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 *tile group 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) tile group**: A *tile group* 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. **random access decodable leading (RADL) access unit**: An *access unit* in which the *coded picture* is a *RADL picture*.
  13. **random access decodable leading (RADL) picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type equal to RADL\_NUT.

NOTE – All RADL pictures are leading pictures. RADL pictures are not used as reference pictures for the decoding process of trailing pictures of the same associated IRAP picture. When present, all RADL pictures precede, in decoding order, all trailing pictures of the same associated IRAP picture.

* 1. **random access skipped leading (RASL) access unit**: An *access unit* in which the *coded picture* is a *RASL picture.*
  2. **random access skipped leading (RASL) picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type equal to RASL\_NUT.

NOTE – All RASL pictures are leading pictures of an associated CRA picture. When the associated CRA picture has NoRaslOutputFlag equal to 1, the RASL picture is not output and may not be correctly decodable, as the RASL picture may contain references to pictures that are not present in the bitstream. RASL pictures are not used as reference pictures for the decoding process of non-RASL pictures. When present, all RASL pictures precede, in decoding order, all trailing pictures of the same associated CRA picture.

* 1. **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.
  2. **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.
  3. **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.*
  4. **reference index**: An index into a *reference picture list*.
  5. **reference picture**: A *picture* that is a *short-term reference picture* or a *long-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 tile group.*

NOTE – Two reference picture lists, reference picture list 0 and reference picture list 1, are generated for each tile group of a non-IDR picture. The set of unique pictures referred to by all entries in the two reference picture lists associated with a picture consists of all reference pictures that may be used for inter prediction of the associated picture or any picture following the associated picture in decoding order. For the decoding process of a P tile group, only reference picture list 0 is used for inter prediction. For the decoding process of a B tile group, both reference picture list 0 and reference picture list 1 are used for inter prediction. For decoding the tile group data of an I tile group, no reference picture list is used for for inter prediction.

* 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* *tile group*.
  2. **reference picture list 1**: The second *reference picture list* used for *inter prediction* of a *B tile group*.
  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 *tile group 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 (STRP)**: 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. **source**: A term used to describe the video material or some of its attributes before encoding.
  4. **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. **step-wise temporal sub-layer access (STSA) access unit**: An *access unit* in which the *coded picture* is an *STSA picture*.
  2. **step-wise temporal sub-layer access (STSA) picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type equal to STSA\_NUT.

NOTE – An STSA picture does not use pictures with the same TemporalId as the STSA picture for inter prediction reference. Pictures following an STSA picture in decoding order with the same TemporalId as the STSA picture do not use pictures prior to the STSA picture in decoding order with the same TemporalId as the STSA picture for inter prediction reference. An STSA picture enables up-switching, at the STSA picture, to the sub-layer containing the STSA picture, from the immediately lower sub-layer. STSA pictures must have TemporalId greater than 0.

* 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. **sub-bitstream extraction process**: A specified process by which *NAL units* in a *bitstream* that do not belong to a target set, determined by a target highest TemporalId, are removed from the *bitstream*, with the output sub-bitstream consisting of the NAL units in the *bitstream* that belong to the target set.
  3. **sub-layer**: A temporal scalable layer of a temporal scalable *bitstream*, consisting of *VCL NAL units* with a particular value of the TemporalId variable and the associated *non-VCL NAL units*.
  4. **syntax element**: An element of data represented in the *bitstream*.
  5. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
  6. **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*.
  7. **tier**: A specified category of *level* constraints imposed on values of the *syntax elements* in the *bitstream*, where the *level* constraints are nested within a *tier* and a *decoder* conforming to a certain *tier* and *level* would be capable of decoding all *bitstreams* that conform to the same *tier* or the lower *tier* of that *level* or any *level* below it.
  8. **tile**: A rectangular region of *CTUs* within a particular *tile column* and a particular *tile row* in a *picture*.
  9. **tile column**: A rectangular region of *CTUs* having a height equal to the height of the *picture* and a width specified by *syntax elements* in the *picture parameter set*.
  10. **tile group**: An integer number of *tiles* of a *picture* that are exclusivelycontained in a single *NAL unit*.
  11. **tile group header**: A part of a coded *tile group* containing the data elements pertaining to the first or all *tiles* represented in the *tile group*.
  12. **tile row**: A rectangular region of *CTUs* having a height specified by *syntax elements* in the *picture parameter set* and a width equal to the width of the *picture*.
  13. **tile scan**: A specific sequential ordering of *CTUs* *partitioning* a *picture* in which the *CTUs* are ordered consecutively in *CTU* *raster scan* in a *tile* whereas *tiles* in a *picture* are ordered consecutively in a *raster scan* of the *tiles* of the *picture*.
  14. **trailing picture**: A non-IRAP *picture* that follows the *associated IRAP picture* in *output order* and that is not an STSA picture.

NOTE – Trailing pictures associated with an IRAP picture also follow the IRAP picture in decoding order. Pictures that follow the associated IRAP picture in output order and precede the associated IRAP picture in decoding order are not allowed.

* 1. **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.
  2. **transform block**: A rectangular MxN *block* of samples resulting from a *transform* in the *decoding process*.
  3. **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*.
  4. **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.
  5. **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.
  6. **tree**: A tree is a finite set of nodes with a unique root node.
  7. **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.
  8. **video coding layer (VCL) NAL unit**: A collective term for *coded tile group 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

APS Adaptation Parameter Set

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

IBC Intra Block Copy

IRAP Intra Random Access Point

LPS Least Probable Symbol

LSB Least Significant Bit

LTRP Long-Term Reference Picture

LMCS Luma Mapping with Chroma Scaling

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

SAO Sample Adaptive Offset

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

STRP Short-Term Reference Picture

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)

ClipH( o, W, x ) = (5‑9)

Cos( x ) the trigonometric cosine function operating on an argument x in units of radians. (5‑10)

Floor( x ) the largest integer less than or equal to x. (5‑11)

GetCurrMsb( a, b, c, d ) = (5‑12)

Ln( x ) the natural logarithm of x (the base-e logarithm, where e is the natural logarithm base constant 2.718 281 828...). (5‑13)

Log2( x ) the base-2 logarithm of x. (5‑14)

Log10( x ) the base-10 logarithm of x. (5‑15)

Min( x, y ) = (5‑16)

Max( x, y ) = (5‑17)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (5‑18)

Sign( x ) = (5‑19)

Sin( x ) the trigonometric sine function operating on an argument x in units of radians (5‑20)

Sqrt( x ) = (5‑21)

Swap( x, y ) = ( y, x ) (5‑22)

Tan( x ) the trigonometric tangent function operating on an argument x in units of radians (5‑23)

## 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, tile groups, tiles, and CTUs

### Partitioning of pictures into tile groups and tiles

This subclause specifies how a picture is partitioned into tile groups and tiles.

Pictures are divided into tile groups and tiles. A tile is a sequence of CTUs that cover a rectangular region of a picture. A tile group contains a number of tiles of a picture.

Two modes of tile groups are supported, namely the raster-scan tile group mode and the rectangular tile group mode. In the raster-scan tile group mode, a tile group contains a sequence of tiles in tile raster scan of a picture. In the rectangular tile group mode, a tile group contains a number of tiles of a picture that collectively form a rectangular region of the picture. The tiles within a rectangular tile group are in the order of tile raster scan of the tile group.

Figure 6‑4 shows an example of raster-scan tile group partitioning of a picture, where the picture is divided into 12 tiles and 3 raster-scan tile groups.

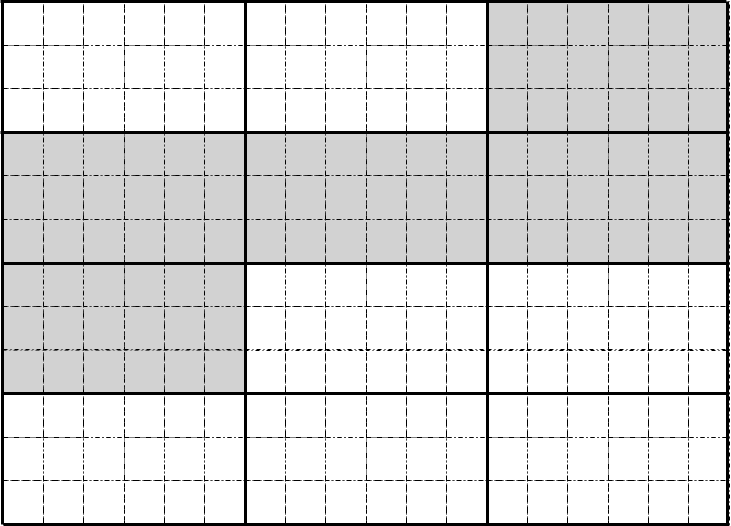


Figure 6‑4 – A picture with 18 by 12 luma CTUs that is partitioned into 12 tiles and 3 tile groups (informative)

Figure 6‑5 shows an example of rectangular tile group partitioning of a picture, where the picture is divided into 24 tiles (6 tile columns and 4 tile rows) and 9 rectangular tile groups.

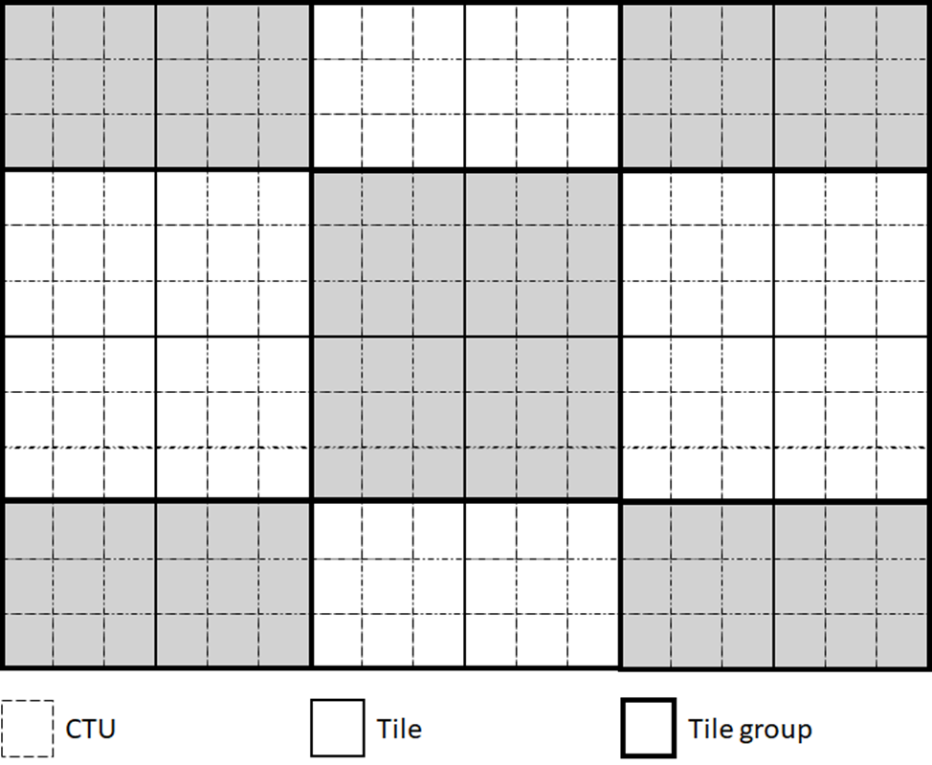


Figure 6‑5 – A picture with 18 by 12 luma CTUs that is partitioned into 24 tiles and 9 tile groups (informative)

When a picture is coded using three separate colour planes (separate\_colour\_plane\_flag is equal to 1), a tile group 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 tile groups having the same colour\_plane\_id value. Coded tile groups 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 tile group NAL units with that value of colour\_plane\_id shall be in the order of increasing CTU address in tile scan order for the first CTU of each coded tile group NAL unit.

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

### 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 tile columns

– The division of each picture into tile rows

– The division of each tile column into tiles

– The division of each tile row into tiles

– The division of each tile into CTUs

– The division of each picture into tile groups

– The division of each tile group into tiles

– The division of each tile group 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 quad split process

Inputs to this process are:

* a coding block size cbSize,
* a multi-type tree depth mttDepth.

Output of this process is the variable allowSplitQt.

The variable allowSplitQt is derived as follows:

* If one or more of the following conditions are true, allowSplitQt is set equal to FALSE:
* cbSize is less than or equal to MinQtSizeY

– mttDepth is not equal to 0

* Otherwise, allowSplitQt is set equal to TRUE.

### Allowed binary split process

Inputs to this process are:

* 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 maximum multi-type tree depth with offset maxMttDepth,
* a maximum binary tree size maxBtSize,
* a partition index partIdx.

Output of this process is the variable allowBtSplit.

Table 6‑2 – Specification of parallelTtSplit 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 maxBtSize
* cbHeight is greater than maxBtSize
* mttDepth is greater than or equal to maxMttDepth
* 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\_VER
* cbHeight is greater than MaxTbSizeY
* x0 + cbWidth is greater than pic\_width\_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
* cbWidth is greater than MaxTbSizeY
* 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 less 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 if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_VER
* cbWidth is less than or equal to MaxTbSizeY
* cbHeight is greater than MaxTbSizeY
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_HOR
* cbWidth is greater than MaxTbSizeY
* cbHeight is less than or equal to MaxTbSizeY

– Otherwise, allowBtSplit is set equal to TRUE.

### Allowed ternary split process

Inputs to this process are:

* 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
* a maximum multi-type tree depth with offset maxMttDepth,
* a maximum ternary tree size maxTtSize.

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 Min( MaxTbSizeY, maxTtSize )
* cbHeight is greater than Min( MaxTbSizeY, maxTtSize )
* mttDepth is greater than or equal to maxMttDepth
* 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.

### Derivation process for neighbouring block availability

Inputs to this process are:

* the luma location ( xCurr, yCurr ) of the top-left sample of the current block relative to the top-left luma sample of the current picture,
* the luma location ( xNbY, yNbY ) covered by a neighbouring block relative to the top-left luma sample of the current picture.

Output of this process is the availability of the neighbouring block covering the location ( xNbY, yNbY ), denoted as availableN.

The neighbouring block availability availableN is derived as follows:

– If the neighbouring block is contained in a different tile than the current block, availableN is set equal to FALSE. [Ed. (YK): The case outside of a tile group boundary is covered by this sentence as each tile group always contains an integer number of complete tiles.]

– Otherwise, [Ed. (CJ): the derivation process of availableN is to be specified.]

## Scanning processes

### CTB raster and tile scanning process

The list ColWidth[ i ] for i ranging from 0 to num\_tile\_columns\_minus1, inclusive, specifying the width of the i-th tile column in units of CTBs, is derived as follows:

if( uniform\_tile\_spacing\_flag )  
 for( i = 0; i <= num\_tile\_columns\_minus1; i++ )  
 ColWidth[ i ] = ( ( i + 1 ) \* PicWidthInCtbsY ) / ( num\_tile\_columns\_minus1 + 1 ) −   
 ( i \* PicWidthInCtbsY ) / ( num\_tile\_columns\_minus1 + 1 )  
else {  
 ColWidth[ num\_tile\_columns\_minus1 ] = PicWidthInCtbsY (6‑1)  
 for( i = 0; i < num\_tile\_columns\_minus1; i++ ) {  
 ColWidth[ i ] = tile\_column\_width\_minus1[ i ] + 1  
 ColWidth[ num\_tile\_columns\_minus1 ] −= ColWidth[ i ]  
 }  
}

The list RowHeight[ j ] for j ranging from 0 to num\_tile\_rows\_minus1, inclusive, specifying the height of the j-th tile row in units of CTBs, is derived as follows:

if( uniform\_tile\_spacing\_flag )  
 for( j = 0; j <= num\_tile\_rows\_minus1; j++ )  
 RowHeight[ j ] = ( ( j + 1 ) \* PicHeightInCtbsY ) / ( num\_tile\_rows\_minus1 + 1 ) −   
 ( j \* PicHeightInCtbsY ) / ( num\_tile\_rows\_minus1 + 1 )  
else {  
 RowHeight[ num\_tile\_rows\_minus1 ] = PicHeightInCtbsY (6‑2)  
 for( j = 0; j < num\_tile\_rows\_minus1; j++ ) {  
 RowHeight[ j ] = tile\_row\_height\_minus1[ j ] + 1  
 RowHeight[ num\_tile\_rows\_minus1 ] −= RowHeight[ j ]  
 }  
}

The list ColBd[ i ] for i ranging from 0 to num\_tile\_columns\_minus1 + 1, inclusive, specifying the location of the i-th tile column boundary in units of CTBs, is derived as follows:

for( ColBd[ 0 ] = 0, i = 0; i <= num\_tile\_columns\_minus1; i++ )  
 ColBd[ i + 1 ] = ColBd[ i ] + ColWidth[ i ] (6‑3)

The list RowBd[ j ] for j ranging from 0 to num\_tile\_rows\_minus1 + 1, inclusive, specifying the location of the j-th tile row boundary in units of CTBs, is derived as follows:

for( RowBd[ 0 ] = 0, j = 0; j <= num\_tile\_rows\_minus1; j++ )  
 RowBd[ j + 1 ] = RowBd[ j ] + RowHeight[ j ] (6‑4)

The list CtbAddrRsToTs[ ctbAddrRs ] for ctbAddrRs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in CTB raster scan of a picture to a CTB address in tile scan, is derived as follows:

for( ctbAddrRs = 0; ctbAddrRs < PicSizeInCtbsY; ctbAddrRs++ ) {  
 tbX = ctbAddrRs % PicWidthInCtbsY  
 tbY = ctbAddrRs / PicWidthInCtbsY  
 for( i = 0; i <= num\_tile\_columns\_minus1; i++ )  
 if( tbX >= ColBd[ i ] )  
 tileX = i  
 for( j = 0; j <= num\_tile\_rows\_minus1; j++ ) (6‑5)  
 if( tbY >= RowBd[ j ] )  
 tileY = j  
 CtbAddrRsToTs[ ctbAddrRs ] = 0  
 for( i = 0; i < tileX; i++ )  
 CtbAddrRsToTs[ ctbAddrRs ] += RowHeight[ tileY ] \* ColWidth[ i ]  
 for( j = 0; j < tileY; j++ )  
 CtbAddrRsToTs[ ctbAddrRs ] += PicWidthInCtbsY \* RowHeight[ j ]  
 CtbAddrRsToTs[ ctbAddrRs ] += ( tbY − RowBd[ tileY ] ) \* ColWidth[ tileX ] + tbX − ColBd[ tileX ]  
}

The list CtbAddrTsToRs[ ctbAddrTs ] for ctbAddrTs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in tile scan to a CTB address in CTB raster scan of a picture, is derived as follows:

for( ctbAddrRs = 0; ctbAddrRs < PicSizeInCtbsY; ctbAddrRs++ ) (6‑6)  
 CtbAddrTsToRs[ CtbAddrRsToTs[ ctbAddrRs ] ] = ctbAddrRs

The list TileId[ ctbAddrTs ] for ctbAddrTs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in tile scan to a tile ID, is derived as follows:

for( j = 0, tileIdx = 0; j <= num\_tile\_rows\_minus1; j++ )  
 for( i = 0; i <= num\_tile\_columns\_minus1; i++, tileIdx++ )  
 for( y = RowBd[ j ]; y < RowBd[ j + 1 ]; y++ ) (6‑7)  
 for( x = ColBd[ i ]; x < ColBd[ i + 1 ]; x++ )  
 TileId[ CtbAddrRsToTs[ y \* PicWidthInCtbsY+ x ] ] = tileIdx

The list NumCtusInTile[ tileIdx ] for tileIdx ranging from 0 to NumTilesInPic − 1, inclusive, specifying the conversion from a tile index to the number of CTUs in the tile, is derived as follows:

for( j = 0, tileIdx = 0; j <= num\_tile\_rows\_minus1; j++ )  
 for( i = 0; i <= num\_tile\_columns\_minus1; i++, tileIdx++ ) (6‑8)  
 NumCtusInTile[ tileIdx ] = ColWidth[ i ] \* RowHeight[ j ]

The list FirstCtbAddrTs[ tileIdx ] for tileIdx ranging from 0 to NumTilesInPic − 1, inclusive, specifying the conversion from a tile ID to the CTB address in tile scan of the first CTB in the tile are derived as follows:

for( ctbAddrTs = 0, tileIdx = 0, tileStartFlag = 1; ctbAddrTs < PicSizeInCtbsY; ctbAddrTs++ ) {  
 if( tileStartFlag ) {  
 FirstCtbAddrTs[ tileIdx ] = ctbAddrTs (6‑9)  
 tileStartFlag = 0  
 }  
 tileEndFlag = ctbAddrTs = = PicSizeInCtbsY − 1 | | TileId[ ctbAddrTs + 1 ] != TileId[ ctbAddrTs ]  
 if( tileEndFlag ) {  
 tileIdx++  
 tileStartFlag = 1  
 }  
}

The values of ColumnWidthInLumaSamples[ i ], specifying the width of the i-th tile column in units of luma samples, are set equal to ColWidth[ i ]  <<  CtbLog2SizeY for i ranging from 0 to num\_tile\_columns\_minus1, inclusive.

The values of RowHeightInLumaSamples[ j ], specifying the height of the j-th tile row in units of luma samples, are set equal to RowHeight[ j ]  <<  CtbLog2SizeY for j ranging from 0 to num\_tile\_rows\_minus1, inclusive.

### 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 ) { (6‑10)  
 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\_max\_sub\_layers\_minus1** | u(3) |
| **sps\_reserved\_zero\_5bits** | u(5) |
| profile\_tier\_level( sps\_max\_sub\_layers\_minus1 ) |  |
| **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) |
| **log2\_max\_pic\_order\_cnt\_lsb\_minus4** | ue(v) |
| **sps\_max\_dec\_pic\_buffering\_minus1** | ue(v) |
| **long\_term\_ref\_pics\_flag** | u(1) |
| **rpl1\_same\_as\_rpl0\_flag** | u(1) |
| for( i = 0; i < !rpl1\_same\_as\_rpl0\_flag ? 2 : 1; i++ ) { |  |
| **num\_ref\_pic\_lists\_in\_sps**[ i ] | ue(v) |
| for( j = 0; j < num\_ref\_pic\_lists\_in\_sps[ i ]; j++) |  |
| ref\_pic\_list\_struct( i, j ) |  |
| } |  |
| **qtbtt\_dual\_tree\_intra\_flag** | u(1) |
| **log2\_ctu\_size\_minus2** | ue(v) |
| **log2\_min\_luma\_coding\_block\_size\_minus2** | ue(v) |
| **partition\_constraints\_override\_enabled\_flag** | u(1) |
| **sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_luma** | ue(v) |
| **sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_tile\_group** | ue(v) |
| **sps\_max\_mtt\_hierarchy\_depth\_inter\_tile\_group** | ue(v) |
| **sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_luma** | ue(v) |
| if( sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_luma != 0 ) { |  |
| **sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_luma** | ue(v) |
| **sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_luma** | ue(v) |
| } |  |
| if( sps\_max\_mtt\_hierarchy\_depth\_inter\_tile\_groups != 0 ) { |  |
| **sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_tile\_group** | ue(v) |
| **sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_tile\_group** | ue(v) |
| } |  |
| if( qtbtt\_dual\_tree\_intra\_flag ) { |  |
| **sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_chroma** | ue(v) |
| **sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_chroma** | ue(v) |
| if ( sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_chroma != 0 ) { |  |
| **sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_chroma** | ue(v) |
| **sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_chroma** | ue(v) |
| } |  |
| } |  |
| **sps\_sao\_enabled\_flag** | u(1) |
| **sps\_alf\_enabled\_flag** | u(1) |
| **sps\_pcm\_enabled\_flag** | u(1) |
| if( sps\_pcm\_enabled\_flag ) { |  |
| **pcm\_sample\_bit\_depth\_luma\_minus1** | u(4) |
| **pcm\_sample\_bit\_depth\_chroma\_minus1** | u(4) |
| **log2\_min\_pcm\_luma\_coding\_block\_size\_minus3** | ue(v) |
| **log2\_diff\_max\_min\_pcm\_luma\_coding\_block\_size** | ue(v) |
| **pcm\_loop\_filter\_disabled\_flag** | u(1) |
| } |  |
| **sps\_ref\_wraparound\_enabled\_flag** | u(1) |
| if( sps\_ref\_wraparound\_enabled\_flag ) |  |
| **sps\_ref\_wraparound\_offset\_minus1** | ue(v) |
| **sps\_temporal\_mvp\_enabled\_flag** | u(1) |
| if( sps\_temporal\_mvp\_enabled\_flag ) |  |
| **sps\_sbtmvp\_enabled\_flag** | u(1) |
| **sps\_amvr\_enabled\_flag** | u(1) |
| **sps\_bdof\_enabled\_flag** | u(1) |
| **sps\_affine\_amvr\_enabled\_flag** | u(1) |
| **sps\_dmvr\_enabled\_flag** | u(1) |
| **sps\_cclm\_enabled\_flag** | u(1) |
| if( sps\_cclm\_enabled\_flag && chroma\_format\_idc = = 1 ) |  |
| **sps\_cclm\_colocated\_chroma\_flag** | u(1) |
| **sps\_mts\_enabled\_flag** | u(1) |
| if( sps\_mts\_enabled\_flag ) { |  |
| **sps\_explicit\_mts\_intra\_enabled\_flag** | u(1) |
| **sps\_explicit\_mts\_inter\_enabled\_flag** | u(1) |
| } |  |
| **sps\_sbt\_enabled\_flag** | u(1) |
| if( sps\_sbt\_enabled\_flag ) |  |
| **sps\_sbt\_max\_size\_64\_flag** | u(1) |
| **sps\_affine\_enabled\_flag** | u(1) |
| if( sps\_affine\_enabled\_flag ) |  |
| **sps\_affine\_type\_flag** | u(1) |
| **sps\_gbi\_enabled\_flag** | u(1) |
| **sps\_ibc\_enabled\_flag** | u(1) |
| **sps\_ciip\_enabled\_flag** | u(1) |
| **sps\_fpel\_mmvd\_enabled\_flag** | u(1) |
| **sps\_triangle\_enabled\_flag** | u(1) |
| **sps\_lmcs\_enabled\_flag** | u(1) |
| **sps\_ladf\_enabled\_flag** | u(1) |
| if ( sps\_ladf\_enabled\_flag ) { |  |
| **sps\_num\_ladf\_intervals\_minus2** | u(2) |
| **sps\_ladf\_lowest\_interval\_qp\_offset** | se(v) |
| for( i = 0; i < sps\_num\_ladf\_intervals\_minus2 + 1; i++ ) { |  |
| **sps\_ladf\_qp\_offset**[ i ] | se(v) |
| **sps\_ladf\_delta\_threshold\_minus1**[ i ] | ue(v) |
| } |  |
| } |  |
| **sps\_extension\_flag** | u(1) |
| if( sps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **sps\_extension\_data\_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) |
| **single\_tile\_in\_pic\_flag** | u(1) |
| if( !single\_tile\_in\_pic\_flag ) { |  |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| **uniform\_tile\_spacing\_flag** | u(1) |
| if( !uniform\_tile\_spacing\_flag ) { |  |
| for( i = 0; i < num\_tile\_columns\_minus1; i++ ) |  |
| **tile\_column\_width\_minus1**[ i ] | ue(v) |
| for( i = 0; i < num\_tile\_rows\_minus1; i++ ) |  |
| **tile\_row\_height\_minus1**[ i ] | ue(v) |
| } |  |
| **single\_tile\_per\_tile\_group\_flag** | u(1) |
| if( !single\_tile\_per\_tile\_group\_flag ) |  |
| **rect\_tile\_group\_flag** | u(1) |
| if( rect\_tile\_group\_flag && !single\_tile\_per\_tile\_group\_flag ) { |  |
| **num\_tile\_groups\_in\_pic\_minus1** | ue(v) |
| for( i = 0; i <= num\_tile\_groups\_in\_pic\_minus1; i++ ) { |  |
| if( i > 0 ) |  |
| **top\_left\_tile\_idx**[ i ] | u(v) |
| **bottom\_right\_tile\_idx**[ i ] | u(v) |
| } |  |
| } |  |
| **loop\_filter\_across\_tiles\_enabled\_flag** | u(1) |
| if( loop\_filter\_across\_tiles\_enabled\_flag ) |  |
| **loop\_filter\_across\_tile\_groups\_enabled\_flag** | u(1) |
| } |  |
| if( rect\_tile\_group\_flag ) { |  |
| **signalled\_tile\_group\_id\_flag** | u(1) |
| if( signalled\_tile\_group\_id\_flag ) { |  |
| **signalled\_tile\_group\_id\_length\_minus1** | ue(v) |
| for( i = 0; i <= num\_tile\_groups\_in\_pic\_minus1; i++ ) |  |
| **tile\_group\_id**[ i ] | u(v) |
| } |  |
| } |  |
| **cabac\_init\_present\_flag** | u(1) |
| for( i = 0; i < 2; i++ ) |  |
| **num\_ref\_idx\_default\_active\_minus1**[ i ] | ue(v) |
| **rpl1\_idx\_present\_flag** | u(1) |
| **init\_qp\_minus26** | se(v) |
| **transform\_skip\_enabled\_flag** | u(1) |
| if( transform\_skip\_enabled\_flag ) |  |
| **log2\_transform\_skip\_max\_size\_minus2** | ue(v) |
| **cu\_qp\_delta\_enabled\_flag** | u(1) |
| if( cu\_qp\_delta\_enabled\_flag ) |  |
| **cu\_qp\_delta\_subdiv** | ue(v) |
| **pps\_cb\_qp\_offset** | se(v) |
| **pps\_cr\_qp\_offset** | se(v) |
| **pps\_tile\_group\_chroma\_qp\_offsets\_present\_flag** | u(1) |
| **weighted\_pred\_flag** | u(1) |
| **weighted\_bipred\_flag** | u(1) |
| **deblocking\_filter\_control\_present\_flag** | u(1) |
| if( deblocking\_filter\_control\_present\_flag ) { |  |
| **deblocking\_filter\_override\_enabled\_flag** | u(1) |
| **pps\_deblocking\_filter\_disabled\_flag** | u(1) |
| if( !pps\_deblocking\_filter\_disabled\_flag ) { |  |
| **pps\_beta\_offset\_div2** | se(v) |
| **pps\_tc\_offset\_div2** | se(v) |
| } |  |
| } |  |
| **pps\_extension\_flag** | u(1) |
| if( pps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **pps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Adaptation parameter set syntax

|  |  |
| --- | --- |
| adaptation\_parameter\_set\_rbsp( ) { | Descriptor |
| **adaptation\_parameter\_set\_id** | u(5) |
| alf\_data( ) |  |
| **aps\_extension\_flag** | u(1) |
| if( aps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **aps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Access unit delimiter RBSP syntax

|  |  |
| --- | --- |
| access\_unit\_delimiter\_rbsp( ) { | Descriptor |
| **pic\_type** | u(3) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### End of sequence RBSP syntax

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

#### End of bitstream RBSP syntax

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

#### Tile group layer RBSP syntax

|  |  |
| --- | --- |
| tile\_group\_layer\_rbsp( ) { | Descriptor |
| tile\_group\_header( ) |  |
| tile\_group\_data( ) |  |
| rbsp\_tile\_group\_trailing\_bits( ) |  |
| } |  |

#### RBSP tile group trailing bits syntax

|  |  |
| --- | --- |
| rbsp\_tile\_group\_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) |
| } |  |

### Profile, tier, and level syntax

#### General profile, tier, and level syntax

|  |  |
| --- | --- |
| profile\_tier\_level( maxNumSubLayersMinus1 ) { | **Descriptor** |
| **general\_profile\_idc** | u(7) |
| **general\_tier\_flag** | u(1) |
| general\_constraint\_info( ) |  |
| **general\_level\_idc** | u(8) |
| for( i = 0; i < maxNumSubLayersMinus1; i++ ) |  |
| **sub\_layer\_level\_present\_flag**[ i ] | u(1) |
| while( !byte\_aligned( ) ) |  |
| **ptl\_alignment\_zero\_bit** | f(1) |
| for( i = 0; i < maxNumSubLayersMinus1; i++ ) |  |
| if( sub\_layer\_level\_present\_flag[ i ] ) |  |
| **sub\_layer\_level\_idc**[ i ] | u(8) |
| } |  |

#### General constraint information syntax

|  |  |
| --- | --- |
| general\_constraint\_info( ) { | **Descriptor** |
| **general\_progressive\_source\_flag** | u(1) |
| **general\_interlaced\_source\_flag** | u(1) |
| **general\_non\_packed\_constraint\_flag** | u(1) |
| **general\_frame\_only\_constraint\_flag** | u(1) |
| **intra\_only\_constraint\_flag** | u(1) |
| **max\_bitdepth\_constraint\_idc** | u(4) |
| **max\_chroma\_format\_constraint\_idc** | u(2) |
| **frame\_only\_constraint\_flag** | u(1) |
| **no\_qtbtt\_dual\_tree\_intra constraint\_flag** | u(1) |
| **no\_sao\_constraint\_flag** | u(1) |
| **no\_alf\_constraint\_flag** | u(1) |
| **no\_pcm\_constraint\_flag** | u(1) |
| **no\_ref\_wraparound\_constraint\_flag** | u(1) |
| **no\_temporal\_mvp\_constraint\_flag** | u(1) |
| **no\_sbtmvp\_constraint\_flag** | u(1) |
| **no\_amvr\_constraint\_flag** | u(1) |
| **no\_bdof\_constraint\_flag** | u(1) |
| **no\_cclm\_constraint\_flag** | u(1) |
| **no\_mts\_constraint\_flag** | u(1) |
| **no\_affine\_motion\_constraint\_flag** | u(1) |
| **no\_gbi\_constraint\_flag** | u(1) |
| **no\_ciip\_constraint\_flag** | u(1) |
| **no\_triangle\_constraint\_flag** | u(1) |
| **no\_ladf\_constraint\_flag** | u(1) |
| **no\_cpr\_constraint\_flag** | u(1) |
| **no\_qp\_delta\_constraint\_flag** | u(1) |
| **no\_dep\_quant\_constraint\_flag** | u(1) |
| **no\_sign\_data\_hiding\_constraint\_flag** | u(1) |
| // ADD reserved bits for future extensions |  |
| while( !byte\_aligned( ) ) |  |
| **gci\_alignment\_zero\_bit** | f(1) |
| } |  |

### Tile group header syntax

#### General tile group header syntax

|  |  |
| --- | --- |
| tile\_group\_header( ) { | Descriptor |
| **tile\_group\_pic\_parameter\_set\_id** | ue(v) |
| if( rect\_tile\_group\_flag | | NumTilesInPic > 1 ) |  |
| **tile\_group\_address** | u(v) |
| if( !rect\_tile\_group\_flag && !single\_tile\_per\_tile\_group\_flag ) |  |
| **num\_tiles\_in\_tile\_group\_minus1** | ue(v) |
| **tile\_group\_type** | ue(v) |
| **tile\_group\_pic\_order\_cnt\_lsb** | u(v) |
| if( nal\_unit\_type != IDR\_W\_RADL && nal\_unit\_type != IDR\_N\_LP ) { |  |
| for( i = 0; i < 2; i++ ) { |  |
| if( num\_ref\_pic\_lists\_in\_sps[ i ] > 0 &&  ( i = = 0 | | ( i = = 1 && rpl1\_idx\_present\_flag ) ) ) |  |
| **ref\_pic\_list\_sps\_flag**[ i ] | u(1) |
| if( ref\_pic\_list\_sps\_flag[ i ] ) { |  |
| if( num\_ref\_pic\_lists\_in\_sps[ i ] > 1 && |  |
| ( i = = 0 | | ( i = = 1 && rpl1\_idx\_present\_flag ) ) ) |  |
| **ref\_pic\_list\_idx**[ i ] | u(v) |
| } else |  |
| ref\_pic\_list\_struct( i, num\_ref\_pic\_lists\_in\_sps[ i ] ) |  |
| for( j = 0; j < NumLtrpEntries[ i ][ RplsIdx[ i ] ]; j++ ) { |  |
| **delta\_poc\_msb\_present\_flag**[ i ][ j ] | u(1) |
| if( delta\_poc\_msb\_present\_flag[ i ][ j ] ) |  |
| **delta\_poc\_msb\_cycle\_lt**[ i ][ j ] | ue(v) |
| } |  |
| } |  |
| if( tile\_group\_type = = P | | tile\_group\_type = = B ) { |  |
| **num\_ref\_idx\_active\_override\_flag** | u(1) |
| if( num\_ref\_idx\_active\_override\_flag ) |  |
| for( i = 0; i < ( tile\_group\_type = = B ? 2: 1 ); i++ ) |  |
| if( num\_ref\_entries[ i ][ RplsIdx[ i ] ] > 1 ) |  |
| **num\_ref\_idx\_active\_minus1**[ i ] | ue(v) |
| } |  |
| } |  |
| if( partition\_constraints\_override\_enabled\_flag ) { |  |
| **partition\_constraints\_override\_flag** | ue(v) |
| if( partition\_constraints\_override\_flag ) { |  |
| **tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma** | ue(v) |
| **tile\_group\_max\_mtt\_hierarchy\_depth\_luma** | ue(v) |
| if( tile\_group\_max\_mtt\_hierarchy\_depth\_luma != 0 ) |  |
| **tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma** | ue(v) |
| **tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma** | ue(v) |
| } |  |
| if( tile\_group\_type = = I && qtbtt\_dual\_tree\_intra\_flag ) { |  |
| **tile\_group\_log2\_diff\_min\_qt\_min\_cb\_chroma** | ue(v) |
| **tile\_group\_max\_mtt\_hierarchy\_depth\_chroma** | ue(v) |
| if( tile\_group\_max\_mtt\_hierarchy\_depth\_chroma != 0 ) |  |
| **tile\_group\_log2\_diff\_max\_bt\_min\_qt\_chroma** | ue(v) |
| **tile\_group\_log2\_diff\_max\_tt\_min\_qt\_chroma** | ue(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| if ( tile\_group\_type != I ) { |  |
| if( sps\_temporal\_mvp\_enabled\_flag ) |  |
| **tile\_group\_temporal\_mvp\_enabled\_flag** | u(1) |
| if( tile\_group\_type = = B ) |  |
| **mvd\_l1\_zero\_flag** | u(1) |
| if( cabac\_init\_present\_flag ) |  |
| **cabac\_init\_flag** | u(1) |
| if( tile\_group\_temporal\_mvp\_enabled\_flag ) { |  |
| if( tile\_group\_type = = B ) |  |
| **collocated\_from\_l0\_flag** | u(1) |
| } |  |
| if( ( weighted\_pred\_flag && tile\_group\_type = = P ) | |  ( weighted\_bipred\_flag && tile\_group = = B ) ) |  |
| pred\_weight\_table( ) |  |
| **six\_minus\_max\_num\_merge\_cand** | ue(v) |
| if( sps\_affine\_enabled\_flag ) |  |
| **five\_minus\_max\_num\_subblock\_merge\_cand** | ue(v) |
| if( sps\_fpel\_mmvd\_enabled\_flag ) |  |
| **tile\_group\_fpel\_mmvd\_enabled\_flag** | **u(1)** |
| } else if ( sps\_ibc\_enabled\_flag ) |  |
| **six\_minus\_max\_num\_merge\_cand** | ue(v) |
| **tile\_group\_qp\_delta** | se(v) |
| if( pps\_tile\_group\_chroma\_qp\_offsets\_present\_flag ) { |  |
| **tile\_group\_cb\_qp\_offset** | se(v) |
| **tile\_group\_cr\_qp\_offset** | se(v) |
| } |  |
| if( sps\_sao\_enabled\_flag ) { |  |
| **tile\_group\_sao\_luma\_flag** | u(1) |
| if( ChromaArrayType != 0 ) |  |
| **tile\_group\_sao\_chroma\_flag** | u(1) |
| } |  |
| if( sps\_alf\_enabled\_flag ) { |  |
| **tile\_group\_alf\_enabled\_flag** | u(1) |
| if( tile\_group\_alf\_enabled\_flag ) |  |
| **tile\_group\_aps\_id** | u(5) |
| } |  |
| **dep\_quant\_enabled\_flag** | u(1) |
| if( !dep\_quant\_enabled\_flag ) |  |
| **sign\_data\_hiding\_enabled\_flag** | u(1) |
| if( deblocking\_filter\_override\_enabled\_flag ) |  |
| **deblocking\_filter\_override\_flag** | u(1) |
| if( deblocking\_filter\_override\_flag ) { |  |
| **tile\_group\_deblocking\_filter\_disabled\_flag** | u(1) |
| if( !tile\_group\_deblocking\_filter\_disabled\_flag ) { |  |
| **tile\_group\_beta\_offset\_div2** | se(v) |
| **tile\_group\_tc\_offset\_div2** | se(v) |
| } |  |
| } |  |
| if ( sps\_lmcs\_enabled\_flag ) { |  |
| **tile\_group\_lmcs\_model\_present\_flag** | u(1) |
| if ( tile\_group\_lmcs\_model\_present\_flag ) |  |
| lmcs\_data( ) |  |
| **tile\_group\_lmcs\_enabled\_flag** | u(1) |
| if ( tile\_group\_lmcs\_enabled\_flag &&   !( qtbtt\_dual\_tree\_intra\_flag && tile\_group\_type = = I ) ) |  |
| **tile\_group\_chroma\_residual\_scale\_flag** | u(1) |
| } |  |
| if( NumTilesInCurrTileGroup > 1 ) { |  |
| **offset\_len\_minus1** | ue(v) |
| for( i = 0; i < NumTilesInCurrTileGroup − 1; i++ ) |  |
| **entry\_point\_offset\_minus1**[ i ] | u(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Weighted prediction parameters syntax

|  |  |
| --- | --- |
| pred\_weight\_table( ) { | Descriptor |
| **luma\_log2\_weight\_denom** | ue(v) |
| if( ChromaArrayType != 0 ) |  |
| **delta\_chroma\_log2\_weight\_denom** | se(v) |
| for( i = 0; i < NumRefIdxActive[ 0 ]; i++ ) |  |
| **luma\_weight\_l0\_flag**[ i ] | u(1) |
| if( ChromaArrayType != 0 ) |  |
| for( i = 0; i < NumRefIdxActive[ 0 ]; i++ ) |  |
| **chroma\_weight\_l0\_flag**[ i ] | u(1) |
| for( i = 0; i < NumRefIdxActive[ 0 ]; i++ ) { |  |
| if( luma\_weight\_l0\_flag[ i ] ) { |  |
| **delta\_luma\_weight\_l0**[ i ] | se(v) |
| **luma\_offset\_l0**[ i ] | se(v) |
| } |  |
| if( chroma\_weight\_l0\_flag[ i ] ) |  |
| for( j = 0; j < 2; j++ ) { |  |
| **delta\_chroma\_weight\_l0**[ i ][ j ] | se(v) |
| **delta\_chroma\_offset\_l0**[ i ][ j ] | se(v) |
| } |  |
| } |  |
| if( tile\_group\_type = = B ) { |  |
| for( i = 0; i < NumRefIdxActive[ 1 ]; i++ ) |  |
| **luma\_weight\_l1\_flag**[ i ] | u(1) |
| if( ChromaArrayType != 0 ) |  |
| for( i = 0; i < NumRefIdxActive[ 1 ]; i++ ) |  |
| **chroma\_weight\_l1\_flag**[ i ] | u(1) |
| for( i = 0; i < NumRefIdxActive[ 1 ]; i++ ) { |  |
| if( luma\_weight\_l1\_flag[ i ] ) { |  |
| **delta\_luma\_weight\_l1**[ i ] | se(v) |
| **luma\_offset\_l1**[ i ] | se(v) |
| } |  |
| if( chroma\_weight\_l1\_flag[ i ] ) |  |
| for( j = 0; j < 2; j++ ) { |  |
| **delta\_chroma\_weight\_l1**[ i ][ j ] | se(v) |
| **delta\_chroma\_offset\_l1**[ i ][ j ] | se(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### Adaptive loop filter data syntax

|  |  |
| --- | --- |
| alf\_data( ) { | Descriptor |
| **alf\_chroma\_idc** | tu(v) |
| **alf\_luma\_num\_filters\_signalled\_minus1** | tb(v) |
| 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 < 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 < 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) |
| 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) |
| } |  |
| } |  |
| } |  |

#### Luma mapping with chroma scaling data syntax

|  |  |
| --- | --- |
| lmcs\_data () { | **Descriptor** |
| **lmcs\_min\_bin\_idx** | ue(v) |
| **lmcs\_delta\_max\_bin\_idx** | ue(v) |
| **lmcs\_delta\_cw\_prec\_minus1** | ue(v) |
| for ( i = lmcs\_min\_bin\_idx; i <= LmcsMaxBinIdx; i++ ) { |  |
| **lmcs\_delta\_abs\_cw**[ i ] | u(v) |
| if ( lmcs\_delta\_abs\_cw[ i ] ) > 0 ) |  |
| **lmcs\_delta\_sign\_cw\_flag**[ i ] | u(1) |
| } |  |
| } |  |

### Reference picture list structure syntax

|  |  |
| --- | --- |
| ref\_pic\_list\_struct( listIdx, rplsIdx ) { | Descriptor |
| **num\_ref\_entries**[ listIdx ][ rplsIdx ] | ue(v) |
| for( i = 0; i < num\_ref\_entries[ listIdx ][ rplsIdx ]; i++) { |  |
| if( long\_term\_ref\_pics\_flag ) |  |
| **st\_ref\_pic\_flag**[ listIdx ][ rplsIdx ][ i ] | u(1) |
| if( st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] ) { |  |
| **abs\_delta\_poc\_st**[ listIdx ][ rplsIdx ][ i ] | ue(v) |
| if( abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ] > 0 ) |  |
| **strp\_entry\_sign\_flag**[ listIdx ][ rplsIdx ][ i ] | u(1) |
| } else |  |
| **poc\_lsb\_lt**[ listIdx ][ rplsIdx ][ i ] | u(v) |
| } |  |
| } |  |

### Tile group data syntax

#### General tile group data syntax

|  |  |
| --- | --- |
| tile\_group\_data( ) { | Descriptor |
| for( i = 0; i < NumTilesInCurrTileGroup; i++, tileIdx++ ) { |  |
| ctbAddrInTs = FirstCtbAddrTs[ TgTileIdx[ i ] ] |  |
| for( j = 0; j < NumCtusInTile[ TgTileIdx[ i ] ]; j++, ctbAddrInTs++ ) { |  |
| if( ( j % ColWidth[ TgTileIdx[ i ] ] ) = = 0 ) { |  |
| NumHmvpCand = 0 |  |
| NumHmvpIbcCand = 0 |  |
| } |  |
| CtbAddrInRs = CtbAddrTsToRs[ ctbAddrInTs ] |  |
| coding\_tree\_unit( ) |  |
| } |  |
| **end\_of\_tile\_one\_bit** /\* equal to 1 \*/ | ae(v) |
| if( i < NumTilesInCurrTileGroup − 1 ) |  |
| byte\_alignment( ) |  |
| } |  |
| } |  |

#### Coding tree unit syntax

|  |  |
| --- | --- |
| coding\_tree\_unit( ) { | Descriptor |
| xCtb = ( CtbAddrInRs % PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| yCtb = ( CtbAddrInRs / PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| if( tile\_group\_sao\_luma\_flag | | tile\_group\_sao\_chroma\_flag ) |  |
| sao( xCtb  >>  CtbLog2SizeY, yCtb  >>  CtbLog2SizeY ) |  |
| if( tile\_group\_alf\_enabled\_flag ){ |  |
| **alf\_ctb\_flag**[ 0 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| 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( tile\_group\_type = = I  &&  qtbtt\_dual\_tree\_intra\_flag ) |  |
| dual\_tree\_implicit\_qt\_split ( xCtb, yCtb, CtbSizeY, 0 ) |  |
| else |  |
| coding\_tree( xCtb, yCtb, CtbSizeY, CtbSizeY, 1, 0, 0, 0, 0, 0, SINGLE\_TREE ) |  |
| } |  |

|  |  |
| --- | --- |
| dual\_tree\_implicit\_qt\_split( x0, y0, cbSize, cqtDepth ) { | Descriptor |
| cbSubdiv = 2 \* cqtDepth |  |
| if( cbSize > 64 ) { |  |
| if( cu\_qp\_delta\_enabled\_flag && cbSubdiv <= cu\_qp\_delta\_subdiv ) { |  |
| IsCuQpDeltaCoded = 0 |  |
| CuQpDeltaVal = 0 |  |
| CuQgTopLeftX = x0 |  |
| CuQgTopLeftY = y0 |  |
| } |  |
| x1 = x0 + ( cbSize / 2 ) |  |
| y1 = y0 + ( cbSize / 2 ) |  |
| dual\_tree\_implicit\_qt\_split( x0, y0, cbSize / 2, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x1, y0, cbSize / 2, cqtDepth + 1 ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x0, y1, cbSize / 2, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples && y1 < pic\_height\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x1, y1, cbSize / 2, cqtDepth + 1 ) |  |
| } else { |  |
| coding\_tree( x0, y0, cbSize, cbSize, 1, cbSubdiv, cqtDepth, 0, 0, 0,   DUAL\_TREE\_LUMA ) |  |
| coding\_tree( x0, y0, cbSize, cbSize, 0, cbSubdiv, cqtDepth, 0, 0, 0,   DUAL\_TREE\_CHROMA ) |  |
| } |  |
| } |  |

#### Sample adaptive offset syntax

|  |  |
| --- | --- |
| sao( rx, ry ) { | Descriptor |
| if( rx > 0 ) { |  |
| leftCtbInTile = TileId[ CtbAddrInTs ] = = TileId[ CtbAddrRsToTs[ CtbAddrInRs − 1 ] ] |  |
| if( leftCtbInTile ) |  |
| **sao\_merge\_left\_flag** | ae(v) |
| } |  |
| if( ry > 0 && !sao\_merge\_left\_flag ) { |  |
| upCtbInTile = TileId[ CtbAddrInTs ] = =   TileId[ CtbAddrRsToTs[ CtbAddrInRs − PicWidthInCtbsY ] ] |  |
| if( upCtbInTile ) |  |
| **sao\_merge\_up\_flag** | ae(v) |
| } |  |
| if( !sao\_merge\_up\_flag && !sao\_merge\_left\_flag ) |  |
| for( cIdx = 0; cIdx < ( ChromaArrayType != 0 ? 3 : 1 ); cIdx++ ) |  |
| if( ( tile\_group\_sao\_luma\_flag && cIdx = = 0 ) | |  ( tile\_group\_sao\_chroma\_flag && cIdx > 0 ) ) { |  |
| if( cIdx = = 0 ) |  |
| **sao\_type\_idx\_luma** | ae(v) |
| else if( cIdx = = 1 ) |  |
| **sao\_type\_idx\_chroma** | ae(v) |
| if( SaoTypeIdx[ cIdx ][ rx ][ ry ] != 0 ) { |  |
| for( i = 0; i < 4; i++ ) |  |
| **sao\_offset\_abs**[ cIdx ][ rx ][ ry ][ i ] | ae(v) |
| if( SaoTypeIdx[ cIdx ][ rx ][ ry ] = = 1 ) { |  |
| for( i = 0; i < 4; i++ ) |  |
| if( sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] != 0 ) |  |
| **sao\_offset\_sign**[ cIdx ][ rx ][ ry ][ i ] | ae(v) |
| **sao\_band\_position**[ cIdx ][ rx ][ ry ] | ae(v) |
| } else { |  |
| if( cIdx = = 0 ) |  |
| **sao\_eo\_class\_luma** | ae(v) |
| if( cIdx = = 1 ) |  |
| **sao\_eo\_class\_chroma** | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

[Ed. (BB): Adapt syntax once tiles are integrated.]

#### Coding tree syntax

|  |  |
| --- | --- |
| coding\_tree( x0, y0, cbWidth, cbHeight, qgOn, cbSubdiv, cqtDepth, mttDepth, depthOffset, partIdx,    treeType ) { | Descriptor |
| if( ( allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor | | allowSplitQT )   &&( x0 + cbWidth <= pic\_width\_in\_luma\_samples )   && (y0 + cbHeight <= pic\_height\_in\_luma\_samples ) ) |  |
| **split\_cu\_flag** | ae(v) |
| if( cu\_qp\_delta\_enabled\_flag && qgOn && cbSubdiv  <=  cu\_qp\_delta\_subdiv ) { |  |
| IsCuQpDeltaCoded = 0 |  |
| CuQpDeltaVal = 0 |  |
| CuQgTopLeftX = x0 |  |
| CuQgTopLeftY = y0 |  |
| } |  |
| if( split\_cu\_flag ) { |  |
| if( ( allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor ) &&   allowSplitQT ) |  |
| **split\_qt\_flag** | ae(v) |
| if( !split\_qt\_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 ) |  |
| coding\_tree( x0, y0, cbWidth / 2, cbHeight, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_tree( x1, y0, cbWidth / 2, cbHeightY, qgOn, cbSubdiv + 1,   cqtDepth, 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 ) |  |
| coding\_tree( x0, y0, cbWidth, cbHeight / 2, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_tree( x0, y1, cbWidth, cbHeight / 2, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_TT\_VER ) { |  |
| x1 = x0 + ( cbWidth / 4 ) |  |
| x2 = x0 + ( 3 \* cbWidth / 4 ) |  |
| qgOn = qgOn && ( cbSubdiv + 2 <= cu\_qp\_delta\_subdiv ) |  |
| coding\_tree( x0, y0, cbWidth / 4, cbHeight, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| coding\_tree( x1, y0, cbWidth / 2, cbHeight, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| coding\_tree( x2, y0, cbWidth / 4, cbHeight, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 2, treeType ) |  |
| } else { /\* SPLIT\_TT\_HOR \*/ |  |
| y1 = y0 + ( cbHeight / 4 ) |  |
| y2 = y0 + ( 3 \* cbHeight / 4 ) |  |
| qgOn = qgOn && ( cbSubdiv + 2 <= cu\_qp\_delta\_subdiv ) |  |
| coding\_tree( x0, y0, cbWidth, cbHeight / 4, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| coding\_tree( x0, y1, cbWidth, cbHeight / 2, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| coding\_tree( x0, y2, cbWidth, cbHeight / 4, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 2 , treeType) |  |
| } |  |
| } else { |  |
| x1 = x0 + ( cbWidth / 2 ) |  |
| y1 = y0 + ( cbHeight / 2 ) |  |
| coding\_tree( x0, y0, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1, 0, 0, 0, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_tree( x1, y0, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1, 0, 0, 1, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_tree( x0, y1, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1,  0, 0, 2, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples && x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_tree( x1, y1, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1,  0, 0, 3, treeType ) |  |
| } |  |
| } else |  |
| coding\_unit( x0, y0, cbWidth, cbHeight, treeType ) |  |
| } |  |

#### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, cbWidth, cbHeight, treeType ) { | Descriptor |
| if( tile\_group\_type != I | | sps\_ibc\_enabled\_flag ) { |  |
| if( treeType != DUAL\_TREE\_CHROMA ) |  |
| **cu\_skip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( cu\_skip\_flag[ x0 ][ y0 ] = = 0 && tile\_group\_type != I ) |  |
| **pred\_mode\_flag** | ae(v) |
| if( ( ( tile\_group\_type = = I && cu\_skip\_flag[ x0 ][ y0 ] = =0 ) | |  ( tile\_group\_type != I && CuPredMode[ x0 ][ y0 ] != MODE\_INTRA ) ) &&  sps\_ibc\_enabled\_flag ) |  |
| **pred\_mode\_ibc\_flag** | ae(v) |
| } |  |
| if( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA ) { |  |
| if( sps\_pcm\_enabled\_flag &&  cbWidth >= MinIpcmCbSizeY && cbWidth <= MaxIpcmCbSizeY &&  cbHeight >= MinIpcmCbSizeY && cbHeight <= MaxIpcmCbSizeY ) |  |
| **pcm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( pcm\_flag[ x0 ][ y0 ] ) { |  |
| while( !byte\_aligned( ) ) |  |
| **pcm\_alignment\_zero\_bit** | f(1) |
| pcm\_sample( cbWidth, cbHeight, treeType) |  |
| } else { |  |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| if( ( y0 % CtbSizeY )  >  0 ) |  |
| **intra\_luma\_ref\_idx**[ x0 ][ y0 ] | ae(v) |
| if (intra\_luma\_ref\_idx[ x0 ][ y0 ] = = 0 &&   ( cbWidth <= MaxTbSizeY | | cbHeight <= MaxTbSizeY ) &&   ( cbWidth \* cbHeight > MinTbSizeY \* MinTbSizeY )) |  |
| **intra\_subpartitions\_mode\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] = = 1 &&   cbWidth <= MaxTbSizeY && cbHeight <= MaxTbSizeY ) |  |
| **intra\_subpartitions\_split\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_luma\_ref\_idx[ x0 ][ y0 ] = = 0 &&   intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **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 if( treeType != DUAL\_TREE\_CHROMA ) { /\* MODE\_INTER or MODE\_IBC \*/ |  |
| if( cu\_skip\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **merge\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_flag[ x0 ][ y0 ] ) { |  |
| merge\_data( x0, y0, cbWidth, cbHeight ) |  |
| } else if ( CuPredMode[ x0 ][ y0 ] = = MODE\_IBC ) { |  |
| mvd\_coding( x0, y0, 0, 0 ) |  |
| **mvp\_l0\_flag**[ x0 ][ y0 ] | ae(v) |
| if( sps\_amvr\_enabled\_flag &&   ( MvdL0[ x0 ][ y0 ][ 0 ] != 0 | | MvdL0[ x0 ][ y0 ][ 1 ] != 0 ) ) { |  |
| **amvr\_precision\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } else { |  |
| if( tile\_group\_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\_BI && !inter\_affine\_flag[ x0 ][ y0 ] &&  RefIdxSymL0 > −1 && RefIdxSymL1 > −1 ) |  |
| **sym\_mvd\_flag**[ x0 ][ y0 ] | ae(v) |
| if( inter\_pred\_idc[ x0 ][ y0 ] != PRED\_L1 ) { |  |
| if( NumRefIdxActive[ 0 ] > 1 && !sym\_mvd\_flag[ x0 ][ y0 ] ) |  |
| **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( NumRefIdxActive[ 1 ] > 1 && !sym\_mvd\_flag[ x0 ][ y0 ] ) |  |
| **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 { |  |
| if( sym\_mvd\_flag[ x0 ][ y0 ] ) { |  |
| MvdL1[ x0 ][ y0 ][ 0 ] = −MvdL0[ x0 ][ y0 ][ 0 ] |  |
| MvdL1[ x0 ][ y0 ][ 1 ] = −MvdL0[ x0 ][ y0 ][ 1 ] |  |
| } 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 ) ) | |  ( sps\_affine\_amvr\_enabled\_flag && inter\_affine\_flag[ x0 ][ y0 ] = = 1 &&  ( MvdCpL0[ x0 ][ y0 ][ 0 ] [ 0 ] != 0 | | MvdCpL0[ x0 ][ y0 ][ 0 ] [ 1 ] != 0 | |  MvdCpL1[ x0 ][ y0 ][ 0 ] [ 0 ] != 0 | | MvdCpL1[ x0 ][ y0 ][ 0 ] [ 1 ] != 0 | |  MvdCpL0[ x0 ][ y0 ][ 1 ] [ 0 ] != 0 | | MvdCpL0[ x0 ][ y0 ][ 1 ] [ 1 ] != 0 | |  MvdCpL1[ x0 ][ y0 ][ 1 ] [ 0 ] != 0 | | MvdCpL1[ x0 ][ y0 ][ 1 ] [ 1 ] != 0 | |  MvdCpL0[ x0 ][ y0 ][ 2 ] [ 0 ] != 0 | | MvdCpL0[ x0 ][ y0 ][ 2 ] [ 1 ] != 0 | |  MvdCpL1[ x0 ][ y0 ][ 2 ] [ 0 ] != 0 | | MvdCpL1[ x0 ][ y0 ][ 2 ] [ 1 ] != 0 ) ) { |  |
| **amvr\_flag**[ x0 ][ y0 ] | ae(v) |
| if( amvr\_flag[ x0 ][ y0 ] ) |  |
| **amvr\_precision\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( sps\_gbi\_enabled\_flag && inter\_pred\_idc[ x0 ][ y0 ] = = PRED\_BI &&  luma\_weight\_l0\_flag[ ref\_idx\_l0 [ x0 ][ y0 ] ] = = 0 &&  luma\_weight\_l1\_flag[ ref\_idx\_l1 [ x0 ][ y0 ] ] = = 0 &&  chroma\_weight\_l0\_flag[ ref\_idx\_l0 [ x0 ][ y0 ] ] = = 0 &&  chroma\_weight\_l1\_flag[ ref\_idx\_l1 [ x0 ][ y0 ] ] = = 0 &&  cbWidth \* cbHeight >= 256 ) |  |
| **gbi\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| if( !pcm\_flag[ x0 ][ y0 ] ) { |  |
| if( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA && merge\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **cu\_cbf** | ae(v) |
| if( cu\_cbf ) { |  |
| if( CuPredMode[ x0 ][ y0 ] = = MODE\_INTER && sps\_sbt\_enabled\_flag &&   !ciip\_flag[ x0 ][ y0 ] ) { |  |
| if( cbWidth  <=  MaxSbtSize && cbHeight  <=  MaxSbtSize ) { |  |
| allowSbtVerH = cbWidth  >=  8 |  |
| allowSbtVerQ = cbWidth  >=  16 |  |
| allowSbtHorH = cbHeight  >=  8 |  |
| allowSbtHorQ = cbHeight  >=  16 |  |
| if( allowSbtVerH | | allowSbtHorH | | allowSbtVerQ | | allowSbtHorQ ) |  |
| **cu\_sbt\_flag** | ae(v) |
| } |  |
| if( cu\_sbt\_flag ) { |  |
| if( ( allowSbtVerH | | allowSbtHorH ) && ( allowSbtVerQ | | allowSbtHorQ) ) |  |
| **cu\_sbt\_quad\_flag** | ae(v) |
| if( ( cu\_sbt\_quad\_flag && allowSbtVerQ && allowSbtHorQ ) | |   ( !cu\_sbt\_quad\_flag && allowSbtVerH && allowSbtHorH ) ) |  |
| **cu\_sbt\_horizontal\_flag** | ae(v) |
| **cu\_sbt\_pos\_flag** | ae(v) |
| } |  |
| } |  |
| transform\_tree( x0, y0, cbWidth, cbHeight, treeType ) |  |
| } |  |
| } |  |
| } |  |

#### PCM sample syntax

|  |  |
| --- | --- |
| pcm\_sample( cbWidth, cbHeight, treeType ) { | **Descriptor** |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| for( i = 0; i < cbWidth \* cbHeight; i++ ) |  |
| **pcm\_sample\_luma**[ i ] | u(v) |
| } |  |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA) { |  |
| for( i = 0; i < 2 \* ( ( cbWidth \* cbHeight ) / ( SubWidthC \* SubHeightC ) ); i++ ) |  |
| **pcm\_sample\_chroma**[ i ] | u(v) |
| } |  |
| } |  |

#### Merge data syntax

|  |  |
| --- | --- |
| merge\_data( x0, y0, cbWidth, cbHeight ) { | Descriptor |
| if ( CuPredMode[ x0 ][ y0 ] = = MODE\_IBC ) { |  |
| if( MaxNumMergeCand > 1 ) |  |
| merge\_idx[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| **mmvd\_flag**[ x0 ][ y0 ] | ae(v) |
| if( mmvd\_flag[ x0 ][ y0 ] = = 1 ) { |  |
| **mmvd\_merge\_flag**[ x0 ][ y0 ] | ae(v) |
| **mmvd\_distance\_idx**[ x0 ][ y0 ] | ae(v) |
| **mmvd\_direction\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( MaxNumSubblockMergeCand > 0 && cbWidth >= 8 && cbHeight >= 8 ) |  |
| **merge\_subblock\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_subblock\_flag[ x0 ][ y0 ] = = 1 ) { |  |
| if( MaxNumSubblockMergeCand > 1 ) |  |
| **merge\_subblock\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( sps\_ciip\_enabled\_flag && cu\_skip\_flag[ x0 ][ y0 ] = = 0 &&   ( cbWidth \* cbHeight ) >= 64 && cbWidth < 128 && cbHeight < 128 ) { |  |
| **ciip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( ciip\_flag[ x0 ][ y0 ] ) { |  |
| if ( cbWidth <= 2 \* cbHeight | | cbHeight <= 2 \* cbWidth ) |  |
| **ciip\_luma\_mpm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( ciip\_luma\_mpm\_flag[ x0 ][ y0 ] ) |  |
| **ciip\_luma\_mpm\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| if( sps\_triangle\_enabled\_flag && tile\_group\_type = = B &&  ciip\_flag[ x0 ][ y0 ] = = 0 && cbWidth \* cbHeight >= 64 ) |  |
| **merge\_triangle\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_triangle\_flag[ x0 ][ y0 ] ) { |  |
| **merge\_triangle\_split\_dir**[ x0 ][ y0 ] | ae(v) |
| **merge\_triangle\_idx0**[ x0 ][ y0 ] | ae(v) |
| **merge\_triangle\_idx1**[ x0 ][ y0 ] | ae(v) |
| } else if( MaxNumMergeCand > 1 ) |  |
| **merge\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### 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 |
| InferTuCbfLuma = 1 |  |
| if( IntraSubPartSplitType = = NO\_ISP\_SPLIT ) { |  |
| 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, 0 ) |  |
| } |  |
| } else if( cu\_sbt\_flag ) { |  |
| if( !cu\_sbt\_horizontal\_flag ) { |  |
| trafoWidth = tbWidth \* SbtNumFourthsTb0 / 4 |  |
| transform\_unit( x0, y0, trafoWidth, tbHeight, treeType , 0 ) |  |
| transform\_unit( x0 + trafoWidth, y0, tbWidth − trafoWidth , tbHeight, treeType , 1 ) |  |
| } else { |  |
| trafoHeight = tbHeight \* SbtNumFourthsTb0 / 4 |  |
| transform\_unit( x0, y0, tbWidth, trafoHeight, treeType , 0 ) |  |
| transform\_unit( x0, y0 + trafoHeight, tbWidth, tbHeight − trafoHeight, treeType , 1 ) |  |
| } |  |
| } else if( IntraSubPartitionsSplitType = = ISP\_HOR\_SPLIT ) { |  |
| trafoHeight = tbHeight / NumIntraSubPartitions |  |
| for( partIdx = 0; partIdx < NumIntraSubPartitions; partIdx++ ) |  |
| transform\_unit( x0, y0 + trafoHeight \* partIdx, tbWidth, trafoHeight, treeType, partIdx ) |  |
| } else if( IntraSubPartitionsSplitType = = ISP\_VER\_SPLIT ) { |  |
| trafoWidth = tbWidth / NumIntraSubPartitions |  |
| for( partIdx = 0; partIdx < NumIntraSubPartitions; partIdx++ ) |  |
| transform\_unit( x0 + trafoWidth \* partIdx, y0, trafoWidth, tbHeight, treeType, partIdx ) |  |
| } |  |
| } |  |

#### Transform unit syntax

|  |  |
| --- | --- |
| transform\_unit( x0, y0, tbWidth, tbHeight, treeType, subTuIndex ) { | Descriptor |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| if( ( IntraSubPartitionsSplitType =  = ISP\_NO\_SPLIT && !( cu\_sbt\_flag &&   ( ( subTuIndex  = = 0  &&  cu\_sbt\_pos\_flag )  | |    ( subTuIndex  = = 1  &&  !cu\_sbt\_pos\_flag ) ) ) )  | |    ( IntraSubPartitionsSplitType != ISP\_NO\_SPLIT &&   ( subTuIndex < NumIntraSubPartitions − 1 | | !InferTuCbfLuma ) ) ) |  |
| **tu\_cbf\_luma**[ x0 ][ y0 ] | ae(v) |
| if (IntraSubPartitionsSplitType != ISP\_NO\_SPLIT ) |  |
| InferTuCbfLuma = InferTuCbfLuma && !tu\_cbf\_luma[ x0 ][ y0 ] |  |
| } |  |
| if( ( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA ) { |  |
| if( ( IntraSubPartitionsSplitType =  = ISP\_NO\_SPLIT && !( cu\_sbt\_flag &&   ( ( subTuIndex  = = 0  &&  cu\_sbt\_pos\_flag )  | |    ( subTuIndex  = = 1  &&  !cu\_sbt\_pos\_flag ) ) ) )  | |    ( IntraSubPartitionsSplitType != ISP\_NO\_SPLIT &&   ( subTuIndex = = NumIntraSubPartitions − 1 ) ) ) { |  |
| **tu\_cbf\_cb**[ x0 ][ y0 ] | ae(v) |
| **tu\_cbf\_cr**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| if( IntraSubPartitionsSplitType != ISP\_NO\_SPLIT &&   treeType = = SINGLE\_TREE && subTuIndex = = NumIntraSubPartitions − 1 ) ) |  |
| xC = CbPosX[ x0 ][ y0 ] |  |
| yC = CbPosY[ x0 ][ y0 ] |  |
| wC = CbWidth[ x0 ][ y0 ] / 2 |  |
| hC = CbHeight[ x0 ][ y0 ] / 2 |  |
| } else |  |
| xC = x0 |  |
| yC = y0 |  |
| wC = tbWidth / SubWidthC |  |
| hC = tbHeight / SubHeightC |  |
| } |  |
| if( ( tu\_cbf\_luma[ x0 ][ y0 ] | | tu\_cbf\_cb[ x0 ][ y0 ] | | tu\_cbf\_cr[ x0 ][ y0 ] )  &&  treeType != DUAL\_TREE\_CHROMA ) { |  |
| if( cu\_qp\_delta\_enabled\_flag && !IsCuQpDeltaCoded ) { |  |
| **cu\_qp\_delta\_abs** | ae(v) |
| if( cu\_qp\_delta\_abs ) |  |
| **cu\_qp\_delta\_sign\_flag** | ae(v) |
| } |  |
| } |  |
| if( tu\_cbf\_luma[ x0 ][ y0 ]   &&  treeType != DUAL\_TREE\_CHROMA  &&  ( tbWidth  <=  32 )  &&  ( tbHeight  <=  32 )   &&  ( IntraSubPartitionsSplit[ x0 ][ y0 ]  = =  ISP\_NO\_SPLIT ) &&  ( !cu\_sbt\_flag ) ) { |  |
| if( transform\_skip\_enabled\_flag && tbWidth <= MaxTsSize && tbHeight <= MaxTsSize ) |  |
| **transform\_skip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( (( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA && sps\_explicit\_mts\_inter\_enabled\_flag )   | | ( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA && sps\_explicit\_mts\_intra\_enabled\_flag ))  && ( tbWidth  <=  32 ) && ( tbHeight  <=  32 ) && ( !transform\_skip\_flag[ x0 ][ y0 ] ) ) |  |
| **tu\_mts\_idx**[ 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( xC, yC, Log2( wC ), Log2( hC ), 1 ) |  |
| if( tu\_cbf\_cr[ x0 ][ y0 ] ) |  |
| residual\_coding( xC, yC, Log2( wC ), Log2( hC ), 2 ) |  |
| } |  |

#### Residual coding syntax

|  |  |  |
| --- | --- | --- |
| residual\_coding( x0, y0, log2TbWidth, log2TbHeight, cIdx ) { | Descriptor | |
| if( ( tu\_mts\_idx[ x0 ][ y0 ] > 0 | |  ( cu\_sbt\_flag  &&  log2TbWidth < 6  &&  log2TbHeight < 6 ) ) && cIdx = = 0 && log2TbWidth > 4 ) |  | |
| log2TbWidth = 4 |  | |
| else |  | |
| log2TbWidth = Min( log2TbWidth, 5 ) |  | |
| if( tu\_mts\_idx[ x0 ][ y0 ] > 0 | |  ( cu\_sbt\_flag  &&  log2TbWidth < 6  &&  log2TbHeight < 6 ) )  && cIdx = = 0 && log2TbHeight > 4 ) |  | |
| log2TbHeight = 4 |  | |
| else |  | |
| log2TbHeight = Min( log2TbHeight, 5 ) |  | |
| if( log2TbWidth > 0 ) |  | |
| **last\_sig\_coeff\_x\_prefix** | ae(v) |
| if( log2TbHeight > 0 ) |  |
| **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) |
| log2SbW = ( Min( log2TbWidth, log2TbHeight ) < 2 ? 1 : 2 ) |  |
| log2SbH = log2SbW |  |
| if ( log2TbWidth < 2 && cIdx = = 0 ) { |  |
| log2SbW = log2TbWidth |  |
| log2SbH = 4 − log2SbW |  |
| } else if ( log2TbHeight < 2 && cIdx = = 0 ) { |  |
| log2SbH = log2TbHeight |  |
| log2SbW = 4 − log2SbH |  |
| } |  |
| numSbCoeff = 1 << ( log2SbW + log2SbH ) |  |
| lastScanPos = numSbCoeff |  |
| lastSubBlock = ( 1  <<  ( log2TbWidth + log2TbHeight − ( log2SbW + log2SbH ) ) ) − 1 |  |
| do { |  |
| if( lastScanPos = = 0 ) { |  |
| lastScanPos = numSbCoeff |  |
| lastSubBlock− − |  |
| } |  |
| lastScanPos− − |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ lastSubBlock ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ lastSubBlock ][ 1 ] |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ lastScanPos ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ lastScanPos ][ 1 ] |  |
| } while( ( xC != LastSignificantCoeffX ) | | ( yC != LastSignificantCoeffY ) ) |  |
| QState = 0 |  |
| for( i = lastSubBlock; i >= 0; i− − ) { |  |
| startQStateSb = QState |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ lastSubBlock ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ lastSubBlock ][ 1 ] |  |
| inferSbDcSigCoeffFlag = 0 |  |
| if( ( i < lastSubBlock ) && ( i > 0 ) ) { |  |
| **coded\_sub\_block\_flag**[ xS ][ yS ] | ae(v) |
| inferSbDcSigCoeffFlag = 1 |  |
| } |  |
| firstSigScanPosSb = numSbCoeff |  |
| lastSigScanPosSb = −1 |  |
| remBinsPass1 = ( ( log2SbW + log2SbH ) < 4 ? 8 : 32 ) |  |
| firstPosMode0 = ( i = = lastSubBlock ? lastScanPos : numSbCoeff − 1 ) |  |
| firstPosMode1 = −1 |  |
| for( n = firstPosMode0; n >= 0 && remBinsPass1 >= 4; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( coded\_sub\_block\_flag[ xS ][ yS ] && ( n > 0 | | !inferSbDcSigCoeffFlag ) &&   ( xC != LastSignificantCoeffX | | yC != Last SignificantCoeffY ) ) { |  |
| **sig\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| remBinsPass1− − |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) |  |
| inferSbDcSigCoeffFlag = 0 |  |
| } |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) { |  | |
| **abs\_level\_gt1\_flag**[ n ] | ae(v) |
| remBinsPass1− − |  |
| if( abs\_level\_gt1\_flag[ n ] ) { |  |
| **par\_level\_flag**[ n ] | ae(v) |
| remBinsPass1− − |  |
| **abs\_level\_gt3\_flag**[ n ] | ae(v) |
| remBinsPass1− − |  |
| } |  |
| if( lastSigScanPosSb = = −1 ) |  |
| lastSigScanPosSb = n |  |
| firstSigScanPosSb = n |  |
| } |  |
| AbsLevelPass1[ xC ][ yC ] = sig\_coeff\_flag[ xC ][ yC ] + par\_level\_flag[ n ] +   abs\_level\_gt1\_flag[ n ] + 2 \* abs\_level\_gt3\_flag[ n ] |  |
| if( dep\_quant\_enabled\_flag ) |  |
| QState = QStateTransTable[ QState ][ AbsLevelPass1[ xC ][ yC ] & 1 ] |  |
| if( remBinsPass1 < 4 ) |  |
| firstPosMode1 = n − 1 |  |
| } |  |
| for( n = numSbCoeff − 1; n >= firstPosMode1; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( abs\_level\_gt3\_flag[ n ] ) |  |
| **abs\_remainder**[ n ] | ae(v) |
| AbsLevel[ xC ][ yC ] = AbsLevelPass1[ xC ][ yC ] +2 \* abs\_remainder[ n ] |  |
| } |  |
| for( n = firstPosMode1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| **dec\_abs\_level**[ n ] | ae(v) |
| if(AbsLevel[ xC ][ yC ] > 0 ) |  |
| firstSigScanPosSb = n |  |
| if( dep\_quant\_enabled\_flag ) |  |
| QState = QStateTransTable[ QState ][ AbsLevel[ xC ][ yC ] & 1 ] |  |
| } |  |
| 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 << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( ( AbsLevel[ xC ][ yC ] > 0 ) &&   ( !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 << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( AbsLevel[ xC ][ yC ] > 0 ) |  |
| 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 << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( AbsLevel[ xC ][ yC ] > 0 ) { |  |
| 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 ] |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  | |

## 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.

NAL units that have nal\_unit\_type in the range of UNSPEC28..UNSPEC31, inclusive, for which semantics are not specified, shall not affect the decoding process specified in this Specification.

NOTE 1 – NAL unit types in the range of UNSPEC28..UNSPEC31 may be used as determined by the application. No decoding process for these values of nal\_unit\_type is specified in this Specification. Since different applications might use these NAL unit types for different purposes, particular care must be exercised in the design of encoders that generate NAL units with these nal\_unit\_type values, and in the design of decoders that interpret the content of NAL units with these nal\_unit\_type values. This Specification does not define any management for these values. These nal\_unit\_type values might only be suitable for use in contexts in which "collisions" of usage (i.e., different definitions of the meaning of the NAL unit content for the same nal\_unit\_type value) are unimportant, or not possible, or are managed – e.g., defined or managed in the controlling application or transport specification, or by controlling the environment in which bitstreams are distributed.

For purposes other than determining the amount of data in the decoding units of the bitstream (as specified in Annex C), decoders shall ignore (remove from the bitstream and discard) the contents of all NAL units that use reserved values of nal\_unit\_type.

NOTE 2 – This requirement allows future definition of compatible extensions to this Specification.

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 | TRAIL\_NUT | Coded tile group of a non- STSA trailing picture  tile\_group\_layer\_rbsp( ) | VCL |
| 1 | STSA\_NUT | Coded tile group of an STSA picture tile\_group\_layer\_rbsp( ) | VCL |
| 2 | RASL\_NUT | Coded tile group of a RASL picture tile\_group\_layer\_rbsp( ) | VCL |
| 3 | RADL\_NUT | Coded tile group of a RADL picture tile\_group\_layer\_rbsp( ) | VCL |
| 4..7 | RSV\_VCL\_4.. RSV\_VCL\_7 | Reserved non-IRAP VCL NAL unit types | VCL |
| 8 9 | IDR\_W\_RADL IDR\_N\_LP | Coded tile group of an IDR picture  tile\_group\_layer\_rbsp( ) | VCL |
| 10 | CRA\_NUT | Coded tile group of a CRA picture tile\_group\_layer\_rbsp( ) | VCL |
| 11 12 13 | RSV\_IRAP\_VCL11 RSV\_IRAP\_VCL12 RSV\_IRAP\_VCL13 | Reserved IRAP VCL NAL unit types | VCL |
| 14..15 | RSV\_VCL14.. RSV\_VCL15 | Reserved non-IRAP VCL NAL unit types | 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 | APS\_NUT | Adaptation parameter set adaptation\_parameter\_set\_rbsp( ) | non-VCL |
| 19 | AUD\_NUT | Access unit delimiter access\_unit\_delimiter\_rbsp( ) | non-VCL |
| 20 | EOS\_NUT | End of sequence end\_of\_seq\_rbsp( ) | non-VCL |
| 21 | EOB\_NUT | End of bitstream end\_of\_bitstream\_rbsp( ) | non-VCL |
| 22, 23 | PREFIX\_SEI\_NUT SUFFIX\_SEI\_NUT | Supplemental enhancement information sei\_rbsp( ) | non-VCL |
| 24..27 | RSV\_NVCL24.. RSV\_NVCL27 | Reserved non-VCL NAL unit types | non-VCL |
| 28..31 | UNSPEC28.. UNSPEC31 | Unspecified non-VCL NAL unit types | non-VCL |

NOTE 3 – A clean random access (CRA) picture may have associated RASL or RADL pictures present in the bitstream.

NOTE 4 – An instantaneous decoding refresh (IDR) picture having nal\_unit\_type equal to IDR\_N\_LP does not have associated leading pictures present in the bitstream. An IDR picture having nal\_unit\_type equal to IDR\_W\_RADL does not have associated RASL pictures present in the bitstream, but may have associated RADL pictures in the bitstream.

**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 in the range of IDR\_W\_RADL to RSV\_IRAP\_VCL13, inclusive, the coded tile group 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 APS\_NUT, TemporalId shall be equal to that of the access unit containing the NAL unit.

– 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'.

#### Encapsulation of an SODB within an RBSP (informative)

This clause does not form an integral part of this Specification.

The form of encapsulation of an SODB within an RBSP and the use of the emulation\_prevention\_three\_byte for encapsulation of an RBSP within a NAL unit is described for the following purposes:

– To prevent the emulation of start codes within NAL units while allowing any arbitrary SODB to be represented within a NAL unit,

– To enable identification of the end of the SODB within the NAL unit by searching the RBSP for the rbsp\_stop\_one\_bit starting at the end of the RBSP,

– To enable a NAL unit to have a size greater than that of the SODB under some circumstances (using one or more cabac\_zero\_word syntax elements).

The encoder can produce a NAL unit from an RBSP by the following procedure:

1. The RBSP data are searched for byte-aligned bits of the following binary patterns:

'00000000 00000000 000000xx' (where 'xx' represents any two-bit pattern: '00', '01', '10', or '11'),

and a byte equal to 0x03 is inserted to replace the bit pattern with the pattern:

'00000000 00000000 00000011 000000xx',

and finally, when the last byte of the RBSP data is equal to 0x00 (which can only occur when the RBSP ends in a cabac\_zero\_word), a final byte equal to 0x03 is appended to the end of the data. The last zero byte of a byte‑aligned three-byte sequence 0x000000 in the RBSP (which is replaced by the four-byte sequence 0x00000300) is taken into account when searching the RBSP data for the next occurrence of byte-aligned bits with the binary patterns specified above.

1. The resulting sequence of bytes is then prefixed with the NAL unit header, within which the nal\_unit\_type indicates the type of RBSP data structure in the NAL unit.

The process specified above results in the construction of the entire NAL unit.

This process can allow any SODB to be represented in a NAL unit while ensuring both of the following:

– No byte-aligned start code prefix is emulated within the NAL unit.

* No sequence of 8 zero-valued bits followed by a start code prefix, regardless of byte-alignment, is emulated within the NAL unit.

#### Order of NAL units and association to coded pictures, access units and coded video sequences

##### General

This clause specifies constraints on the order of NAL units in the bitstream.

Any order of NAL units in the bitstream obeying these constraints is referred to in the text as the decoding order of NAL units. Within a NAL unit, the syntax in clauses 7.3, D.2 and E.2 specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

##### Order of SPS, PPS and APS RBSPs and their activation

TBD.

##### Order of access units and association to CVSs

A bitstream conforming to this Specification consists of one or more CVSs.

A CVS consists of one or more access units. The order of NAL units and coded pictures and their association to access units is described in clause 7.4.2.4.4.

The first access unit of a CVS is an IRAP access unit with NoRaslOutputFlag equal to 1.

It is a requirement of bitstream conformance that, when present, the next access unit after an access unit that contains an end of sequence NAL unit or an end of bitstream NAL unit shall be an IRAP access unit, which may be an IDR access unit or a CRA access unit.

##### Order of NAL units and coded pictures and their association to access units

This clause specifies the order of NAL units and coded pictures and their association to access units for CVSs that conform to one or more of the profiles specified in Annex A and that are decoded using the decoding process specified in clauses 2 through 10.

An access unit consists of one coded picture, zero or more VCL NAL units and zero or more non-VCL NAL units. The association of VCL NAL units to coded pictures is described in clause 7.4.2.4.5.

The first access unit in the bitstream starts with the first NAL unit of the bitstream.

Let firstBlPicNalUnit be the first VCL NAL unit of a coded picture. The first of any of the following NAL units preceding firstBlPicNalUnit and succeeding the last VCL NAL unit preceding firstBlPicNalUnit, if any, specifies the start of a new access unit:

– access unit delimiter NAL unit (when present),

– SPS NAL unit (when present),

– PPS NAL unit (when present),

– Prefix SEI NAL unit (when present),

– NAL units with nal\_unit\_type in the range of RSV\_NVCL24..RSV\_NVCL25 (when present),

– NAL units with nal\_unit\_type in the range of UNSPEC28..UNSPEC29 (when present).

NOTE – The first NAL unit preceding firstBlPicNalUnit and succeeding the last VCL NAL unit preceding firstBlPicNalUnit, if any, can only be one of the above-listed NAL units.

When there is none of the above NAL units preceding firstBlPicNalUnit and succeeding the last VCL NAL preceding firstBlPicNalUnit, if any, firstBlPicNalUnit starts a new access unit.

The order of the coded pictures and non-VCL NAL units within an access unit shall obey the following constraints:

– When an access unit delimiter NAL unit is present, it shall be the first NAL unit. There shall be at most one access unit delimiter NAL unit in any access unit.

– When any SPS NAL units, PPS NAL units, prefix SEI NAL units, NAL units with nal\_unit\_type in the range of RSV\_NVCL24..RSV\_NVCL25, or NAL units with nal\_unit\_type in the range of UNSPEC28..UNSPEC29 are present, they shall not follow the last VCL NAL unit of the access unit.

– NAL units having nal\_unit\_type equal to SUFFIX\_SEI\_NUT or in the range of RSV\_NVCL26..RSV\_NVCL27, or in the range of UNSPEC30..UNSPEC31 shall not precede the first VCL NAL unit of the access unit.

– When an end of sequence NAL unit is present, it shall be the last NAL unit among all NAL units with in the access unit other than an end of bitstream NAL unit (when present).

– When an end of bitstream NAL unit is present, it shall be the last NAL unit in the access unit.

[Ed. A Figure should be added here]

##### Order of VCL NAL units and association to coded pictures

TBD.

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

#### Sequence parameter set RBSP semantics

**sps\_max\_sub\_layers\_minus1** plus 1 specifies the maximum number of temporal sub-layers that may be present in each CVS referring to the SPS. The value of sps\_max\_sub\_layers\_minus1 shall be in the range of 0 to 6, inclusive.

**sps\_reserved\_zero\_5bits** shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for sps\_reserved\_zero\_5bits are reserved for future use by ITU-T | ISO/IEC.

**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.

**log2\_max\_pic\_order\_cnt\_lsb\_minus4** specifies the value of the variable MaxPicOrderCntLsb that is used in the decoding process for picture order count as follows:

MaxPicOrderCntLsb = 2( log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 ) (7‑6)

The value of log2\_max\_pic\_order\_cnt\_lsb\_minus4 shall be in the range of 0 to 12, inclusive.

**sps\_max\_dec\_pic\_buffering\_minus1** plus 1 specifies the maximum required size of the decoded picture buffer for the CVS in units of picture storage buffers. The value of sps\_max\_dec\_pic\_buffering\_minus1 shall be in the range of 0 to MaxDpbSize − 1, inclusive, where MaxDpbSize is as specified somewhere else. [Ed. (YK): Change to reference to the clause specifying MaxDpbSize when available.]

**long\_term\_ref\_pics\_flag** equal to 0 specifies that no LTRP is used for inter prediction of any coded picture in the CVS. long\_term\_ref\_pics\_flag equal to 1 specifies that LTRPs may be used for inter prediction of one or more coded pictures in the CVS.

**rpl1\_same\_as\_rpl0\_flag** equal to 1 specifies that the syntax structures num\_ref\_pic\_lists\_in\_sps[ 1 ] and ref\_pic\_list\_struct( 1, rplsIdx ) are not present and the following applies:

– The value of num\_ref\_pic\_lists\_in\_sps[ 1 ] is inferred to be equal to the value of num\_ref\_pic\_lists\_in\_sps[ 0 ].

– The value of each of syntax elements in ref\_pic\_list\_struct( 1, rplsIdx ) is inferred to be equal to the value of corresponding syntax element in ref\_pic\_list\_struct( 0, rplsIdx ) for rplsIdx ranging from 0 to num\_ref\_pic\_lists\_in\_sps[ 0 ] − 1.

**num\_ref\_pic\_lists\_in\_sps**[ i ] specifies the number of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures with listIdx equal to i included in the SPS. The value of num\_ref\_pic\_lists\_in\_sps[ i ] shall be in the range of 0 to 64, inclusive.

NOTE 2 – For each value of listIdx (equal to 0 or 1), a decoder should allocate memory for a total number of num\_ref\_pic\_lists\_in\_sps[ i ] + 1 ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures since there may be one ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure directly signalled in the tile group headers of a current picture.

**qtbtt\_dual\_tree\_intra\_flag** equal to 1 specifies that for I tile groups, 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\_tree syntax structure for luma and chroma.

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

**log2\_min\_luma\_coding\_block\_size\_minus2** plus 2 specifies the minimum luma coding block size.

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

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

CtbSizeY = 1  <<  CtbLog2SizeY (7‑8)

MinCbLog2SizeY = log2\_min\_luma\_coding\_block\_size\_minus2 + 2 (7‑9)

MinCbSizeY = 1  <<  MinCbLog2SizeY (7‑10)

MinTbLog2SizeY = 2 (7‑11)

MaxTbLog2SizeY = 6 (7‑12)

MinTbSizeY = 1  <<  MinTbLog2SizeY (7‑13)

MaxTbSizeY = 1  <<  MaxTbLog2SizeY (7‑14)

PicWidthInCtbsY = Ceil( pic\_width\_in\_luma\_samples ÷ CtbSizeY ) (7‑15)

PicHeightInCtbsY = Ceil( pic\_height\_in\_luma\_samples ÷ CtbSizeY ) (7‑16)

PicSizeInCtbsY = PicWidthInCtbsY \* PicHeightInCtbsY (7‑17)

PicWidthInMinCbsY = pic\_width\_in\_luma\_samples / MinCbSizeY (7‑18)

PicHeightInMinCbsY = pic\_height\_in\_luma\_samples / MinCbSizeY (7‑19)

PicSizeInMinCbsY = PicWidthInMinCbsY \* PicHeightInMinCbsY (7‑20)

PicSizeInSamplesY = pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples (7‑21)

PicWidthInSamplesC = pic\_width\_in\_luma\_samples / SubWidthC (7‑22)

PicHeightInSamplesC = pic\_height\_in\_luma\_samples / SubHeightC (7‑23)

[Ed. (BB): Currently the maximum transform size (64x64 luma samples and corresponding chroma sample size) and the minimum transform size (4x4 luma samples and corresponding chroma samples) is fixed, 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 (7‑24)

CtbHeightC = CtbSizeY / SubHeightC (7‑25)

For log2BlockWidth ranging from 0 to 4 and for 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 inputs, and the output is assigned to DiagScanOrder[ log2BlockWidth ][ log2BlockHeight ].

**partition\_constraints\_override\_enabled\_flag** equal to 1 specifies the presence of partition\_constraints\_override\_flag in the tile group headers for tile groups referring to the SPS. partition\_constraints\_override\_enabled\_flag equal to 0 specifies the absence of partition\_constraints\_override\_ flag in the tile group headers for tile groups referring to the SPS.

**sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_luma** specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum coding block size in luma samples for luma CUs in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. The base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU is derived as follows:

MinQtLog2SizeIntraY = sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_luma + MinCbLog2SizeY (7‑26)

**sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_tile\_group** specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum luma coding block size in luma samples for luma CUs in tile groups with tile\_group\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_tile\_group shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. The base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU is derived as follows:

MinQtLog2SizeInterY = sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_tile\_group + MinCbLog2SizeY (7‑27)

**sps\_max\_mtt\_hierarchy\_depth\_inter\_tile\_group** specifies the default maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in tile groups with tile\_group\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default maximum hierarchy depth can be overridden by tile\_group\_max\_mtt\_hierarchy\_depth\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_max\_mtt\_hierarchy\_depth\_inter\_tile\_group shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

**sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_luma** specifies the default maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default maximum hierarchy depth can be overridden by tile\_group\_max\_mtt\_hierarchy\_depth\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

**sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_luma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraY, inclusive. When sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_luma is not present, the value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_luma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_luma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraY, inclusive. When sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_luma is not present, the value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_luma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_tile\_group** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in tile groups with tile\_group\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_tile\_group shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeInterY, inclusive. When sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_tile\_group is not present, the value of sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_tile\_group is inferred to be equal to 0.

**sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_tile\_group** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in tile groups with tile\_group\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_tile\_group shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeInterY, inclusive. When sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_tile\_group is not present, the value of sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_tile\_group is inferred to be equal to 0.

**sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_chroma** specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA and the base 2 logarithm of the minimum coding block size in luma samples for chroma CUs with treeType equal to DUAL\_TREE\_CHROMA in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_min\_qt\_min\_cb\_chroma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_chroma is inferred to be equal to 0. The base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a CTU with treeType equal to DUAL\_TREE\_CHROMA is derived as follows:

MinQtLog2SizeIntraC = sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_chroma + MinCbLog2SizeY (7‑28)

**sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_chroma** specifies the default maximum hierarchy depth for chroma coding units resulting from multi-type tree splitting of a chroma quadtree leaf with treeType equal to DUAL\_TREE\_CHROMA in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default maximum hierarchy depth can be overridden by tile\_group\_max\_mtt\_hierarchy\_depth\_chroma present in the tile group header of the tile groups referring to the SPS. The value of sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_chroma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_chroma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_max\_bt\_min\_qt\_chroma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraC, inclusive. When sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_chroma is not present, the value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_chroma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_chroma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in tile groups with tile\_group\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_flag is equal to 1, the default difference can be overridden by tile\_group\_log2\_diff\_max\_tt\_min\_qt\_chroma present in the tile group header of the tile groups referring to the SPS. The value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraC, inclusive. When sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_chroma is not present, the value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_chroma is inferred to be equal to 0.

**sps\_sao\_enabled\_flag** equal to 1 specifies that the sample adaptive offset process is applied to the reconstructed picture after the deblocking filter process. sps\_sao\_enabled\_flag equal to 0 specifies that the sample adaptive offset process is not applied to the reconstructed picture after the deblocking filter process.

**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\_pcm\_enabled\_flag** equal to 0 specifies that PCM-related syntax (pcm\_sample\_bit\_depth\_luma\_minus1, pcm\_sample\_‌bit\_depth\_chroma\_minus1, log2\_min\_pcm\_luma\_coding\_block\_size\_minus3, log2\_diff\_max\_min\_pcm\_luma\_‌coding\_block\_size, pcm\_loop\_filter\_disabled\_flag, pcm\_flag, pcm\_alignment\_zero\_bit syntax elements and pcm\_sample( ) syntax structure) is not present in the CVS.

NOTE 3 – When MinCbLog2SizeY is equal to 6 and sps\_pcm\_enabled\_flag is equal to 1, PCM sample data-related syntax (pcm\_flag, pcm\_alignment\_zero\_bit syntax elements and pcm\_sample( ) syntax structure) is not present in the CVS, because the maximum size of coding blocks that can convey PCM sample data-related syntax is restricted to be less than or equal to Min( CtbLog2SizeY, 5 ). Hence, MinCbLog2SizeY equal to 6 with sps\_pcm\_enabled\_flag equal to 1 is not an appropriate setting to convey PCM sample data in the CVS.

**pcm\_sample\_bit\_depth\_luma\_minus1** specifies the number of bits used to represent each of PCM sample values of the luma component as follows:

PcmBitDepthY = pcm\_sample\_bit\_depth\_luma\_minus1 + 1 (7‑29)

The value of PcmBitDepthY shall be less than or equal to the value of BitDepthY.

**pcm\_sample\_bit\_depth\_chroma\_minus1** specifies the number of bits used to represent each of PCM sample values of the chroma components as follows:

PcmBitDepthC = pcm\_sample\_bit\_depth\_chroma\_minus1 + 1 (7‑30)

The value of PcmBitDepthC shall be less than or equal to the value of BitDepthC. When ChromaArrayType is equal to 0, pcm\_sample\_bit\_depth\_chroma\_minus1 is not used in the decoding process and decoders shall ignore its value.

**log2\_min\_pcm\_luma\_coding\_block\_size\_minus3** plus 3 specifies the minimum size of coding blocks with pcm\_flag equal to 1.

The variable Log2MinIpcmCbSizeY is set equal to log2\_min\_pcm\_luma\_coding\_block\_size\_minus3 + 3. The value of Log2MinIpcmCbSizeY shall be in the range of Min( MinCbLog2SizeY, 5 ) to Min( CtbLog2SizeY, 5 ), inclusive.

**log2\_diff\_max\_min\_pcm\_luma\_coding\_block\_size** specifies the difference between the maximum and minimum size of coding blocks with pcm\_flag equal to 1.

The variable Log2MaxIpcmCbSizeY is set equal to log2\_diff\_max\_min\_pcm\_luma\_coding\_block\_size + Log2MinIpcmCbSizeY. The value of Log2MaxIpcmCbSizeY shall be less than or equal to Min( CtbLog2SizeY, 5 ).

**pcm\_loop\_filter\_disabled\_flag** specifies whether the loop filter process is disabled on reconstructed samples in a coding unit with pcm\_flag equal to 1 as follows:

– If pcm\_loop\_filter\_disabled\_flag is equal to 1, the deblocking filter and sample adaptive offset filter processes on the reconstructed samples in a coding unit with pcm\_flag equal to 1 are disabled.

– Otherwise (pcm\_loop\_filter\_disabled\_flag value is equal to 0), the deblocking filter and sample adaptive offset filter processes on the reconstructed samples in a coding unit with pcm\_flag equal to 1 are not disabled.

When pcm\_loop\_filter\_disabled\_flag is not present, it is inferred to be equal to 0.

**sps\_ref\_wraparound\_enabled\_flag** equal to 1 specifies that horizontal wrap-around motion compensation is applied in inter prediction. sps\_ref\_wraparound\_enabled\_flag equal to 0 specifies that horizontal wrap-around motion compensation is not applied.

**sps\_ref\_wraparound\_offset\_minus1** plus 1 specifies the offset used for computing the horizontal wrap-around position in units of MinCbSizeY luma samples. The value of ref\_wraparound\_offset\_minus1 shall be in the range of 0 to ( pic\_width\_in\_luma\_samples / MinCbSizeY ) − 1, inclusive.

**sps\_temporal\_mvp\_enabled\_flag** equal to 1 specifies that tile\_group\_temporal\_mvp\_enabled\_flag is present in the tile group headers of tile groups with tile\_group\_type not equal to I in the CVS. sps\_temporal\_mvp\_enabled\_flag equal to 0 specifies that tile\_group\_temporal\_mvp\_enabled\_flag is not present in tile group 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 tile groups having tile\_group\_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.

**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\_bdof\_enabled\_flag** equal to 0 specifies that the bidirectional optical flow inter prediction is disabled. sps\_bdof\_enabled\_flag equal to 1 specifies that the bidirectional optical flow inter prediction is enabled.

**sps\_affine\_amvr\_enabled\_flag** equal to 1 specifies that adaptive motion vector difference resolution is used in motion vector coding of affine inter mode. sps\_affine\_amvr\_enabled\_flag equal to 0 specifies that adaptive motion vector difference resolution is not used in motion vector coding of affine inter mode.

**sps\_dmvr\_enabled\_flag** equal to 1 specifies that decoder motion vector refinement based inter bi-prediction is enabled. sps\_dmvr\_enabled\_flag equal to 0 specifies that decoder motion vector refinement based inter bi-prediction is disabled.

**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\_cclm\_colocated\_chroma\_flag** equal to 1 specifies that the top-left downsampled luma sample in cross-component linear model intra prediction is collocated with the top-left luma sample. sps\_cclm\_colocated\_chroma\_flag equal to 0 specifies that the top-left downsampled luma sample in cross-component linear model intra prediction is horizontally co-sited with the top-left luma sample but vertically shifted by 0.5 units of luma samples relatively to the top-left luma sample.

**sps\_mts\_enabled\_flag** equal to 1 specifies that sps\_explicit\_mts\_intra\_enabled\_flag is present in the sequence parameter set RBSP syntax and that sps\_explicit\_mts\_inter\_enabled\_flag is present in the sequence parameter set RBSP syntax. sps\_mts\_enabled\_flag equal to 0 specifies that sps\_explicit\_mts\_intra\_enabled\_flag is not present in the sequence parameter set RBSP syntax and that sps\_explicit\_mts\_inter\_enabled\_flag is not present in the sequence parameter set RBSP syntax.

**sps\_explicit\_mts\_intra\_enabled\_flag** equal to 1 specifies that tu\_mts\_idx may be present in the transform unit syntax for intra coding units. sps\_explicit\_mts\_intra\_enabled\_flag equal to 0 specifies that tu\_mts\_idx is not present in the transform unit syntax for intra coding units. When not present, the value of sps\_explicit\_mts\_intra\_enabled\_flag is inferred to be equal to 0.

**sps\_explicit\_mts\_inter\_enabled\_flag** equal to 1 specifies that tu\_mts\_idx may be present in the transform unit syntax for inter coding units. sps\_explicit\_mts\_inter\_enabled\_flag equal to 0 specifies that tu\_mts\_idx is not present in the transform unit syntax for inter coding units. When not present, the value of sps\_explicit\_mts\_inter\_enabled\_flag is inferred to be equal to 0.

**sps\_sbt\_enabled\_flag** equal to 0 specifies that subblock transform for inter-predicted CUs is disabled. sps\_sbt\_enabled\_flag equal to 1 specifies that subblock transform for inter-predicteds CU is enabled.

**sps\_sbt\_max\_size\_64\_flag** equal to 0 specifies that the maximum CU width and height for allowing subblock transform is 32 luma samples. sps\_sbt\_max\_size\_64\_flag equal to 1 specifies that the maximum CU width and height for allowing subblock transform is 64 luma samples.

MaxSbtSize = sps\_sbt\_max\_size\_64\_flag ? 64 : 32 (7‑31)

**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 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\_gbi\_enabled\_flag** specifies whether bi-prediction with CU weights can be used for inter prediction. If sps\_gbi\_enabled\_flag is equal to 0, the syntax shall be constrained such that no bi-prediction with CU weights is used in the CVS, and gbi\_idx is not present in coding unit syntax of the CVS. Otherwise (sps\_gbi\_enabled\_flag is equal to 1), bi-prediction with CU weights can be used in the CVS.

**sps\_ibc\_enabled\_flag** equal to 1 specifies that current picture referencing may be used in decoding of pictures in the CVS. sps\_ibc\_enabled\_flag equal to 0 specifies that current picture referencing is not used in the CVS. When sps\_ibc\_enabled\_flag is not present, it is inferred to be equal to 0.

**sps\_ciip\_enabled\_flag** specifies that ciip\_flag may be present in the coding unit syntax for inter coding units. sps\_ciip\_enabled\_flag equal to 0 specifies that ciip\_flag is not present in the coding unit syntax for inter coding units.

**sps\_fpel\_mmvd\_enabled\_flag** equal to 1 specifies that merge mode with motion vector difference is using integer sample precision. sps\_fpel\_mmvd\_enabled\_flag equal to 0 specifies that merge mode with motion vector difference can use fractional sample precision.

**sps\_triangle\_enabled\_flag** specifies whether triangular shape based motion compensation can be used for inter prediction. sps\_triangle\_enabled\_flag equal to 0 specifies that the syntax shall be constrained such that no triangular shape based motion compensation is used in the CVS, and merge\_triangle\_flag, merge\_triangle\_split\_dir, merge\_triangle\_idx0, and merge\_triangle\_idx1 are not present in coding unit syntax of the CVS. sps\_triangle\_enabled\_flag equal to 1 specifies that triangular shape based motion compensation can be used in the CVS.

**sps\_lmcs\_enabled\_flag** equal to 1 specifies that luma mapping with chroma scaling is used in the CVS. sps\_lmcs\_enabled\_flag equal to 0 specifies that luma mapping with chroma scaling is not used in the CVS.

**sps\_ladf\_enabled\_flag** equal to 1, specifies that sps\_num\_ladf\_intervals\_minus2, sps\_ladf\_lowest\_interval\_qp\_offset, sps\_ladf\_qp\_offset[ i ], and sps\_ladf\_delta\_threshold\_minus1[ i ] are present in the SPS.

**sps\_num\_ladf\_intervals\_minus2** plus 1 specifies the number of sps\_ladf\_delta\_threshold\_minus1[ i ] and sps\_ladf\_qp\_offset[ i ] syntax elements that are present in the SPS. The value of sps\_num\_ladf\_intervals\_minus2 shall be in the range of 0 to 3, inclusive.

**sps\_ladf\_lowest\_interval\_qp\_offset** specifies the offset used to derive the variable qP as specified in clause 8.8.2.6.3. The value of sps\_ladf\_lowest\_interval\_qp\_offset shall be in the range of 0 to 63, inclusive.

**sps\_ladf\_qp\_offset**[ i ] specifies the offset array used to derive the variable qP as specified in clause 8.8.2.6.3. The value of sps\_ladf\_qp\_offset[ i ] shall be in the range of 0 to 63, inclusive.

**sps\_ladf\_delta\_threshold\_minus1**[ i ] is used to compute the values of SpsLadfIntervalLowerBound[ i ], which specifies the lower bound of the i-th luma intensity level interval. The value of sps\_ladf\_delta\_threshold\_minus1[ i ] shall be in the range of 0 to 2BitDepthY − 3, inclusive.

The value of SpsLadfIntervalLowerBound[ 0 ] is set equal to 0.

For each value of i in the range of 0 to sps\_num\_ladf\_intervals\_minus2, inclusive, the variable SpsLadfIntervalLowerBound[ i + 1 ] is derived as follows:

SpsLadfIntervalLowerBound[ i + 1 ] = SpsLadfIntervalLowerBound[ i ] (7‑32)  
 + sps\_ladf\_delta\_threshold\_minus1[ i ] + 1

**sps\_extension\_flag** equal to 0 specifies that no sps\_extension\_data\_flag syntax elements are present in the SPS RBSP syntax structure. sps\_extension\_flag equal to 1 specifies that there are sps\_extension\_data\_flag syntax elements present in the SPS RBSP syntax structure.

**sps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all sps\_extension\_data\_flag syntax elements.

#### 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.

**single\_tile\_in\_pic\_flag** equal to 1 specifies that there is only one tile in each picture referring to the PPS. single\_tile\_in\_pic\_flag equal to 0 specifies that there is more than one tile in each picture referring to the PPS.

It is a requirement of bitstream conformance that the value of single\_tile\_in\_pic\_flag shall be the same for all PPSs that are activated within a CVS.

**num\_tile\_columns\_minus1** plus 1 specifies the number of tile columns partitioning the picture. num\_tile\_columns\_minus1 shall be in the range of 0 to PicWidthInCtbsY − 1, inclusive. When not present, the value of num\_tile\_columns\_minus1 is inferred to be equal to 0.

**num\_tile\_rows\_minus1** plus 1 specifies the number of tile rows partitioning the picture. num\_tile\_rows\_minus1 shall be in the range of 0 to PicHeightInCtbsY − 1, inclusive. When not present, the value of num\_tile\_rows\_minus1 is inferred to be equal to 0.

The variable NumTilesInPic is set equal to ( num\_tile\_columns\_minus1 + 1 ) \* ( num\_tile\_rows\_minus1 + 1 ).

When single\_tile\_in\_pic\_flag is equal to 0, NumTilesInPic shall be greater than 1.

**uniform\_tile\_spacing\_flag** equal to 1 specifies that tile column boundaries and likewise tile row boundaries are distributed uniformly across the picture. uniform\_tile\_spacing\_flag equal to 0 specifies that tile column boundaries and likewise tile row boundaries are not distributed uniformly across the picture but signalled explicitly using the syntax elements tile\_column\_width\_minus1[ i ] and tile\_row\_height\_minus1[ i ]. When not present, the value of uniform\_tile\_spacing\_flag is inferred to be equal to 1.

**tile\_column\_width\_minus1**[ i ] plus 1 specifies the width of the i-th tile column in units of CTBs.

**tile\_row\_height\_minus1**[ i ] plus 1 specifies the height of the i-th tile row in units of CTBs.

The following variables are derived by invoking the CTB raster and tile scanning conversion process as specified in clause 6.5.1:

– The list ColWidth[ i ] for i ranging from 0 to num\_tile\_columns\_minus1, inclusive, specifying the width of the i-th tile column in units of CTBs,

– the list RowHeight[ j ] for j ranging from 0 to num\_tile\_rows\_minus1, inclusive, specifying the height of the j-th tile row in units of CTBs,

– the list ColBd[ i ] for i ranging from 0 to num\_tile\_columns\_minus1 + 1, inclusive, specifying the location of the i-th tile column boundary in units of CTBs,

– the list RowBd[ j ] for j ranging from 0 to num\_tile\_rows\_minus1 + 1, inclusive, specifying the location of the j-th tile row boundary in units of CTBs,

– the list CtbAddrRsToTs[ ctbAddrRs ] for ctbAddrRs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in the CTB raster scan of a picture to a CTB address in the tile scan,

– the list CtbAddrTsToRs[ ctbAddrTs ] for ctbAddrTs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in the tile scan to a CTB address in the CTB raster scan of a picture,

– the list TileId[ ctbAddrTs ] for ctbAddrTs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in tile scan to a tile ID,

– the list NumCtusInTile[ tileIdx ] for tileIdx ranging from 0 to NumTilesInPic − 1, inclusive, specifying the conversion from a tile index to the number of CTUs in the tile,

– the list FirstCtbAddrTs[ tileIdx ] for tileIdx ranging from 0 to NumTilesInPic − 1, inclusive, specifying the conversion from a tile ID to the CTB address in tile scan of the first CTB in the tile,

– the lists ColumnWidthInLumaSamples[ i ] for i ranging from 0 to num\_tile\_columns\_minus1, inclusive, specifying the width of the i-th tile column in units of luma samples,

– the list RowHeightInLumaSamples[ j ] for j ranging from 0 to num\_tile\_rows\_minus1, inclusive, specifying the height of the j-th tile row in units of luma samples.

The values of ColumnWidthInLumaSamples[ i ] for i ranging from 0 to num\_tile\_columns\_minus1, inclusive, and RowHeightInLumaSamples[ j ] for j ranging from 0 to num\_tile\_rows\_minus1, inclusive, shall all be greater than 0. [Ed. (YK): ColumnWidthInLumaSamples[ i ] and RowHeightInLumaSamples[ j ] are currently defined but not used. However, in HEVC, these are used in profile definitions and the semantics of the temporal MCTS SEI messages. Could be similar for VVC. Thus these are kept in the draft text for now.]

**single\_tile\_per\_tile\_group** equal to 1 specifies that each tile group that refers to this PPS includes one tile. single\_tile\_per\_tile\_group equal to 0 specifies that a tile group that refers to this PPS may include more than one tile.

**rect\_tile\_group\_flag** equal to 0 specifies that tiles within each tile group are in raster scan order and the tile group information is not signalled in PPS. rect\_tile\_group\_flag equal to 1 specifies that tiles within each tile group cover a rectangular region of the picture and the tile group information is signalled in the PPS. When single\_tile\_per\_tile\_group\_flag is equal to 1 rect\_tile\_group\_flag is inferred to be equal to 1.

**num\_tile\_groups\_in\_pic\_minus1** plus 1 specifies the number of tile groups in each picture referring to the PPS. The value of num\_tile\_groups\_in\_pic\_minus1 shall be in the range of 0 to NumTilesInPic − 1, inclusive. When not present and single\_tile\_per\_tile\_group\_flag is equal to 1, the value of num\_tile\_groups\_in\_pic\_minus1 is inferred to be equal to NumTilesInPic − 1.

**top\_left\_tile\_idx**[ i ] specifies the tile index of the tile located at the top-left corner of the i-th tile group. The value of top\_left\_tile\_idx[ i ] shall not be equal to the value of top\_left\_tile\_idx[ j ] for any i not equal to j. When not present, the value of top\_left\_tile\_idx[ i ] is inferred to be equal to i. The length of the top\_left\_tile\_idx[ i ] syntax element is Ceil( Log2( NumTilesInPic ) bits.

**bottom\_right\_tile\_idx**[ i ] specifies the tile index of the tile located at the bottom-right corner of the i-th tile group. When single\_tile\_per\_tile\_group\_flag is equal to 1 bottom\_right\_tile\_idx[ i ] is inferred to be equal to top\_left\_tile\_idx[ i ]. The length of the bottom\_right\_tile\_idx[ i ] syntax element is Ceil( Log2( NumTilesInPic) ) bits.

It is a requirement of bitstream conformance that any particular tile shall only be included in one tile group.

The variable NumTilesInTileGroup[ i ], which specifies the number of tiles in the i-th tile group, and related variables, are derived as follows:

deltaTileIdx = bottom\_right\_tile\_idx[ i ] – top\_left\_tile\_idx[ i ]  
NumTileRowsInTileGroupMinus1[ i ] = deltaTileIdx / ( num\_tile\_columns\_minus1 + 1 )  
NumTileColumnsInTileGroupMinus1[ i ] = deltaTileIdx % ( num\_tile\_columns\_minus1 + 1 ) (7‑33)  
NumTilesInTileGroup[ i ] = ( NumTileRowsInTileGroupMinus1[ i ] + 1 ) \*  
 ( NumTileColumnsInTileGroupMinus1[ i ] + 1 )

**loop\_filter\_across\_tiles\_enabled\_flag** equal to 1 specifies that in-loop filtering operations may be performed across tile boundaries in pictures referring to the PPS. loop\_filter\_across\_tiles\_enabled\_flag equal to 0 specifies that in-loop filtering operations are not performed across tile boundaries in pictures referring to the PPS. The in-loop filtering operations include the deblocking filter, sample adaptive offset filter, and adaptive loop filter operations. When not present, the value of loop\_filter\_across\_tiles\_enabled\_flag is inferred to be equal to 1.

**loop\_filter\_across\_tile\_groups\_enabled\_flag** equal to 1 specifies that in-loop filtering operations may be performed across tile group boundaries in pictures referring to the PPS. loop\_filter\_across\_tile\_group\_enabled\_flag equal to 0 specifies that in-loop filtering operations are not performed across tile group boundaries in pictures referring to the PPS. The in-loop filtering operations include the deblocking filter, sample adaptive offset filter, and adaptive loop filter operations. When not present, the value of loop\_filter\_across\_tile\_groups\_enabled\_flag is inferred to be equal to 0.

**signalled\_tile\_group\_id\_flag** equal to 1 specifies that the tile group ID for each tile group is signalled. signalled\_tile\_group\_id\_flag equal to 0 specifies that tile group IDs are not signalled. When rect\_tile\_group\_flag is equal to 0, the value of signalled\_tile\_group\_id\_flag is inferred to be equal to 0.

**signalled\_tile\_group\_id\_length\_minus1** plus 1 specifies the number of bits used to represent the syntax element tile\_group\_id[ i ] when present, and the syntax element tile\_group\_address in tile group headers. The value of signalled\_tile\_group\_id\_length\_minus1 shall be in the range of 0 to 15, inclusive. When not present, the value of signalled\_tile\_group\_id\_length\_minus1 is inferred to be equal to Ceil( Log2( num\_tile\_groups\_in\_pic\_minus1 + 1 ) ) − 1.

**tile\_group\_id**[ i ] specifies the tile group ID of the i-th tile group. The length of the tile\_group\_id[ i ] syntax element is signalled\_tile\_group\_id\_length\_minus1 + 1 bits. When not present, the value of tile\_group\_id[ i ] is inferred to be equal to i, for each i in the range of 0 to num\_tile\_groups\_in\_pic\_minus1, inclusive.

**cabac\_init\_present\_flag** equal to 1 specifies that cabac\_init\_flag is present in tile group headers referring to the PPS. cabac\_init\_present\_flag equal to 0 specifies that cabac\_init\_flag is not present in tile group headers referring to the PPS.

**num\_ref\_idx\_default\_active\_minus1**[ i ] plus 1, when i is equal to 0, specifies the inferred value of the variable NumRefIdxActive[ 0 ] for P or B tile groups with num\_ref\_idx\_active\_override\_flag equal to 0, and, when i is equal to 1, specifies the inferred value of NumRefIdxActive[ 1 ] for B tile groups with num\_ref\_idx\_active\_override\_flag equal to 0. The value of num\_ref\_idx\_default\_active\_minus1[ i ] shall be in the range of 0 to 14, inclusive.

**rpl1\_idx\_present\_flag** equal to 0 specifies that ref\_pic\_list\_sps\_flag[ 1 ] and ref\_pic\_list\_idx[ 1 ] are not present in tile group headers. rpl1\_idx\_present\_flag equal to 1 specifies that ref\_pic\_list\_sps\_flag[ 1 ] and ref\_pic\_list\_idx[ 1 ] may be present in tile group headers.

**init\_qp\_minus26** plus 26 specifies the initial value of TileGroupQpY for each tile group referring to the PPS. The initial value of TileGroupQpY is modified at the tile group layer when a non-zero value of tile\_group\_qp\_delta is decoded. The value of init\_qp\_minus26 shall be in the range of −( 26 + QpBdOffsetY ) to +37, inclusive.

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

**log2\_transform\_skip\_max\_size\_minus2** specifies the maximum block size used for transform skip, and shall be in the range of 0 to 3.

When not present, the value of log2\_transform\_skip\_max\_size\_minus2 is inferred to be equal to 0.

The variable MaxTsSize is set equal to 1 << (log2\_transform\_skip\_max\_size\_minus2 + 2).

**cu\_qp\_delta\_enabled\_flag** equal to 1 specifies that the cu\_qp\_delta\_subdiv syntax element is present in the PPS and that cu\_qp\_delta\_abs may be present in the transform unit syntax. cu\_qp\_delta\_enabled\_flag equal to 0 specifies that the cu\_qp\_delta\_subdiv syntax element is not present in the PPS and that cu\_qp\_delta\_abs is not present in the transform unit syntax.

**cu\_qp\_delta\_subdiv** specifies the maximum cbSubdiv value of coding units that convey cu\_qp\_delta\_abs and cu\_qp\_delta\_sign\_flag. The value range of cu\_qp\_delta\_subdiv is specified as follows:

* If tile\_group\_type is equal to I, the value of cu\_qp\_delta\_subdiv shall be in the range of 0 to 2 \* ( log2\_ctu\_size\_minus2 − log2\_min\_qt\_size\_intra\_tile\_group\_minus2 + MaxMttDepthY ), inclusive.
* Otherwise (tile\_group\_type is not equal to I), the value of cu\_qp\_delta\_subdiv shall be in the range of 0 to 2\* ( log2\_ctu\_size\_minus2 − log2\_min\_qt\_size\_inter\_tile\_group\_minus2 + MaxMttDepthY ), inclusive.

When not present, the value of cu\_qp\_delta\_subdiv is inferred to be equal to 0.

[Ed. (BB): The issue here is that MaxMttDepthY is derived on tile-group-level. In case of partition\_constraints\_override\_enabled\_flag equal to 1, one would need to parse the tile group header in order to know the value of MaxMttDepthY.]

**pps\_cb\_qp\_offset** and **pps\_cr\_qp\_offset** specify the offsets to the luma quantization parameter Qp′Y used for deriving Qp′Cb and Qp′Cr, respectively. The values of pps\_cb\_qp\_offset and pps\_cr\_qp\_offset shall be in the range of −12 to +12, inclusive. When ChromaArrayType is equal to 0, pps\_cb\_qp\_offset and pps\_cr\_qp\_offset are not used in the decoding process and decoders shall ignore their value.

**pps\_tile\_group\_chroma\_qp\_offsets\_present\_flag** equal to 1 indicates that the tile\_group\_cb\_qp\_offset and tile\_group\_cr\_qp\_offset syntax elements are present in the associated tile group headers. pps\_tile\_group\_chroma\_qp\_offsets\_present\_flag equal to 0 indicates that these syntax elements are not present in the associated tile group headers. When ChromaArrayType is equal to 0, pps\_tile\_group\_chroma\_qp\_offsets\_present\_flag shall be equal to 0.

**weighted\_pred\_flag** equal to 0 specifies that weighted prediction is not applied to P tile groups. weighted\_pred\_flag equal to 1 specifies that weighted prediction is applied to P tile groups.

**weighted\_bipred\_flag** equal to 0 specifies that the default weighted prediction is applied to B tile groups. weighted\_bipred\_flag equal to 1 specifies that weighted prediction is applied to B tile groups.

**deblocking\_filter\_control\_present\_flag** equal to 1 specifies the presence of deblocking filter control syntax elements in the PPS. deblocking\_filter\_control\_present\_flag equal to 0 specifies the absence of deblocking filter control syntax elements in the PPS.

**deblocking\_filter\_override\_enabled\_flag** equal to 1 specifies the presence of deblocking\_filter\_override\_flag in the tile group headers for pictures referring to the PPS. deblocking\_filter\_override\_enabled\_flag equal to 0 specifies the absence of deblocking\_filter\_override\_flag in the tile group headers for pictures referring to the PPS. When not present, the value of deblocking\_filter\_override\_enabled\_flag is inferred to be equal to 0.

**pps\_deblocking\_filter\_disabled\_flag** equal to 1 specifies that the operation of deblocking filter is not applied for tile groups referring to the PPS in which tile\_group\_deblocking\_filter\_disabled\_flag is not present. pps\_deblocking\_filter\_disabled\_flag equal to 0 specifies that the operation of the deblocking filter is applied for tile groups referring to the PPS in which tile\_group\_deblocking\_filter\_disabled\_flag is not present. When not present, the value of pps\_deblocking\_filter\_disabled\_flag is inferred to be equal to 0.

**pps\_beta\_offset\_div2** and **pps\_tc\_offset\_div2** specify the default deblocking parameter offsets for β and tC (divided by 2) that are applied for tile groups referring to the PPS, unless the default deblocking parameter offsets are overridden by the deblocking parameter offsets present in the tile group headers of the tile groups referring to the PPS. The values of pps\_beta\_offset\_div2 and pps\_tc\_offset\_div2 shall both be in the range of −6 to 6, inclusive. When not present, the value of pps\_beta\_offset\_div2 and pps\_tc\_offset\_div2 are inferred to be equal to 0.

**pps\_extension\_flag** equal to 0 specifies that no pps\_extension\_data\_flag syntax elements are present in the PPS RBSP syntax structure. pps\_extension\_flag equal to 1 specifies that there are pps\_extension\_data\_flag syntax elements present in the PPS RBSP syntax structure.

**pps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all pps\_extension\_data\_flag syntax elements.

#### Adaptation parameter set semanitcs

**adaptation\_parameter\_set\_id** provides an identifier for the APS for reference by other syntax elements.

NOTE – APSs can be shared across pictures and can be different in different tile groups within a picture.

**aps\_extension\_flag** equal to 0 specifies that no aps\_extension\_data\_flag syntax elements are present in the APS RBSP syntax structure. aps\_extension\_flag equal to 1 specifies that there are aps\_extension\_data\_flag syntax elements present in the APS RBSP syntax structure.

**aps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all aps\_extension\_data\_flag syntax elements.

#### Access unit delimiter RBSP semantics

The access unit delimiter may be used to indicate the type of tile groups present in the coded pictures in the access unit containing the access unit delimiter NAL unit and to simplify the detection of the boundary between access units. There is no normative decoding process associated with the access unit delimiter.

**pic\_type** indicates that the tile\_group\_type values for all tile groups of the coded pictures in the access unit containing the access unit delimiter NAL unit are members of the set listed in Table 7‑2 for the given value of pic\_type. The value of pic\_type shall be equal to 0, 1 or 2 in bitstreams conforming to this version of this Specification. Other values of pic\_type are reserved for future use by ITU‑T | ISO/IEC. Decoders conforming to this version of this Specification shall ignore reserved values of pic\_type.

Table 7‑2 – Interpretation of pic\_type

|  |  |
| --- | --- |
| **pic\_type** | **Tiles\_group\_type values that may be present in the coded picture** |
| 0 | I |
| 1 | P, I |
| 2 | B, P, I |

#### End of sequence RBSP semantics

When present, 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.

#### Tile group layer RBSP semantics

The tile group layer RBSP consists of a tile group header and tile group data.

#### RBSP tile group 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 ) ) (7‑34)

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 tile group 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.

### Profile, tier, and level semantics

#### General profile, tier, and level semantics

When the profile\_tier\_level( ) structure is included in an SPS, the BitstreamInScope is the CVS for which the SPS is the active SPS.

**general\_tier\_flag** specifies the tier context for the interpretation of general\_level\_idc as specified in Annex A.

**general\_profile\_idc** indicates a profile to which BitstreamInScope conforms as specified in Annex A. Bitstreams shall not contain values of general\_profile\_idc other than those specified in Annex A. Other values of general\_profile\_idc are reserved for future use by ITU-T | ISO/IEC.

**general\_level\_idc** indicates a level to which BitstreamInScope conforms as specified in Annex A. Bitstreams shall not contain values of general\_level\_idc other than those specified in Annex A. Other values of general\_level\_idc are reserved for future use by ITU-T | ISO/IEC.

NOTE 1 – A greater value of general\_level\_idc indicates a higher level.

NOTE 2 – When BitstreamInScope conforms to multiple profiles, general\_profile\_idc should indicate the profile that provides the preferred decoded result or the preferred bitstream identification, as determined by the encoder (in a manner not specified in this Specification).

**sub\_layer\_level\_present\_flag**[ i ] equal to 1 specifies that level information is present in the profile\_tier\_level( ) syntax structure for the sub-layer representation with TemporalId equal to i. sub\_layer\_level\_present\_flag[ i ] equal to 0 specifies that level information is not present in the profile\_tier\_level( ) syntax structure for the sub-layer representation with TemporalId equal to i.

**ptl\_alignment\_zero\_bits** shall be equal to 0.

The semantics of the syntax element **sub\_layer\_level\_idc**[ i ] is, apart from the specification of the inference of not present values, the same as the syntax element general\_level\_idc, but apply to the sub-layer representation with TemporalId equal to i.

#### General constraint information semantics

**general\_progressive\_source\_flag** and **general\_interlaced\_source\_flag** are interpreted as follows:

– If general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 0, the source scan type of the pictures in BitstreamInScope should be interpreted as progressive only.

– Otherwise, if general\_progressive\_source\_flag is equal to 0 and general\_interlaced\_source\_flag is equal to 1, the source scan type of the pictures in BitstreamInScope should be interpreted as interlaced only.

– Otherwise, if general\_progressive\_source\_flag is equal to 0 and general\_interlaced\_source\_flag is equal to 0, the source scan type of the pictures in BitstreamInScope should be interpreted as unknown or unspecified.

– Otherwise (general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 1), the source scan type of each picture in BitstreamInScope should be interpreted as unknown or unspecified.

[Ed. If picture timing SEI message is specified for VVC, general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 1 can be specified to indicate the source scan type of each picture at the picture level using the syntax element source\_scan\_type in a picture timing SEI message.]

NOTE 1 – Decoders may ignore the values of general\_progressive\_source\_flag and general\_interlaced\_source\_flag. Moreover, the actual source scan type of the pictures is outside the scope of this Specification and the method by which the encoder selects the values of general\_progressive\_source\_flag and general\_interlaced\_source\_flag is unspecified.

**general\_non\_packed\_constraint\_flag** equal to 1 indicates that the cropped output pictures of the decoded bitstream are suitable to be displayed. general\_non\_packed\_constraint\_flag equal to 0 indicates that the cropped output pictures of the decoded bitstream may require further processing to be suitable for displaying. [Ed. Consider a better naming of this constraint flag, e.g., display\_suitability\_flag or something like that.]

NOTE 2 – Examples for content that requires additional processing before displaying are bitstreams containing one or more frame packing arrangement SEI messages, equirectangular projection SEI messages, or cubemap projection SEI messages. [Ed. This NOTE should be updated at a later phase of VVC standardization based on which SEI messages are adopted into the VVC design.]

**general\_frame\_only\_constraint\_flag** equal to 1 specifies that BitstreamInScope conveys pictures that represent frames. general\_frame\_only\_constraint\_flag equal to 0 specifies that BitstreamInScope conveys pictures that may or may not represent frames.

NOTE 3 – Decoders may ignore the value of general\_frame\_only\_constraint\_flag, as there are no decoding process requirements associated with it.

**intra\_only\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that tile\_group\_type shall be equal to I. intra\_only\_constraint\_flag equal to 0 does not impose a constraint.

**max\_bitdepth\_constraint\_idc** specifies that it is a requirement of bitstream conformance that bit\_depth\_luma\_minus8 and bit\_depth\_chroma\_minus8 shall be in the range of 0 to max\_bitdepth\_constraint\_idc, inclusive.

**max\_chroma\_format\_constraint\_idc** specifies that it is a requirement of bitstream conformance that chroma\_format\_idc shall be in the range of 0 to max\_chroma\_format\_constraint\_idc, inclusive.

**frame\_only\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that the CVS conveys pictures that represent frames. frame\_only\_constraint\_flag equal to 0 does not impose a constraint.

**no\_qtbtt\_dual\_tree\_intra constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that qtbtt\_dual\_tree\_intra\_flag shall be equal to 0. no\_qtbtt\_dual\_tree\_intra constraint\_flag equal to 0 does not impose a constraint.

**no\_sao\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_sao\_enabled\_flag shall be equal to 0. no\_sao\_constraint\_flag equal to 0 does not impose a constraint.

**no\_alf\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_alf\_enabled\_flag shall be equal to 0. no\_alf\_constraint\_flag equal to 0 does not impose a constraint.

**no\_pcm\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_pcm\_enabled\_flag shall be equal to 0. no\_pcm\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ref\_wraparound\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_ref\_wraparound\_enabled\_flag shall be equal to 0. no\_ref\_wraparound\_constraint\_flag equal to 0 does not impose a constraint.

**no\_temporal\_mvp\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_temporal\_mvp\_enabled\_flag shall be equal to 0. no\_temporal\_mvp\_constraint\_flag equal to 0 does not impose a constraint.

**no\_sbtmvp\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_sbtmvp\_enabled\_flag shall be equal to 0. no\_sbtmvp\_constraint\_flag equal to 0 does not impose a constraint.

**no\_amvr\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_amvr\_enabled\_flag shall be equal to 0. no\_amvr\_constraint\_flag equal to 0 does not impose a constraint.

**no\_bdof\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_bdof\_enabled\_flag shall be equal to 0. no\_bdof\_constraint\_flag equal to 0 does not impose a constraint.

**no\_cclm\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_cclm\_enabled\_flag shall be equal to 0. no\_cclm\_constraint\_flag equal to 0 does not impose a constraint.

**no\_mts\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_mts\_enabled\_flag shall be equal to 0. no\_mts\_constraint\_flag equal to 0 does not impose a constraint.

**no\_affine\_motion\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_affine\_enabled\_flag  shall be equal to 0. no\_affine\_motion\_constraint\_flag equal to 0 does not impose a constraint.

**no\_gbi\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_gbi\_enabled\_flag shall be equal to 0. no\_gbi\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ciip\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_ciip\_enabled\_flag shall be equal to 0. no\_cipp\_constraint\_flag equal to 0 does not impose a constraint.

**no\_triangle\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_triangle\_enabled\_flag shall be equal to 0. no\_triangle\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ladf\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_ladf\_enabled\_flag shall be equal to 0. no\_ladf\_constraint\_flag equal to 0 does not impose a constraint.

**no\_cpr\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sps\_cpr\_enabled\_flag shall be equal to 0. no\_cpr\_constraint\_flag equal to 0 does not impose a constraint.

**no\_qp\_delta\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that cu\_qp\_delta\_enabled\_flag shall be equal to 0. no\_qp\_delta\_constraint\_flag equal to 0 does not impose a constraint.

**no\_dep\_quant\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that dep\_quant\_enabled\_flag shall be equal to 0. no\_dep\_quant\_constraint\_flag equal to 0 does not impose a constraint.

**no\_sign\_data\_hiding\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sign\_data\_hiding\_enabled\_flag shall be equal to 0. no\_sign\_data\_hiding\_constraint\_flag equal to 0 does not impose a constraint.

**gci\_alignment\_zero\_bits** shall be equal to 0.

### Tile group header semantics

#### General tile group header semantics

When present, the value of each of the tile group header syntax elements tile\_group\_pic\_parameter\_set\_id, tile\_group\_pic\_order\_cnt\_lsb, and tile\_group\_temporal\_mvp\_enabled\_flag shall be the same in all tile group headers of a coded picture.

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

It is a requirement of bitstream conformance that the value of TemporalId of the current picture shall be greater than or equal to the value of TemporalId of the PPS that has pps\_pic\_parameter\_set\_id equal to tile\_group\_pic\_parameter\_set\_id.

**tile\_group\_address** specifies the tile address of the first tile in the tile group. When not present, the value of tile\_group\_address is inferred to be equal to 0.

If rect\_tile\_group\_flag is equal to 0, the following applies:

* The tile address is the tile ID as specified by Equation 6‑7.
* The length of tile\_group\_address is Ceil( Log2 ( NumTilesInPic ) ) bits.
* The value of tile\_group\_address shall be in the range of 0 to NumTilesInPic − 1, inclusive.

Otherwise (rect\_tile\_group\_flag is equal to 1), the following applies:

* The tile address is the tile group ID of the tile group.
* The length of tile\_group\_address is signalled\_tile\_group\_id\_length\_minus1 + 1 bits.
* If signalled\_tile\_group\_id\_flag is equal to 0, the value of tile\_group\_address shall be in the range of 0 to num\_tile\_groups\_in\_pic\_minus1, inclusive. Otherwise, the value of tile\_group\_address shall be in the range of 0 to 2( signalled\_tile\_group\_id\_length\_minus1 + 1 ) − 1, inclusive.

It is a requirement of bitstream conformance that the following constraints apply:

* The value of tile\_group\_address shall not be equal to the value of tile\_group\_address of any other coded tile group NAL unit of the same coded picture.
* The tile groups of a picture shall be in increasing order of their tile\_group\_address values.
* The shapes of the tile groups of a picture shall be such that each tile, when decoded, shall have its entire left boundary and entire top boundary consisting of a picture boundary or consisting of boundaries of previously decoded tile(s).

**num\_tiles\_in\_tile\_group\_minus1**, when present, specifies the number of tiles in the tile group minus 1. The value of num\_tiles\_in\_tile\_group\_minus1 shall be in the range of 0 to NumTilesInPic − 1, inclusive. When not present, the value of num\_tiles\_in\_tile\_group\_minus1 is inferred to be equal to 0.

The variable NumTilesInCurrTileGroup, which specifies the number of tiles in the current tile group, and TgTileIdx[ i ], which specifies the tile index of the i-th tile in the current tile group, are derived as follows:

if( rect\_tile\_group\_flag ) {  
 tileGroupIdx = 0  
 while( tile\_group\_address != rect\_tile\_group\_id[ tileGroupIdx ] )  
 tileGroupIdx++  
 NumTilesInCurrTileGroup = NumTilesInTileGroup[ tileGroupIdx ]  
 tileIdx = top\_left\_tile\_idx[ tileGroupIdx ]  
 for( j = 0, tIdx = 0; j < NumTileRowsInTileGroupMinus1[ tileGroupIdx ] + 1; (7‑35)  
 j++, tileIdx += num\_tile\_columns\_minus1 + 1 )  
 for( i = 0, currTileIdx = tileIdx; i < NumTileColumnsInTileGroupMinus1[ tileGroupIdx ] + 1;  
 i++, currTileIdx++, tIdx++ )  
 TgTileIdx[ tIdx ] = currTileIdx  
} else {  
 NumTilesInCurrTileGroup = num\_tiles\_in\_tile\_group\_minus1 + 1  
 TgTileIdx[ 0 ] = tile\_group\_address  
 for( i = 1; i < NumTilesInCurrTileGroup; i++ )  
 TgTileIdx[ i ] = TgTileIdx[ i − 1 ] + 1  
}

**tile\_group\_type** specifies the coding type of the tile group according to Table 7‑3.

Table 7‑3 – Name association to tile\_group\_type

|  |  |
| --- | --- |
| tile\_group\_type | Name of tile\_group\_type |
| 0 | B (B tile group) |
| 1 | P (P tile group) |
| 2 | I (I tile group) |

When nal\_unit\_type is a value of nal\_unit\_type in the range of IDR\_W\_RADL to RSV\_IRAP\_VCL13, inclusive, i.e., the picture is an IRAP picture, tile\_group\_type shall be equal to 2.

**tile\_group\_pic\_order\_cnt\_lsb** specifies the picture order count modulo MaxPicOrderCntLsb for the current picture. The length of the tile\_group\_pic\_order\_cnt\_lsb syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits. The value of the tile\_group\_pic\_order\_cnt\_lsb shall be in the range of 0 to MaxPicOrderCntLsb − 1, inclusive.

**ref\_pic\_list\_sps\_flag**[ i ] equal to 1 specifies that reference picture list i of the current tile group is derived based on one of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures with listIdx equal to i in the active SPS. ref\_pic\_list\_sps\_flag[ i ] equal to 0 specifies that reference picture list i of the current tile group is derived based on the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure with listIdx equal to i that is directly included in the tile group headers of the current picture. When num\_ref\_pic\_lists\_in\_sps[ i ] is equal to 0, the value of ref\_pic\_list\_sps\_flag[ i ] is inferred to be equal to 0. When rpl1\_idx\_present\_flag is equal to 0, the value of ref\_pic\_list\_sps\_flag[ 1 ] is inferred to be equal to ref\_pic\_list\_sps\_flag[ 0 ].

**ref\_pic\_list\_idx**[ i ] specifies the index, into the list of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures with listIdx equal to i included in the active SPS, of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure with listIdx equal to i that is used for derivation of reference picture list i of the current picture. The syntax element ref\_pic\_list\_idx[ i ] is represented by Ceil( Log2( num\_ref\_pic\_lists\_in\_sps[ i ] ) ) bits. When not present, the value of ref\_pic\_list\_idx[ i ] is inferred to be equal to 0. The value of ref\_pic\_list\_idx[ i ] shall be in the range of 0 to num\_ref\_pic\_lists\_in\_sps[ i ] − 1, inclusive. When ref\_pic\_list\_sps\_flag[ i ] is equal to 1 and num\_ref\_pic\_lists\_in\_sps[ i ] is equal to 1, the value of ref\_pic\_list\_idx[ i ] is inferred to be equal to 0. When ref\_pic\_list\_sps\_flag[ i ] is equal to 1 and rpl1\_idx\_present\_flag is equal to 0, the value of ref\_pic\_list\_idx[ 1 ] is inferred to be equal to ref\_pic\_list\_idx[ 0 ].

The variable RplsIdx[ i ] is derived as follows:

RplsIdx[ i ] = ref\_pic\_list\_sps\_flag[ i ] ? ref\_pic\_list\_idx[ i ] : num\_ref\_pic\_lists\_in\_sps[ i ] (7‑36)

**delta\_poc\_msb\_present\_flag**[ i ][ j ] equal to 1 specifies that delta\_poc\_msb\_cycle\_lt[ i ][ j ] is present. delta\_poc\_msb\_present\_flag[ i ][ j ]equal to 0 specifies that delta\_poc\_msb\_cycle\_lt[ i ][ j ] is not present.

Let prevTid0Pic be the previous picture in decoding order that has TemporalId equal to 0 and is not a RASL or RADL picture. [Ed. (YK): Check this sentence after the texts of RASL and RADL pictures are more mature.] Let setOfPrevPocVals be a set consisting of the following:

– the PicOrderCntVal of prevTid0Pic,

– the PicOrderCntVal of each picture referred to by entries in RefPicList[ 0 ] and entries in RefPicList[ 1 ] of prevTid0Pic,

– the PicOrderCntVal of each picture that follows prevTid0Pic in decoding order and precedes the current picture in decoding order.

When there is more than one value in setOfPrevPocVals for which the value modulo MaxPicOrderCntLsb is equal to poc\_lsb\_lt[ i ][ RplsIdx[ i ] ][ j ], the value of delta\_poc\_msb\_present\_flag[ i ][ j ] shall be equal to 1.

**delta\_poc\_msb\_cycle\_lt**[ i ][ j ] specifies the value of the variable FullPocLt[ i ][ j ] as follows:

if( j = = 0 )  
 deltaMsbCycle[ i ][ j ] = delta\_poc\_msb\_cycle\_lt[ i ][ j ]  
else (7‑37)  
 deltaMsbCycle[ i ][ j ] = delta\_poc\_msb\_cycle\_lt[ i ][ j ] + deltaMsbCycle[ i ][ j − 1 ]  
FullPocLt[ i ][ RplsIdx[ i ] ][ j ] = PicOrderCntVal − deltaMsbCycle[ i ][ j ] \* MaxPicOrderCntLsb −  
 ( PicOrderCntVal & ( MaxPicOrderCntLsb − 1 ) ) + poc\_lsb\_lt[ i ][ RplsIdx[ i ] ][ j ]

The value of delta\_poc\_msb\_cycle\_lt[ i ][ j ] shall be in the range of 0 to 2(32 − log2\_max\_pic\_order\_cnt\_lsb\_minus4 − 4 ), inclusive. When not present, the value of delta\_poc\_msb\_cycle\_lt[ i ][ j ] is inferred to be equal to 0.

**num\_ref\_idx\_active\_override\_flag** equal to 1 specifies that the syntax element num\_ref\_idx\_active\_minus1[ 0 ] is present for P and B tile groups and that the syntax element num\_ref\_idx\_active\_minus1[ 1 ] is present for B tile groups. num\_ref\_idx\_active\_override\_flag equal to 0 specifies that the syntax elements num\_ref\_idx\_active\_minus1[ 0 ] and num\_ref\_idx\_active\_minus1[ 1 ] are not present.

**num\_ref\_idx\_active\_minus1**[ i ] is used for the derivation of the variable NumRefIdxActive[ i ] as specified by Equation 7‑38. The value of num\_ref\_idx\_active\_minus1[ i ] shall be in the range of 0 to 14, inclusive.

For i equal to 0 or 1, when the current tile group is a B tile group, num\_ref\_idx\_active\_override\_flag is equal to 1, and num\_ref\_idx\_active\_minus1[ i ] is not present, num\_ref\_idx\_active\_minus1[ i ] is inferred to be equal to 0.

When the current tile group is a P tile group, num\_ref\_idx\_active\_override\_flag is equal to 1, and num\_ref\_idx\_active\_minus1[ 0 ] is not present, num\_ref\_idx\_active\_minus1[ 0 ] is inferred to be equal to 0.

The variable NumRefIdxActive[ i ] is derived as follows:

for( i = 0; i < 2; i++ ) {  
 if( tile\_group\_type = = B | | ( tile\_group\_type = = P && i = = 0 ) ) {  
 if( num\_ref\_idx\_active\_override\_flag = = 1 )  
 NumRefIdxActive[ i ] = num\_ref\_idx\_active\_minus1[ i ] + 1 (7‑38)  
 else {  
 if( num\_ref\_entries[ i ][ RplsIdx[ i ] ] >= num\_ref\_idx\_default\_active\_minus1[ i ] + 1 )  
 NumRefIdxActive[ i ] = num\_ref\_idx\_default\_active\_minus1[ i ] + 1  
 else  
 NumRefIdxActive[ i ] = num\_ref\_entries[ i ][ RplsIdx[ i ] ]  
 }  
 } else // tile\_group\_type = = I | | ( tile\_group\_type = = P && i = = 1 )  
 NumRefIdxActive[ i ] = 0  
}

The value of NumRefIdxActive[ i ] − 1 specifies the maximum reference index for reference picture list i that may be used to decode the tile group. When the value of NumRefIdxActive[ i ] is equal to 0, no reference index for reference picture list i may be used to decode the tile group.

The variable CurrPicIsOnlyRef, specifying that the current decoded picture is the only reference picture for the current tile group, is derived as follows:

CurrPicIsOnlyRef = sps\_cpr\_enabled\_flag && ( tile\_group\_type  = =  P ) && (7‑39)  
 ( num\_ref\_idx\_active\_minus1[ 0 ]  = =  0 )

**partition\_constraints\_override\_flag** equal to 1 specifies that partition constraint parameters are present in the tile group header. partition\_constraints\_override\_flag equal to 0 specifies that partition constraint parameters are not present in the tile group header. When not present, the value of partition\_constraints\_override\_flag is inferred to be equal to 0.

**tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma** specifies the difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum coding block size in luma samples for luma CUs in the current tile group. The value of tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma is inferred as follows:

* If tile\_group\_type equal to 2 (I), the value of tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma is inferred to be equal to sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_luma
* Otherwise (tile\_group\_type equal to 0 (B) or 1 (P)), the value of tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma is inferred to be equal to sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_tile\_group.

**tile\_group\_max\_mtt\_hierarchy\_depth\_luma** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in the current tile group. The value of tile\_group\_max\_mtt\_hierarchy\_depth\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of tile\_group\_max\_mtt\_hierarchy\_depth\_luma is inferred as follows:

* If tile\_group\_type equal to 2 (I), the value of tile\_group\_max\_mtt\_hierarchy\_depth\_luma is inferred to be equal to sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_group\_luma
* Otherwise (tile\_group\_type equal to 0 (B) or 1 (P)), the value of tile\_group\_max\_mtt\_hierarchy\_depth\_luma is inferred to be equal to sps\_max\_mtt\_hierarchy\_depth\_inter\_tile\_group.

**tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in the current tile group. The value of tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeY, inclusive. When not present, the value of tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma is inferred as follows:

* If tile\_group\_type equal to 2 (I), the value of tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_luma
* Otherwise (tile\_group\_type equal to 0 (B) or 1 (P)), the value of tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_tile\_group.

**tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in in the current tile group. The value of tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeY, inclusive. When not present, the value of tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma is inferred as follows:

* If tile\_group\_type equal to 2 (I), the value of tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_luma
* Otherwise (tile\_group\_type equal to 0 (B) or 1 (P)), the value of tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_tile\_group.

**tile\_group\_log2\_diff\_min\_qt\_min\_cb\_chroma** specifies the difference between the base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA and the base 2 logarithm of the minimum coding block size in luma samples for chroma CUs with treeType equal to DUAL\_TREE\_CHROMA in the current tile group. The value of tile\_group\_log2\_diff\_min\_qt\_min\_cb\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of tile\_group\_log2\_diff\_min\_qt\_min\_cb\_chroma is inferred to be equal to sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_tile\_group\_chroma.

**tile\_group\_max\_mtt\_hierarchy\_depth\_chroma** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf with treeType equal to DUAL\_TREE\_CHROMA in the current tile group. The value of tile\_group\_max\_mtt\_hierarchy\_depth\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the values of tile\_group\_max\_mtt\_hierarchy\_depth\_chroma is inferred to be equal to sps\_max\_mtt\_hierarchy\_depth\_intra\_tile\_groups\_chroma.

**tile\_group\_log2\_diff\_max\_bt\_min\_qt\_chroma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in the current tile group. The value of tile\_group\_log2\_diff\_max\_bt\_min\_qt\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeC, inclusive. When not present, the value of tile\_group\_log2\_diff\_max\_bt\_min\_qt\_chroma is inferred to be equal to sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_tile\_group\_chroma

**tile\_group\_log2\_diff\_max\_tt\_min\_qt\_chroma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in the current tile group. The value of tile\_group\_log2\_diff\_max\_tt\_min\_qt\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeC, inclusive. When not present, the value of tile\_group\_log2\_diff\_max\_tt\_min\_qt\_chroma is inferred to be equal to sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_tile\_group\_chroma

The variables MinQtLog2SizeY, MinQtLog2SizeC, MinQtSizeY, MinQtSizeC, MaxBtSizeY, MaxBtSizeC, MinBtSizeY, MaxTtSizeY, MaxTtSizeC, MinTtSizeY, MaxMttDepthY and MaxMttDepthC are derived as follows:

MinQtLog2SizeY = MinCbLog2SizeY + tile\_group\_log2\_diff\_min\_qt\_min\_cb\_luma (7‑40)

MinQtLog2SizeC = MinCbLog2SizeY + tile\_group\_log2\_diff\_min\_qt\_min\_cb\_chroma (7‑41)

MinQtSizeY = 1  <<  MinQtLog2SizeY (7‑42)

MinQtSizeC = 1  <<  MinQtLog2SizeC (7‑43)

MaxBtSizeY = 1  <<  ( MinQtLog2SizeY + tile\_group\_log2\_diff\_max\_bt\_min\_qt\_luma ) (7‑44)

MaxBtSizeC = 1  <<  ( MinQtLog2SizeC + tile\_group\_log2\_diff\_max\_bt\_min\_qt\_chroma ) (7‑45)

MinBtSizeY = 1  <<  MinCbLog2SizeY (7‑46)

MaxTtSizeY = 1  <<  ( MinQtLog2SizeY + tile\_group\_log2\_diff\_max\_tt\_min\_qt\_luma ) (7‑47)

MaxTtSizeC = 1  <<  ( MinQtLog2SizeC + tile\_group\_log2\_diff\_max\_tt\_min\_qt\_chroma ) (7‑48)

MinTtSizeY = 1  <<  MinCbLog2SizeY (7‑49)

MaxMttDepthY = tile\_group\_max\_mtt\_hierarchy\_depth\_luma (7‑50)

MaxMttDepthC = tile\_group\_max\_mtt\_hierarchy\_depth\_chroma (7‑51)

**tile\_group\_temporal\_mvp\_enabled\_flag** specifies whether temporal motion vector predictors can be used for inter prediction. If tile\_group\_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 (tile\_group\_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 tile\_group\_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.

**cabac\_init\_flag** specifies the method for determining the initialization table used in the initialization process for context variables. When cabac\_init\_flag is not present, it is inferred to be equal to 0.

**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 tile group subtracted from 6. The maximum number of merging MVP candidates, MaxNumMergeCand is derived as follows:

MaxNumMergeCand = 6 − six\_minus\_max\_num\_merge\_cand (7‑52)

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

**five\_minus\_max\_num\_subblock\_merge\_cand** specifies the maximum number of subblock-based merging motion vector prediction (MVP) candidates supported in the tile group subtracted from 5. When five\_minus\_max\_num\_subblock\_merge\_cand is not present, it is inferred to be equal to 5 − sps\_sbtmvp\_enabled\_flag. The maximum number of subblock-based merging MVP candidates, MaxNumSubblockMergeCand is derived as follows:

MaxNumSubblockMergeCand = 5 − five\_minus\_max\_num\_subblock\_merge\_cand (7‑53)

The value of MaxNumSubblockMergeCand shall be in the range of 0 to 5, inclusive.

**tile\_group\_fpel\_mmvd\_enabled\_flag** equal to 1 specifies that merge mode with motion vector difference uses integer sample precision in the current tile group. tile\_group\_fpel\_mmvd\_enabled\_flag equal to 0 specifies that merge mode with motion vector difference can use fractional sample precision in the current tile group. When not present, the value of tile\_group\_fpel\_mmvd\_enabled\_flag is inferred to be 0.

**tile\_group\_qp\_delta** specifies the initial value of QpY to be used for the coding blocks in the tile group until modified by the value of CuQpDeltaVal in the coding unit layer. The initial value of the QpY quantization parameter for the tile group, TileGroupQpY, is derived as follows:

TileGroupQpY = 26 + init\_qp\_minus26 + tile\_group\_qp\_delta (7‑54)

The value of TileGroupQpY shall be in the range of −QpBdOffsetY to +63, inclusive.

**tile\_group\_cb\_qp\_offset** specifies a difference to be added to the value of pps\_cb\_qp\_offset when determining the value of the Qp′Cb quantization parameter. The value of tile\_group\_cb\_qp\_offset shall be in the range of −12 to +12, inclusive. When tile\_group\_cb\_qp\_offset is not present, it is inferred to be equal to 0. The value of pps\_cb\_qp\_offset + tile\_group\_cb\_qp\_offset shall be in the range of −12 to +12, inclusive.

**tile\_group\_cr\_qp\_offset** specifies a difference to be added to the value of pps\_cr\_qp\_offset when determining the value of the Qp′Cr quantization parameter. The value of tile\_group\_cr\_qp\_offset shall be in the range of −12 to +12, inclusive. When tile\_group\_cr\_qp\_offset is not present, it is inferred to be equal to 0. The value of pps\_cr\_qp\_offset + tile\_group\_cr\_qp\_offset shall be in the range of −12 to +12, inclusive.

**tile\_group\_sao\_luma\_flag** equal to 1 specifies that SAO is enabled for the luma component in the current tile group; tile\_group\_sao\_luma\_flag equal to 0 specifies that SAO is disabled for the luma component in the current tile group. When tile\_group\_sao\_luma\_flag is not present, it is inferred to be equal to 0.

**tile\_group\_sao\_chroma\_flag** equal to 1 specifies that SAO is enabled for the chroma component in the current tile group; tile\_group\_sao\_chroma\_flag equal to 0 specifies that SAO is disabled for the chroma component in the current tile group. When tile\_group\_sao\_chroma\_flag is not present, it is inferred to be equal to 0.

**tile\_group\_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 tile group. tile\_group\_alf\_enabled\_flag equal to 0 specifies that adaptive loop filter is disabled for all colour components in a tile group.

**tile\_group\_aps\_id** specifies the adaptation\_parameter\_set\_id of the APS that the tile group refers to. The TemporalId of the APS NAL unit having adaptation\_parameter\_set\_id equal to tile\_group\_aps\_id shall be less than or equal to the TemporalId of the coded tile group NAL unit.

When multiple APSs with the same value of adaptation\_parameter\_set\_id are referred to by two or more tile groups of the same picture, the multiple APSs with the same value of adaptation\_parameter\_set\_id shall have the same content.

**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.

**deblocking\_filter\_override\_flag** equal to 1 specifies that deblocking parameters are present in the tile group header. deblocking\_filter\_override\_flag equal to 0 specifies that deblocking parameters are not present in the tile group header. When not present, the value of deblocking\_filter\_override\_flag is inferred to be equal to 0.

**tile\_group\_deblocking\_filter\_disabled\_flag** equal to 1 specifies that the operation of the deblocking filter is not applied for the current tile group. tile\_group\_deblocking\_filter\_disabled\_flag equal to 0 specifies that the operation of the deblocking filter is applied for the current tile group. When tile\_group\_deblocking\_filter\_disabled\_flag is not present, it is inferred to be equal to pps\_deblocking\_filter\_disabled\_flag.

**tile\_group\_beta\_offset\_div2** and **tile\_group\_tc\_offset\_div2** specify the deblocking parameter offsets for β and tC (divided by 2) for the current tile group. The values of tile\_group\_beta\_offset\_div2 and tile\_group\_tc\_offset\_div2 shall both be in the range of −6 to 6, inclusive. When not present, the values of tile\_group\_beta\_offset\_div2 and tile\_group\_tc\_offset\_div2 are inferred to be equal to pps\_beta\_offset\_div2 and pps\_tc\_offset\_div2, respectively.

**tile\_group\_lmcs\_model\_present\_flag** equal to 1 specifies that lmcs\_data() is present in the tile group header. tile\_group\_lmcs\_model\_present\_flag equal to 0 specifies that lmcs\_data() is not present in the tile group header. When tile\_group\_lmcs\_model\_present\_flag is not present, it is inferred to be equal to 0.

**tile\_group\_lmcs\_enabled\_flag** equal to 1 specifies that luma mappin with chroma scaling is enabled for the current tile group. tile\_group\_lmcs\_enabled\_flag equal to 0 specifies that luma mapping with chroma scaling is not enabled for the current tile group. When tile\_group\_lmcs\_enabled\_flag is not present, it is inferred to be equal to 0.

**tile\_group\_chroma\_residual\_scale\_flag** equal to 1 specifies that chroma residual scaling is enabled for the current tile group. tile\_group\_ chroma\_residual\_scale\_flag equal to 0 specifies that chroma residual scaling is not enabled for the current tile group. When tile\_group\_ chroma\_residual\_scale\_flag is not present, it is inferred to be equal to 0.

**offset\_len\_minus1** plus 1 specifies the length, in bits, of the entry\_point\_offset\_minus1[ i ] syntax elements. The value of offset\_len\_minus1 shall be in the range of 0 to 31, inclusive.

**entry\_point\_offset\_minus1**[ i ] plus 1 specifies the i-th entry point offset in bytes, and is represented by offset\_len\_minus1 plus 1 bits. The tile group data that follow the tile group header consists of NumTilesInCurrTileGroup subsets, with subset index values ranging from 0 to NumTilesInCurrTileGroup − 1, inclusive. The first byte of the tile group data is considered byte 0. When present, emulation prevention bytes that appear in the tile group data portion of the coded tile group NAL unit are counted as part of the tile group data for purposes of subset identification. Subset 0 consists of bytes 0 to entry\_point\_offset\_minus1[ 0 ], inclusive, of the coded tile group data, subset k, with k in the range of 1 to NumTilesInCurrTileGroup − 2, inclusive, consists of bytes firstByte[ k ] to lastByte[ k ], inclusive, of the coded tile group data with firstByte[ k ] and lastByte[ k ] defined as:

(7‑55)

lastByte[ k ] = firstByte[ k ] + entry\_point\_offset\_minus1[ k ] (7‑56)

The last subset (with subset index equal to NumTilesInCurrTileGroup − 1) consists of the remaining bytes of the coded tile group data.

Each subset shall consist of all coded bits of all CTUs in the tile group that are within the same tile.

#### Weighted prediction parameters semantics

**luma\_log2\_weight\_denom** is the base 2 logarithm of the denominator for all luma weighting factors. The value of luma\_log2\_weight\_denom shall be in the range of 0 to 7, inclusive.

**delta\_chroma\_log2\_weight\_denom** is the difference of the base 2 logarithm of the denominator for all chroma weighting factors. When delta\_chroma\_log2\_weight\_denom is not present, it is inferred to be equal to 0.

The variable ChromaLog2WeightDenom is derived to be equal to luma\_log2\_weight\_denom + delta\_chroma\_log2\_weight\_denom and the value shall be in the range of 0 to 7, inclusive.

**luma\_weight\_l0\_flag**[ i ] equal to 1 specifies that weighting factors for the luma component of list 0 prediction using RefPicList[ 0 ][ i ] are present. luma\_weight\_l0\_flag[ i ] equal to 0 specifies that these weighting factors are not present.

**chroma\_weight\_l0\_flag**[ i ] equal to 1 specifies that weighting factors for the chroma prediction values of list 0 prediction using RefPicList[ 0 ][ i ] are present. chroma\_weight\_l0\_flag[ i ] equal to 0 specifies that these weighting factors are not present. When chroma\_weight\_l0\_flag[ i ] is not present, it is inferred to be equal to 0.

**delta\_luma\_weight\_l0**[ i ] is the difference of the weighting factor applied to the luma prediction value for list 0 prediction using RefPicList[ 0 ][ i ].

The variable LumaWeightL0[ i ] is derived to be equal to ( 1  <<  luma\_log2\_weight\_denom ) + delta\_luma\_weight\_l0[ i ]. When luma\_weight\_l0\_flag[ i ] is equal to 1, the value of delta\_luma\_weight\_l0[ i ] shall be in the range of −128 to 127, inclusive. When luma\_weight\_l0\_flag[ i ]is equal to 0, LumaWeightL0[ i ] is inferred to be equal to 2luma\_log2\_weight\_denom.

**luma\_offset\_l0**[ i ] is the additive offset applied to the luma prediction value for list 0 prediction using RefPicList[ 0 ][ i ]. The value of luma\_offset\_l0[ i ] shall be in the range of −WpOffsetHalfRangeY to WpOffsetHalfRangeY − 1, inclusive. When luma\_weight\_l0\_flag[ i ]is equal to 0, luma\_offset\_l0[ i ] is inferred to be equal to 0.

**delta\_chroma\_weight\_l0**[ i ][ j ] is the difference of the weighting factor applied to the chroma prediction values for list 0 prediction using RefPicList[ 0 ][ i ] with j equal to 0 for Cb and j equal to 1 for Cr.

The variable ChromaWeightL0[ i ][ j ] is derived to be equal to ( 1  <<  ChromaLog2WeightDenom ) + delta\_chroma\_weight\_l0[ i ][ j ]. When chroma\_weight\_l0\_flag[ i ] is equal to 1, the value of delta\_chroma\_weight\_l0[ i ][ j ] shall be in the range of −128 to 127, inclusive. When chroma\_weight\_l0\_flag[ i ] is equal to 0**,** ChromaWeightL0[ i ][ j ] is inferred to be equal to 2ChromaLog2WeightDenom.

**delta\_chroma\_offset\_l0**[ i ][ j ] is the difference of the additive offset applied to the chroma prediction values for list 0 prediction using RefPicList[ 0 ][ i ] with j equal to 0 for Cb and j equal to 1 for Cr.

The variable ChromaOffsetL0[ i ][ j ] is derived as follows:

ChromaOffsetL0[ i ][ j ] = Clip3( −WpOffsetHalfRangeC, WpOffsetHalfRangeC − 1,  
 ( WpOffsetHalfRangeC + delta\_chroma\_offset\_l0[ i ][ j ] − (7‑57)  
 ( ( WpOffsetHalfRangeC \* ChromaWeightL0[ i ][ j ] )  >>  ChromaLog2WeightDenom ) ) )

The value of delta\_chroma\_offset\_l0[ i ][ j ] shall be in the range of −4 \* WpOffsetHalfRangeC to 4 \* WpOffsetHalfRangeC − 1, inclusive. When chroma\_weight\_l0\_flag[ i ] is equal to 0**,** ChromaOffsetL0[ i ][ j ] is inferred to be equal to 0.

**luma\_weight\_l1\_flag**[ i ]**, chroma\_weight\_l1\_flag**[ i ]**, delta\_luma\_weight\_l1**[ i ], **luma\_offset\_l1**[ i ], **delta\_chroma\_weight\_l1**[ i ][ j ] and **delta\_chroma\_offset\_l1**[ i ][ j ] have the same semantics as luma\_weight\_l0\_flag[ i ], chroma\_weight\_l0\_flag[ i ], delta\_luma\_weight\_l0[ i ], luma\_offset\_l0[ i ], delta\_chroma\_weight\_l0[ i ][ j ] and delta\_chroma\_offset\_l0[ i ][ j ], respectively, with l0, L0, list 0 and List0 replaced by l1, L1, list 1 and List1, respectively.

The variable sumWeightL0Flags is derived to be equal to the sum of luma\_weight\_l0\_flag[ i ] + 2 \* chroma\_weight\_l0\_flag[ i ], for i = 0..NumRefIdxActive[ 0 ] − 1.

When tile\_group\_type is equal to B, the variable sumWeightL1Flags is derived to be equal to the sum of luma\_weight\_l1\_flag[ i ] + 2 \* chroma\_weight\_l1\_flag[ i ], for i = 0..NumRefIdxActive[ 1 ] − 1.

It is a requirement of bitstream conformance that, when tile\_group\_type is equal to P, sumWeightL0Flags shall be less than or equal to 24 and when tile\_group\_type is equal to B, the sum of sumWeightL0Flags and sumWeightL1Flags shall be less than or equal to 24.

#### Adaptive loop filter data semantics

**alf\_chroma\_idc** equal to 0 specifies that the adaptive loop filter is not applied to Cb and Cr colour components. alf\_chroma\_idc equal to 1 indicates that the adaptive loop filter is applied to the Cb colour component. alf\_chroma\_idc equal to 2 indicates that the adaptive loop filter is applied to the Cr colour component. alf\_chroma\_idc equal to 3 indicates that the adaptive loop filter is applied to 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\_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 binary binarization tb(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. alf\_luma\_eg\_order\_increase\_flag[ i ] equal to 0 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is not 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 ] (7‑58)

**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. When not present, alf\_luma\_coeff\_flag[ sigFiltIdx ] is set equal to 1.

**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.

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 } (7‑59)

k = expGoOrderY[ golombOrderIdxY[ j ] ] (7‑60)

**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 ] is equal to 1), 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.

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 ] \* (7‑61)  
 ( 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 ]  (7‑62)

The luma filter coefficients AlfCoeffL with elements AlfCoeffL[ filtIdx ][ j ], with filtIdx = 0..NumAlfFilters − 1 and j = 0..11 are derived as follows

AlfCoeffL[ filtIdx ][ j ] = filterCoefficients[ alf\_luma\_coeff\_delta\_idx[ filtIdx ] ][ j ] (7‑63)

The last filter coefficients AlfCoeffL[ filtIdx ][ 12 ] for filtIdx = 0..NumAlfFilters − 1 are derived as follows:

AlfCoeffL[ filtIdx ][ 12 ] = 128 − Σk ( AlfCoeffL[ filtIdx ][ k ]  <<  1 ), with k = 0..11 (7‑64)

It is a requirement of bitstream conformance that the values of AlfCoeffL[ filtIdx ][ j ] with filtIdx = 0..NumAlfFilters − 1, j = 0..11 shall be in the range of −27 to 27 − 1, inclusive and that the values of AlfCoeffL[ filtIdx ][ 12 ] shall be in the range of 0 to 28 − 1, inclusive.

**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. alf\_chroma\_eg\_order\_increase\_flag[ i ] equal to 0 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is not 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 ] (7‑65)

**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 27 − 1, inclusive.

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

golombOrderIdxC[ ] = { 0, 0, 1, 0, 0, 1 } (7‑66)

k = expGoOrderC[ golombOrderIdxC[ j ] ] (7‑67)

**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 ] is equal to 1), 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.

The chroma filter coefficients AlfCoeffC with elements cC[ j ], with j = 0..5 are derived as follows:

AlfCoeffC[ j ] = alf\_chroma\_coeff\_abs[ j ] \* ( 1 − 2 \* alf\_chroma\_coeff\_sign[ j ] ) (7‑68)

The last filter coefficient for j = 6 is derived as follows:

AlfCoeffC[ 6 ] = 128 − Σk ( AlfCoeffC[ k ]  <<  1 ), with k = 0..5 (7‑69)

It is a requirement of bitstream conformance that the values of AlfCoeffC[ j ] with j = 0..5 shall be in the range of −27 − 1 to 27 − 1, inclusive and that the values of AlfCoeffC[ 6 ] shall be in the range of 0 to 28 − 1, inclusive.

#### Luma mapping with chroma scaling data semantics

**lmcs\_min\_bin\_idx** specifies the minimum bin index used in the luma mapping with chroma scaling construction process. The value of lmcs\_min\_bin\_idx shall be in the range of 0 to 15, inclusive.

**lmcs\_delta\_max\_bin\_idx** specifies the delta value between 15 and the maximum bin index LmcsMaxBinIdx used in the luma mapping with chroma scaling construction process. The value of lmcs\_delta\_max\_bin\_idx shall be in the range of 0 to 15, inclusive. The value of LmcsMaxBinIdx is set equal to 15 − lmcs\_delta\_max\_bin\_idx. The value of LmcsMaxBinIdx shall be larger than or equal to lmcs\_min\_bin\_idx.

**lmcs\_delta\_cw\_prec\_minus1** plus 1 specifies the number of bits used for the representation of the syntax lmcs\_delta\_abs\_cw[ i ]. The value of lmcs\_delta\_cw\_prec\_minus1 shall be in the range of 0 to BitDepthY − 2, inclusive.

**lmcs\_delta\_abs\_cw**[ i ] specifies the absolute delta codeword value for the ith bin.

**lmcs\_delta\_sign\_cw\_flag**[ i ] specifies the sign of the variable lmcsDeltaCW[ i ] as follows:

* If lmcs\_delta\_sign\_cw\_flag[ i ] is equal to 0, lmcsDeltaCW[ i ] is a positive value.
* Otherwise ( lmcs\_delta\_sign\_cw\_flag[ i ] is not equal to 0 ), lmcsDeltaCW[ i ] is a negative value.

When lmcs\_delta\_sign\_cw\_flag[ i ] is not present, it is inferred to be equal to 0.

The variable OrgCW is derived as follows:

OrgCW = (1 << BitDepthY ) / 16 (7‑70)

The variable lmcsDeltaCW[ i ], with i = lmcs\_min\_bin\_idx..LmcsMaxBinIdx, is derived as follows:

lmcsDeltaCW[ i ] = ( 1 − 2 \* lmcs\_delta\_sign\_cw\_flag[ i ] ) \* lmcs\_delta\_abs\_cw[ i ] (7‑71)

The variable lmcsCW[ i ] is derived as follows:

* For i = 0.. lmcs\_min\_bin\_idx − 1, lmcsCW[ i ] is set equal 0.
* For i = lmcs\_min\_bin\_idx..LmcsMaxBinIdx, the following applies:

lmcsCW[ i ] = OrgCW + lmcsDeltaCW[ i ] (7‑72)

The value of lmcsCW[ i ] shall be in the range of (OrgCW>>3) to (OrgCW<<3 − 1), inclusive.

* For i = LmcsMaxBinIdx + 1..15, lmcsCW[ i ] is set equal 0.

It is a requirement of bitstream conformance that the following condition is true:

<= (1 << BitDepthY ) − 1 (7‑73)

The variable InputPivot[ i ], with i = 0..16, is derived as follows:

InputPivot[ i ] = i \* OrgCW (7‑74)

The variable LmcsPivot[ i ] with i = 0..16, the variables ScaleCoeff[ i ] and InvScaleCoeff[ i ] with i = 0..15, are derived as follows:

LmcsPivot[ 0 ] = 0;  
for( i = 0; i <= 15; i++ ) {  
 LmcsPivot[ i + 1 ] = LmcsPivot[ i ] + lmcsCW[ i ]   
 ScaleCoeff[ i ] = ( lmcsCW[ i ] \* (1 << 14) + (1 << (Log2(OrgCW) − 1))) >> (Log2(OrgCW)) (7‑75)  
 if ( lmcsCW[ i ] = = 0 )  
 InvScaleCoeff[ i ] = 0  
 else  
 InvScaleCoeff[ i ] = OrgCW \* (1 << 14) / lmcsCW[ i ]  
}

The variable ChromaScaleCoeff[ i ], with i = 0…15, is derived as follows:

chromaResidualScaleLut[ ] = {16384, 16384, 16384, 16384, 16384, 16384, 16384, 8192, 8192, 8192, 8192, 5461,   
 5461, 5461, 5461, 4096, 4096, 4096, 4096, 3277, 3277, 3277, 3277, 2731, 2731, 2731, 2731, 2341, 2341,   
 2341, 2048, 2048, 2048, 1820, 1820, 1820, 1638, 1638, 1638, 1638, 1489, 1489, 1489, 1489, 1365, 1365,   
 1365, 1365, 1260, 1260, 1260, 1260, 1170, 1170, 1170, 1170, 1092, 1092, 1092, 1092, 1024, 1024, 1024,   
 1024 }  
if ( lmcsCW[ i ] = = 0 )   
 ChromaScaleCoeff[ i ] = (1 << 11)   
else {  
 binCW = BitDepthY > 10  ?  ( lmcsCW[ i ] >> ( BitDepthY − 10 ) )  : (7‑76)  
 BitDepthY < 10  ?  ( lmcsCW[ i ] << ( 10 − BitDepthY ) )  :  lmcsCW[ i ]  
 ChromaScaleCoeff[ i ] = chromaResidualScaleLut[ Clip3( 1, 64, binCW >> 1 ) − 1 ]  
}

The variables ClipRange, LmcsMinVal, and LmcsMaxVal are derived as follows:

ClipRange = ((lmcs\_min\_bin\_idx > 0) && ( LmcsMaxBinIdx < 15 ) (7‑77)

LmcsMinVal = 16 << (BitDepthY − 8) (7‑78)

LmcsMaxVal = 235 << (BitDepthY − 8) (7‑79)

NOTE – Arrays InputPivot[ i ] and LmcsPivot[ i ], ScaleCoeff[ i ], and InvScaleCoeff[ i ], ChromaScaleCoeff[ i ], ClipRange, LmcsMinVal and LmcsMaxVal, are updated only when tile\_group\_lmcs\_model\_present\_flag is equal to 1. Thus, the lmcs model may be sent with an IRAP picture, for example, but lmcs is disabled for that IRAP picture.

### Reference picture list structure semantics

The ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure may be present in an SPS or in a tile group header. Depending on whether the syntax structure is included in a tile group header or an SPS, the following applies:

– If present in a tile group header, the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure specifies reference picture list listIdx of the current picture (the picture containing the tile group).

– Otherwise (present in an SPS), the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure specifies a candidate for reference picture list listIdx, and the term "the current picture" in the semantics specified in the remainder of this clause refers to each picture that 1) has one or more tile groups containing ref\_pic\_list\_idx[ listIdx ] equal to an index into the list of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures included in the SPS, and 2) is in a CVS that has the SPS as the active SPS.

**num\_ref\_entries**[ listIdx ][ rplsIdx ] specifies the number of entries in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure. The value of num\_ref\_entries[ listIdx ][ rplsIdx ] shall be in the range of 0 to sps\_max\_dec\_pic\_buffering\_minus1 + 14, inclusive.

**st\_ref\_pic\_flag**[ listIdx ][ rplsIdx ][ i ] equal to 1 specifies that the i-th entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure is an STRP entry. st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] equal to 0 specifies that the i-th entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure is an LTRP entry. When not present, the value of st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] is inferred to be equal to 1.

The variable NumLtrpEntries[ listIdx ][ rplsIdx ] is derived as follows:

for( i = 0, NumLtrpEntries[ listIdx ][ rplsIdx ] = 0; i < num\_ref\_entries[ listIdx ][ rplsIdx ]; i++ )  
 if( !st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] ) (7‑80)  
 NumLtrpEntries[ listIdx ][ rplsIdx ]++

**abs\_delta\_poc\_st**[ listIdx ][ rplsIdx ][ i ], when the i-th entry is the first STRP entry in ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure, specifies the absolute difference between the picture order count values of the current picture and the picture referred to by the i-th entry, or, when the i-th entry is an STRP entry but not the first STRP entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure, specifies the absolute difference between the picture order count values of the pictures referred to by the i-th entry and by the previous STRP entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure.

The value of abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ] shall be in the range of 0 to 215 − 1, inclusive.

**strp\_entry\_sign\_flag**[ listIdx ][ rplsIdx ][ i ] equal to 1 specifies that i-th entry in the syntax structure ref\_pic\_list\_struct( listIdx, rplsIdx ) has a value greater than or equal to 0. strp\_entry\_sign\_flag[ listIdx ][ rplsIdx ] equal to 0 specifies that the i-th entry in the syntax structure ref\_pic\_list\_struct( listIdx, rplsIdx ) has a value less than 0. When not present, the value of strp\_entry\_sign\_flag[ i ][ j ] is inferred to be equal to 1.

The list DeltaPocSt[ listIdx ][ rplsIdx ] is derived as follows:

for( i = 0; i < num\_ref\_entries[ listIdx ][ rplsIdx ]; i++ ) {  
 if( st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] ) { (7‑81)  
 DeltaPocSt[ listIdx ][ rplsIdx ][ i ] = ( strp\_entry\_sign\_flag[ listIdx ][ rplsIdx ][ i ]) ?  
 abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ] : 0 − abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ]  
 }  
}

**poc\_lsb\_lt**[ listIdx ][ rplsIdx ][ i ] specifies the value of the picture order count modulo MaxPicOrderCntLsb of the picture referred to by the i-th entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure. The length of the poc\_lsb\_lt[ listIdx ][ rplsIdx ][ i ] syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits.

### Tile group data semantics

#### General tile group data semantics

**end\_of\_tile\_one\_bit** shall be equal to 1.

#### Coding tree unit semantics

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

The array IsInSmr[ x ][ y ] specifying whether the sample at ( x, y ) is located inside a shared merging candidate list region, is initialized as follows for x = 0..CtbSizeY − 1 and y = 0..CtbSizeY − 1:

IsInSmr[ x ][ y ] = FALSE (7‑82)

**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 to be equal to 0.

#### Sample adaptive offset semantics

**sao\_merge\_left\_flag** equal to 1 specifies that the syntax elements sao\_type\_idx\_luma, sao\_type\_idx\_chroma, sao\_band\_position, sao\_eo\_class\_luma, sao\_eo\_class\_chroma, sao\_offset\_abs and sao\_offset\_sign are derived from the corresponding syntax elements of the left CTB. sao\_merge\_left\_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the left CTB. When sao\_merge\_left\_flag is not present, it is inferred to be equal to 0.

**sao\_merge\_up\_flag** equal to 1 specifies that the syntax elements sao\_type\_idx\_luma, sao\_type\_idx\_chroma, sao\_band\_position, sao\_eo\_class\_luma, sao\_eo\_class\_chroma, sao\_offset\_abs and sao\_offset\_sign are derived from the corresponding syntax elements of the above CTB. sao\_merge\_up\_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the above CTB. When sao\_merge\_up\_flag is not present, it is inferred to be equal to 0.

**sao\_type\_idx\_luma** specifies the offset type for the luma component. The array SaoTypeIdx[ cIdx ][ rx ][ ry ] specifies the offset type as specified in Table 7‑4 for the CTB at the location ( rx, ry ) for the colour component cIdx. The value of SaoTypeIdx[ 0 ][ rx ][ ry ] is derived as follows:

* If sao\_type\_idx\_luma is present, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to sao\_type\_idx\_luma.
* Otherwise (sao\_type\_idx\_luma is not present), SaoTypeIdx[ 0 ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to SaoTypeIdx[ 0 ][ rx − 1 ][ ry ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to SaoTypeIdx[ 0 ][ rx ][ ry − 1 ].
* Otherwise, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to 0.

**sao\_type\_idx\_chroma** specifies the offset type for the chroma components. The values of SaoTypeIdx[ cIdx ][ rx ][ ry ] are derived as follows for cIdx equal to 1..2:

* If sao\_type\_idx\_chroma is present, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to sao\_type\_idx\_chroma.
* Otherwise (sao\_type\_idx\_chroma is not present), SaoTypeIdx[ cIdx ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to SaoTypeIdx[ cIdx ][ rx − 1 ][ ry ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to SaoTypeIdx[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to 0.

Table 7‑4 – Specification of the SAO type

|  |  |
| --- | --- |
| **SaoTypeIdx[ cIdx ][ rx ][ ry ]** | **SAO type (informative)** |
| 0 | Not applied |
| 1 | Band offset |
| 2 | Edge offset |

**sao\_offset\_abs**[ cIdx ][ rx ][ ry ][ i ] specifies the offset value of i-th category for the CTB at the location ( rx, ry ) for the colour component cIdx.

When sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is not present, it is inferred as follows:

* If sao\_merge\_left\_flag is equal to 1, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal to sao\_offset\_abs[ cIdx ][ rx − 1 ][ ry ][ i ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal to sao\_offset\_abs[ cIdx ][ rx ][ ry − 1 ][ i ].
* Otherwise, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal to 0.

**sao\_offset\_sign**[ cIdx ][ rx ][ ry ][ i ] specifies the sign of the offset value of i-th category for the CTB at the location ( rx, ry ) for the colour component cIdx.

When sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is not present, it is inferred as follows:

* If sao\_merge\_left\_flag is equal to 1, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal to sao\_offset\_sign[ cIdx ][ rx − 1 ][ ry ][ i ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal to sao\_offset\_sign[ cIdx ][ rx ][ ry − 1 ][ i ].
* Otherwise, if SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2, the following applies:
* If i is equal to 0 or 1, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal 0.
* Otherwise (i is equal to 2 or 3), sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal 1.
* Otherwise, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal 0.

The list SaoOffsetVal[ cIdx ][ rx ][ ry ][ i ] for i ranging from 0 to 4, inclusive, is derived as follows:

SaoOffsetVal[ cIdx ][ rx ][ ry ][ 0 ] = 0  
for( i = 0; i < 4; i++ )  
SaoOffsetVal[ cIdx ][ rx ][ ry ][ i + 1 ] = ( 1 − 2 \* sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] ) \* (7‑83)  
 sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ]

**sao\_band\_position**[ cIdx ][ rx ][ ry ] specifies the displacement of the band offset of the sample range when SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 1.

When sao\_band\_position[ cIdx ][ rx ][ ry ] is not present, it is inferred as follows:

* If sao\_merge\_left\_flag is equal to 1, sao\_band\_position[ cIdx ][ rx ][ ry ] is inferred to be equal to sao\_band\_position[ cIdx ][ rx − 1 ][ ry ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_band\_position[ cIdx ][ rx ][ ry ] is inferred to be equal to sao\_band\_position[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, sao\_band\_position[ cIdx ][ rx ][ ry ] is inferred to be equal to 0.

**sao\_eo\_class**\_**luma** specifies the edge offset class for the luma component. The array SaoEoClass[ cIdx ][ rx ][ ry ] specifies the offset type as specified in Table 7‑5 for the CTB at the location ( rx, ry ) for the colour component cIdx. The value of SaoEoClass[ 0 ][ rx ][ ry ] is derived as follows:

* If sao\_eo\_class\_luma is present, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to sao\_eo\_class\_luma.
* Otherwise (sao\_eo\_class\_luma is not present), SaoEoClass[ 0 ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to SaoEoClass[ 0 ][ rx − 1 ][ ry  ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to SaoEoClass[ 0 ][ rx ][ ry − 1 ].
* Otherwise, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to 0.

**sao\_eo\_class\_chroma** specifies the edge offset class for the chroma components. The values of SaoEoClass[ cIdx ][ rx ][ ry ] are derived as follows for cIdx equal to 1..2:

* If sao\_eo\_class\_chroma is present, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to sao\_eo\_class\_chroma.
* Otherwise (sao\_eo\_class\_chroma is not present), SaoEoClass[ cIdx ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to SaoEoClass[ cIdx ][ rx − 1 ][ ry  ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to SaoEoClass[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to 0.

Table 7‑5 – Specification of the SAO edge offset class

|  |  |
| --- | --- |
| **SaoEoClass[ cIdx ][ rx ][ ry ]** | **SAO edge offset class (informative)** |
| 0 | 1D 0-degree edge offset |
| 1 | 1D 90-degree edge offset |
| 2 | 1D 135-degree edge offset |
| 3 | 1D 45-degree edge offset |

#### Coding tree semantics

When all of the following conditions are true, IsInSmr[ x ][ y ] is set equal to TRUE for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* IsInSmr[ x0 ][ y0 ] is equal to FALSE
* cbWidth \* cbHeight / 4 is less than 32
* treeType is not equal to DUAL\_TREE\_CHROMA

When IsInSmr[ x0 ][ y0 ] is equal to TRUE. the arrays SmrX[ x ][ y ], SmrY[ x ][ y ], SmrW[ x ][ y ] and SmrH[ x ][ y ] are derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

SmrX[ x ][ y ] = x0 (7‑84)

SmrY[ x ][ y ] = y0 (7‑85)

SmrW[ x ][ y ] = ( 1  <<  cbWidth ) (7‑86)

SmrH[ x ][ y ] = ( 1  <<  cbHeight ) (7‑87)

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

* The allowed quad split process as specified in clause 6.4.1 is invoked with the coding block size cbSize set equal to cbWidth, the location ( x0, y0 ), the and current multi-type tree depth mttDepth as inputs, and the output is assigned to allowSplitQt.
* The variables maxBtSize, maxTtSize and maxMttDepth are derived as follows:
* If treeType is equal to DUAL\_TREE\_CHROMA, maxBtSize, maxTtSize and maxMttDepth are set equal to MaxBtSizeC, MaxTtSizeC and MaxMttDepthC + depthOffset, respectively.
* Otherwise, maxBtSize, maxTtSize and maxMttDepth are set equal to MaxBtSizeY, MaxTtSizeY and MaxMttDepthY + depthOffset, respectively.
* The allowed binary split process as specified in clause 6.4.2 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 maximum multi-type tree depth with offset maxMttDepth, the maximum binary tree size maxBtSize, and the current partition index partIdx as inputs, and the output is assigned to allowSplitBtVer.
* The allowed binary split process as specified in clause 6.4.2 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 maximum multi-type tree depth with offset maxMttDepth, the maximum binary tree size maxBtSize, and the current partition index partIdx as inputs, and the output is assigned to allowSplitBtHor.
* The allowed ternary split process as specified in clause 6.4.3 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 maximum multi-type tree depth with offset maxMttDepth, and the maximum ternary tree size maxTtSize as inputs, and the output is assigned to allowSplitTtVer.
* The allowed ternary split process as specified in clause 6.4.3 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 maximum multi-type tree depth with offset maxMttDepth, and the maximum ternary tree size maxTtSize as inputs, and the output is assigned to allowSplitTtHor.

**split\_cu\_flag** equal to 0 specifies that a coding unit is not split. split\_cu\_flag equal to 1 specifies that a coding unit is split into four coding units using a quad split as indicated by the syntax element split\_qt\_flag, or 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 split\_cu\_flag is not present, the value of split\_cu\_flag is inferred as follows:

* If one or more of the following conditions are true, the value of 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 split\_cu\_flag is inferred to be equal to 0.

**split\_qt\_flag** specifies whether a coding unit is split into coding units with half horizontal and vertical size.

When split\_qt\_flag is not present, the following applies:

* If allowSplitQt is equal to TRUE, the value of split\_qt\_flag is inferred to be equal to 1.
* Otherwise, the value of split\_qt\_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 7‑6 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 7‑6 – 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 |

When all of the following conditions are true, IsInSmr[ x ][ y ] is set equal to TRUE for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* IsInSmr[ x0 ][ y0 ] is equal to FALSE
* One of the following conditions is true:
* mtt\_split\_cu\_binary\_flag is equal to 1 and  cbWidth \* cbHeight / 2 is less than 32
* mtt\_split\_cu\_binary\_flag is equal to 0 and  cbWidth \* cbHeight / 4 is less than 32
* treeType is not equal to DUAL\_TREE\_CHROMA

When IsInSmr[ x0 ][ y0 ] is equal to TRUE. the arrays SmrX[ x ][ y ], SmrY[ x ][ y ], SmrW[ x ][ y ] and SmrH[ x ][ y ] are derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

SmrX[ x ][ y ] = x0 (7‑88)

SmrY[ x ][ y ] = y0 (7‑89)

SmrW[ x ][ y ] = cbWidth (7‑90)

SmrH[ x ][ y ] = cbHeight (7‑91)

#### Coding unit semantics

When all the following conditions are true, the history-based motion vector predictor list for the shared merging candidate list region is updated by setting NumHmvpSmrCand equal to the NumHmvpCand, and setting HmvpSmrCandList[ i ] equal to HmvpCandList[ i ] for i = 0..NumHmvpCand − 1:

* IsInSmr[ x0 ][ y0 ] is equal to TRUE
* SmrX[ x0 ][ y0 ]  is equal to x0
* SmrY[ x0 ][ y0 ]  is equal to y0

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

CbPosX[ x ][ y ] = x0 (7‑92)

CbPosY[ x ][ y ] = y0 (7‑93)

CbWidth[ x ][ y ] = cbWidth (7‑94)

CbHeight[ x ][ y ] = cbHeight (7‑95)

**cu\_skip\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B tile group, no more syntax elements except one or more of the following are parsed after cu\_skip\_flag[ x0 ][ y0 ]: the IBC mode flag pred\_mode\_ibc\_flag [ x0 ][ y0 ], the merge plus MVD flag mmvd\_flag[ x0 ][ y0 ], the merge plus MVD index mmvd\_merge\_flag[ x0 ][ y0 ], the merge plus MVD distance index mmvd distance\_idx[ x0 ][ y0 ], the merge plus MVD direction index mmvd\_direction\_idx[ x0 ][ y0 ], the merging candidate index merge\_idx[ x0 ][ y0 ] the subblock-based merge flag merge\_subblock\_flag[ x0 ][ y0 ], the subblock-based merging candidate index merge\_subblock\_idx[ x0 ][ y0 ], the merge triangle flag merge\_triangle\_flag[ x0 ][ y0 ], and the merge triangle index merge\_triangle\_index[ x0 ][ y0 ]; when decoding an I tile group, no more syntax elements except merge\_idx[ 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, it is inferred to be equal to 1 when decoding an I tile group, and equal to 0 when decoding a P or B tile group, respectively.

**pred\_mode\_ibc\_flag** equal to 1 specifies that the current coding unit is coded in IBC prediction mode. pred\_mode\_ibc\_flag equal to 0 specifies that the current coding unit is not coded in IBC prediction mode.

When pred\_mode\_ibc\_flag is not present, it is inferred to be equal to the value of sps\_ibc\_enabled\_flag when decoding an I tile group, and 0 when decoding a P or B tile group, respectively.

When pred\_mode\_ibc\_flag is equal to 1, the variable CuPredMode[ x ][ y ] is set to be equal to MODE\_IBC for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

**pcm\_flag**[ x0 ][ y0 ] equal to 1 specifies that the pcm\_sample( ) syntax structure is present and the transform\_tree( ) syntax structure is not present in the coding unit including the luma coding block at the location ( x0, y0 ). pcm\_flag[ x0 ][ y0 ] equal to 0 specifies that pcm\_sample( ) syntax structure is not present. When pcm\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The value of pcm\_flag[ x0 + i ][ y0 + j ] with i = 1..cbWidth − 1, j = 1..cbHeight − 1 is inferred to be equal to pcm\_flag[ x0 ][ y0 ].

**pcm\_alignment\_zero\_bit** is a bit equal to 0.

**intra\_luma\_ref\_idx**[ x0 ][ y0 ] specifies the intra prediction reference line index IntraLumaRefLineIdx[ x ][ y ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1 as specified in Table 7‑7.

When intra\_luma\_ref\_idx[ x0 ][ y0 ] is not present it is inferred to be equal to 0.

Table 7‑7 – Specification of IntraLumaRefLineIdx[ x ][ y ] based on intra\_luma\_ref\_idx[ x0 ][ y0 ].

|  |  |
| --- | --- |
| intra\_luma\_ref\_idx[ x0 ][ y0 ] | IntraLumaRefLineIdx[ x ][ y ] x = x0..x0 + cbWidth − 1 y = y0..y0 + cbHeight − 1 |
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |

**intra\_subpartitions\_mode\_flag**[ x0 ][ y0 ] equal to 1 specifies that the current intra coding unit is partitioned into NumIntraSubPartitions[ x0 ][ y0 ] rectangular transform block subpartitions. intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] equal to 0 specifies that the current intra coding unit is not partitioned into rectangular transform block subpartitions.

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

**intra\_subpartitions\_split\_flag**[ x0 ][ y0 ] specifies whether the intra subpartitions split type is horizontal or vertical. When intra\_subpartitions\_split\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If cbHeight is greater than MaxTbSizeY, intra\_subpartitions\_split\_flag[ x0 ][ y0 ] is inferred to be equal to 0.
* Otherwise (cbWidth is greater than MaxTbSizeY), intra\_subpartitions\_split\_flag[ x0 ][ y0 ] is inferred to be equal to 1.

The variable IntraSubPartitionsSplitType specifies the type of split used for the current luma coding block as illustrated in Table 7‑8. IntraSubPartitionsSplitType is derived as follows:

* If intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] is equal to 0, IntraSubPartitionsSplitType is set equal to 0.
* Otherwise, the IntraSubPartitionsSplitType is set equal to 1 + intra\_subpartitions\_split\_flag[ x0 ][ y0 ].

**Table 7‑8 – Name association to IntraSubPartitionsSplitType**

|  |  |
| --- | --- |
| IntraSubPartitionsSplitType | Name of IntraSubPartitionsSplitType |
| 0 | ISP\_NO\_SPLIT |
| 1 | ISP\_HOR\_SPLIT |
| 2 | ISP\_VER\_SPLIT |

The variable NumIntraSubPartitions specifies the number of transform block subpartitions an intra luma coding block is divided into. NumIntraSubPartitions is derived as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, NumIntraSubPartitions is set equal to 1.
* Otherwise, if one of the following conditions is true, NumIntraSubPartitions is set equal to 2:
  + cbWidth is equal to 4 and cbHeight is equal to 8,
  + cbWidth is equal to 8 and cbHeight is equal to 4.
* Otherwise, NumIntraSubPartitions is set equal to 4.

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 8.4.2.

When intra\_luma\_mpm\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 1.

**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.

**inter\_pred\_idc**[ x0 ][ y0 ] specifies whether list0, list1, or bi-prediction is used for the current coding unit according to Table 7‑9. 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 7‑9 – Name association to inter prediction mode

|  |  |  |
| --- | --- | --- |
| **inter\_pred\_idc** | **Name of inter\_pred\_idc** | |
| ( cbWidth + cbHeight )  !=  8 | ( cbWidth + cbHeight )  = =  8 |
| 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.

**sym\_mvd\_flag**[ x0 ][ y0 ] equal to 1 specifies that the syntax elements ref\_idx\_l0[ x0 ][ y0 ] and ref\_idx\_l1[ x0 ][ y0 ], and the mvd\_coding( x0, y0, refList ,cpIdx ) syntax structure for refList equal to 1 are not presented. 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 sym\_mvd\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**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 as follows:

* If sym\_mvd\_flag[ x0 ][ y0 ] is equal to 1, ref\_idx\_l0[ x0 ][ y0 ] is inferred to be equal to RefIdxSymL0.
* Otherwise (sym\_mvd\_flag[ x0 ][ y0 ] is equal to 0), ref\_idx\_l0[ x0 ][ y0 ] is inferred to be equal to 0.

**ref\_idx\_l1**[ x0 ][ y0 ] has the same semantics as ref\_idx\_l0, with l0, L0 and list 0 replaced by l1, L1 and list 1, respectively.

**inter\_affine\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B tile group, 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 tile group, 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 7‑10. 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\_subblock\_flag[ x0 ][ y0 ] (7‑96)

* 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 ] (7‑97)

Table 7‑10 – 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\_flag**[ 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. amvr\_flag[ x0 ][ y0 ] equal to 0 specifies that the resolution of the motion vector difference is 1/4 of a luma sample. amvr\_flag[ x0 ][ y0 ] equal to 1 specifies that the resolution of the motion vector difference is further specified by amvr\_precision\_flag[ x0 ][ y0 ].

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

* If CuPredMode[ x0 ][ y0 ] is equal to MODE\_IBC, amvr\_flag[ x0 ][ y0 ] is inferred to be equal to 1.
* Otherwise ( CuPredMode[ x0 ][ y0 ] is not equal to MODE\_IBC ), amvr\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**amvr\_precision\_flag**[ x0 ][ y0 ] equal to 0 specifies that the resolution of the motion vector difference is one integer luma sample if inter\_affine\_flag[ x0 ][ y0 ] is equal to 0, and 1/16 of a luma sample otherwise. amvr\_precision\_flag[ x0 ][ y0 ] equal to 1 specifies that the resolution of the motion vector difference is four luma samples if inter\_affine\_flag[ x0 ][ y0 ] is equal to 0, and one integer luma sample otherwise. 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\_precision\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The motion vector differences are modified as follows:

* If inter\_affine\_flag[ x0 ][ y0 ] is equal to 0, the variable MvShift is derived and the variables MvdL0[ x0 ][ y0 ][ 0 ], MvdL0[ x0 ][ y0 ][ 1 ], MvdL1[ x0 ][ y0 ][ 0 ], MvdL1[ x0 ][ y0 ][ 1 ] are modified as follows:

MvShift = ( amvr\_flag[ x0 ][ y0 ] + amvr\_precision\_flag[ x0 ][ y0 ] ) << 1 (7‑98)

MvdL0[ x0 ][ y0 ][ 0 ] = MvdL0[ x0 ][ y0 ][ 0 ]  << ( MvShift + 2 ) (7‑99)

MvdL0[ x0 ][ y0 ][ 1 ] = MvdL0[ x0 ][ y0 ][ 1 ]  <<  ( MvShift + 2 ) (7‑100)

MvdL1[ x0 ][ y0 ][ 0 ] = MvdL1[ x0 ][ y0 ][ 0 ]<<( MvShift + 2 ) (7‑101)

MvdL1[ x0 ][ y0 ][ 1 ] = MvdL1[ x0 ][ y0 ][ 1 ]  << ( MvShift + 2 ) (7‑102)

* Otherwise (inter\_affine\_flag[ x0 ][ y0 ] is equal to 1), the variable MvShift is derived and the variables MvdCpL0[ x0 ][ y0 ][ 0 ][ 0 ], MvdCpL0[ x0 ][ y0 ][ 0 ][ 1 ], MvdCpL0[ x0 ][ y0 ][ 1 ][ 0 ], MvdCpL0[ x0 ][ y0 ][ 1 ][ 1 ], MvdCpL0[ x0 ][ y0 ][ 2 ][ 0 ] and MvdCpL0[ x0 ][ y0 ][ 2 ][ 1 ] are modified as follows:

MvShift = amvr\_precision\_flag[ x0 ][ y0 ]  ?   
  ( amvr\_precision\_flag[ x0 ][ y0 ]  <<  1 )  :  ( − (amvr\_flag[ x0 ][ y0 ]  <<  1) ) ) (7‑103)

MvdCpL0[ x0 ][ y0 ][ 0 ][ 0 ] = MvdCpL0[ x0 ][ y0 ][ 0 ][ 0 ]  << ( MvShift + 2 ) (7‑104)

MvdCpL1[ x0 ][ y0 ] [ 0 ][ 1 ] = MvdCpL1[ x0 ][ y0 ][ 0 ][ 1 ]  <<  ( MvShift + 2 ) (7‑105)

MvdCpL0[ x0 ][ y0 ][ 1 ][ 0 ] = MvdCpL0[ x0 ][ y0 ][ 1 ][ 0 ]  << ( MvShift + 2 ) (7‑106)

MvdCpL1[ x0 ][ y0 ] [ 1 ][ 1 ] = MvdCpL1[ x0 ][ y0 ][ 1 ][ 1 ]  <<  ( MvShift + 2 ) (7‑107)

MvdCpL0[ x0 ][ y0 ][ 2 ][ 0 ] = MvdCpL0[ x0 ][ y0 ][ 2 ][ 0 ]  << ( MvShift + 2 ) (7‑108)

MvdCpL1[ x0 ][ y0 ] [ 2 ][ 1 ] = MvdCpL1[ x0 ][ y0 ][ 2 ][ 1 ]  <<  ( MvShift + 2 ) (7‑109)

**gbi\_idx**[ x0 ][ y0 ] specifies the weight index of bi-prediction with CU weights. 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 gbi\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**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.

**cu\_sbt\_flag** equal to 1 specifies that for the current coding unit, subblock transform is used. cu\_sbt\_flag equal to 0 specifies that for the current coding unit, subblock transform is not used.

When cu\_sbt\_flag is not present, its value is inferred to be equal to 0.

NOTE – : When subblock transform is used, a coding unit is split into two transform units; one transform unit has residual data, the other does not have residual data.

**cu\_sbt\_quad\_flag** equal to 1 specifies that for the current coding unit, the subblock transform includes a transform unit of 1/4 size of the current coding unit. cu\_sbt\_quad\_flag equal to 0 specifies that for the current coding unit the subblock transform includes a transform unit of 1/2 size of the current coding unit.

When cu\_sbt\_quad\_flag is not present, its value is inferred to be equal to 0.

**cu\_sbt\_horizontal\_flag** equal to 1 specifies that the current coding unit is split horizontally into 2 transform units. cu\_sbt\_horizontal\_flag[ x0 ][ y0 ] equal to 0 specifies that the current coding unit is split vertically into 2 transform units.

When cu\_sbt\_horizontal\_flag is not present, its value is derived as follows:

* If cu\_sbt\_quad\_flag is equal to 1, cu\_sbt\_horizontal\_flag is set to be equal to allowSbtHorQ.
* Otherwise (cu\_sbt\_quad\_flag is equal to 0), cu\_sbt\_horizontal\_flag is set to be equal to allowSbtHorH.

**cu\_sbt\_pos\_flag** equal to 1 specifies that the tu\_cbf\_luma, tu\_cbf\_cb and tu\_cbf\_cr of the first transform unit in the current coding unit are not present in the bitstream. cu\_sbt\_pos\_flag equal to 0 specifies that the tu\_cbf\_luma, tu\_cbf\_cb and tu\_cbf\_cr of the second transform unit in the current coding unit are not present in the bitstream.

The variable SbtNumFourthsTb0 is derived as follows:

sbtMinNumFourths = cu\_sbt\_quad\_flag  ?  1  :  2 (7‑110)

SbtNumFourthsTb0 = cu\_sbt\_pos\_flag  ?  ( 4 − sbtMinNumFourths )  :  sbtMinNumFourths (7‑111)

#### PCM sample semantics

**pcm\_sample\_luma**[ i ] represents a coded luma sample value in the raster scan within the coding unit. The number of bits used to represent each of these samples is PcmBitDepthY.

**pcm\_sample\_chroma**[ i ] represents a coded chroma sample value in the raster scan within the coding unit. The first half of the values represent coded Cb samples and the remaining half of the values represent coded Cr samples. The number of bits used to represent each of these samples is PcmBitDepthC.

#### Merge data semantics

**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.

**mmvd\_flag**[ x0 ][ y0 ] equal to 1 specifies that merge mode with motion vector difference is used to generate the inter prediction parameters of 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 mmvd\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**mmvd\_merge\_flag**[ x0 ][ y0 ] specifies whether the first (0) or the second (1) candidate in the merging candidate list is used with the motion vector difference derived from mmvd\_distance\_idx[ x0 ][ y0 ] and mmvd\_direction\_idx[ x0 ][ y0 ]. 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.

**mmvd\_distance\_idx**[ x0 ][ y0 ] specifies the index used to derive MmvdDistance[ x0 ][ y0 ] as specified in Table 7‑11. 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 7‑11 – Specification of MmvdDistance[ x0 ][ y0 ] based on mmvd\_distance\_idx[ x0 ][ y0 ].

|  |  |  |
| --- | --- | --- |
| mmvd\_distance\_idx[ x0 ][ y0 ] | MmvdDistance[ x0 ][ y0 ] | |
| tile\_group\_fpel\_mmvd\_enabled\_flag = = 0 | tile\_group\_fpel\_mmvd\_enabled\_flag = = 1 |
| 0 | 1 | 4 |
| 1 | 2 | 8 |
| 2 | 4 | 16 |
| 3 | 8 | 32 |
| 4 | 16 | 64 |
| 5 | 32 | 128 |
| 6 | 64 | 256 |
| 7 | 128 | 512 |

**mmvd\_direction\_idx**[ x0 ][ y0 ] specifies index used to derive MmvdSign[ x0 ][ y0 ] as specified in Table 7‑12. 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 7‑12 – Specification of MmvdSign[ x0 ][ y0 ] based on mmvd\_direction\_idx[ x0 ][ y0 ]

|  |  |  |
| --- | --- | --- |
| mmvd\_direction\_idx[ x0 ][ y0 ] | MmvdSign[ x0 ][ y0 ][0] | MmvdSign[ x0 ][ y0 ][1] |
| 0 | +1 | 0 |
| 1 | -1 | 0 |
| 2 | 0 | +1 |
| 3 | 0 | -1 |

Both components of of the merge plus MVD offset MmvdOffset[ x0 ][ y0 ] are derived as follows:

MmvdOffset[ x0 ][ y0 ][ 0 ] = ( MmvdDistance[ x0 ][ y0 ] << 2 ) \* MmvdSign[ x0 ][ y0 ][0] (7‑112)

MmvdOffset[ x0 ][ y0 ][ 1 ] = ( MmvdDistance[ x0 ][ y0 ] << 2 ) \* MmvdSign[ x0 ][ y0 ][1] (7‑113)

**merge\_subblock\_flag**[ x0 ][ y0 ] specifies whether the subblock-based inter prediction parameters for the current coding unit are inferred from neighbouring blocks. 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\_subblock\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**merge\_subblock\_idx**[ x0 ][ y0 ] specifies the merging candidate index of the subblock-based 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\_subblock\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**ciip\_flag**[ x0 ][ y0 ] specifies whether the combined inter-picture merge and intra-picture prediction is applied 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 ciip\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The syntax elements **ciip\_luma\_mpm\_flag**[ x0 ][ y0 ], and **ciip\_luma\_mpm\_idx**[ x0 ][ y0 ] specify the intra prediction mode for luma samples used in combined inter-picture merge and intra-picture 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 intra prediction mode is derived according to clause 8.5.6.

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

* If cbWidth is greater than 2 \* cbHeight or cbHeight is greater than 2 \* cbWidth, ciip\_luma\_mpm\_flag[ x0 ][ y0 ] is inferred to be equal to 1.
* Otherwise, ciip\_luma\_mpm\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**merge\_triangle\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a B tile group, triangular shape based motion compensation is used to generate the prediction samples of the current coding unit. merge\_triangle\_flag[ x0 ][ y0 ] equal to 0 specifies that the coding unit is not predicted by triangular shape based motion compensation. When merge\_triangle\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**merge\_triangle\_split\_dir**[ x0 ][ y0 ] specifies the splitting direction of merge triangle mode. 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\_triangle\_split\_dir[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**merge\_triangle\_idx0**[ x0 ][ y0 ] specifies the first merging candidate index of the triangular shape based motion compensation 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\_triangle\_idx0[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**merge\_triangle\_idx1**[ x0 ][ y0 ] specifies the second merging candidate index of the triangular shape based motion compensation 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\_triangle\_idx1[ x0 ][ y0 ] is not present, it 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 as follows:

* If mmvd\_flag[ x0 ][ y0 ] is equal to 1, merge\_idx[ x0 ][ y0 ] is inferred to be equal to mmvd\_merge\_flag[ x0 ][ y0 ].
* Otherwise (mmvd\_flag[ x0 ][ y0 ] is equal to 0), merge\_idx[ x0 ][ y0 ] is inferred to be equal to 0.

#### 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 ] ) (7‑114)

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 derived 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 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 ] for compIdx = 0..1.
* Otherwise (refList is equal to 1), MvdCpL1[ x0 ][ y0 ][ cpIdx ][ compIdx ] is set equal to lMvd[ compIdx ] 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 7.3.6.11, 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 in the current CU, its value is inferred as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, tu\_cbf\_luma[ x0 ][ y0 ] is inferred to be equal to 0.
* Otherwise, tu\_cbf\_luma[ x0 ][ y0 ] is inferred to be equal to 1.

**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 in the current CU, 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 in the current CU, its value is inferred to be equal to 0.

**cu\_qp\_delta\_abs** specifies the absolute value of the difference CuQpDeltaVal between the quantization parameter of the current coding unit and its prediction.

**cu\_qp\_delta\_sign\_flag** specifies the sign of CuQpDeltaVal as follows:

* If cu\_qp\_delta\_sign\_flag is equal to 0, the corresponding CuQpDeltaVal has a positive value.
* Otherwise (cu\_qp\_delta\_sign\_flag is equal to 1), the corresponding CuQpDeltaVal has a negative value.

When cu\_qp\_delta\_sign\_flag is not present, it is inferred to be equal to 0.

When cu\_qp\_delta\_abs is present, the variables IsCuQpDeltaCoded and CuQpDeltaVal are derived as follows:

IsCuQpDeltaCoded = 1 (7‑115)

CuQpDeltaVal = cu\_qp\_delta\_abs \* ( 1 − 2 \* cu\_qp\_delta\_sign\_flag ) (7‑116)

The value of CuQpDeltaVal shall be in the range of −( 32 + QpBdOffsetY / 2 ) to +( 31 + QpBdOffsetY / 2 ), inclusive.

**transform\_skip\_flag**[ x0 ][ y0 ] specifies whether a transform is applied to the luma 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. transform\_skip\_flag[ x0 ][ y0 ] equal to 1 specifies that no transform is applied to the luma transform block. transform\_skip\_flag[ x0 ][ y0 ] equal to 0 specifies that the decision whether transform is applied to the luma transform block or not depends on other syntax elements. When transform\_skip\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**tu\_mts\_idx**[ x0 ][ y0 ] specifies which transform kernels are applied to the residual samples along the horizontal and vertical direction 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 tu\_mts\_idx[ 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 7.3.6.11, it is inferred to be equal to 0. When the value of AbsLevelPass1[ xC ][ yC ] is not specified in clause 7.3.6.11, 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 ) (7‑117)

CoeffMax = ( 1 << 15 ) − 1 (7‑118)

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

QStateTransTable[ ][ ] = { { 0, 2 }, { 2, 0 }, { 1, 3 }, { 3, 1 } } (7‑119)

**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.

When last\_sig\_coeff\_x\_prefix is not present, it is inferred to be 0.

**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.

When last\_sig\_coeff\_y\_prefix is not present, it is inferred to be 0.

**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 (7‑120)

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

LastSignificantCoeffX = ( 1  <<  ( (last\_sig\_coeff\_x\_prefix  >>  1 ) − 1 ) ) \* (7‑121)  
 ( 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 (7‑122)

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

LastSignificantCoeffY = ( 1  <<  ( ( last\_sig\_coeff\_y\_prefix  >>  1 ) − 1 ) ) \* (7‑123)  
 ( 2 + ( last\_sig\_coeff\_y\_prefix & 1 ) ) + last\_sig\_coeff\_y\_suffix

**coded\_sub\_block\_flag**[ xS ][ yS ] specifies the following for the subblock at location ( xS, yS ) within the current transform block, where a subblock 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 subblock 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 subblock at location ( xS, yS ).
* Otherwise, at least one of the 16 transform coefficient levels of the subblock 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 true, sig\_coeff\_flag[ 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.

**abs\_level\_gt1\_flag**[ n ] specifies whether the absolute value of the transform coefficient level (at scanning position n) is greater than 1. When abs\_level\_gt1\_flag[ n ] is not present, it 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.

**abs\_level\_gt3\_flag**[ n ] specifies whether the absolute value of the transform coefficient level (at scanning position n) is greater than 3. When abs\_level\_gt3\_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.

**dec\_abs\_level**[ n ] is an intermediate value that is coded with Golomb-Rice code at the scanning position n. Given ZeroPos[ n ] that is derived in clause 9.5.3.2 during the parsing of dec\_abs\_level[ n ], the absolute value of a transform coefficient level at location ( xC, yC ) AbsLevel[ xC ][ yC ] is derived using as follows:

* If dec\_abs\_level[ n ] is equal to ZeroPos[ n ], AbsLevel[ xC ][ yC ] is set equal to 0.
* Otherwise if dec\_abs\_level[ n ] is less than ZeroPos[ n ], AbsLevel[ xC ][ yC ] is set equal to dec\_abs\_level[ n ] + 1;
* Otherwise (dec\_abs\_level[ n ] is greater than ZeroPos[ n ]), AbsLevel[ xC ][ yC ] is set equal to dec\_abs\_level[ n ].

It is a requirement of bitstream conformance that the value of dec\_abs\_level[ 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.

# Decoding process

## General decoding process

* + 1. **General**

Input to this process is a bitstream. Output of this process is a list of decoded pictures.

The decoding process is specified such that all decoders that conform to a specified profile and level will produce numerically identical cropped decoded output pictures when invoking the decoding process associated with that profile for a bitstream conforming to that profile and level. Any decoding process that produces identical cropped decoded output pictures to those produced by the process described herein (with the correct output order or output timing, as specified) conforms to the decoding process requirements of this Specification.

At the beginning of decoding a CVS, after activating the SPS RBSP that is active for the entire CVS and before decoding any VCL NAL units of the CVS, clause 8.1.2 is invoked with the CVS as input.

* + 1. **CVS decoding process**

Input to this process is a CVS. Output of this process is a list of decoded pictures.

The variable HighestTid, which identifies the highest temporal sub-layer to be decoded, is specified as follows:

– If some external means, not specified in this Specification, is available to set HighestTid, HighestTid is set by the external means.

– Otherwise, HighestTid is set equal to sps\_max\_sub\_layers\_minus1.

The sub-bitstream extraction process as specified in clause 10 is applied with the CVS and HighestTid as inputs, and the output is assigned to a bitstream referred to as BitstreamToDecode.

Clause 8.1.3 is repeatedly invoked for each coded picture in BitstreamToDecode in decoding order.

* + 1. **Decoding process for a coded picture**

The decoding processes specified in this clause apply to each coded picture, referred to as the current picture and denoted by the variable CurrPic, in BitstreamToDecode.

Depending on the value of chroma\_format\_idc, the number of sample arrays of the current picture is as follows:

– If chroma\_format\_idc is equal to 0, the current picture consists of 1 sample array SL.

– Otherwise (chroma\_format\_idc is not equal to 0), the current picture consists of 3 sample arrays SL, SCb, SCr.

The decoding process for the current picture takes as inputs the syntax elements and upper-case variables from clause 7. When interpreting the semantics of each syntax element in each NAL unit, the term "the bitstream" (or part thereof, e.g., a CVS of the bitstream) refers to BitstreamToDecode (or part thereof).

When the current picture is an IRAP picture, the following applies:

– If the current picture is an IDR picture, the first picture in the bitstream in decoding order, or the first picture that follows an end of sequence NAL unit in decoding order, the variable NoRaslOutputFlag is set equal to 1.

– Otherwise, if some external means not specified in this Specification is available to set the variable HandleCraAsCvsStartFlag to a value for the current picture, the variable HandleCraAsCvsStartFlag is set equal to the value provided by the external means and the variable NoRaslOutputFlag is set equal to HandleCraAsCvsStartFlag.

– Otherwise, the variable HandleCraAsCvsStartFlag is set equal to 0 and the variable NoRaslOutputFlag is set equal to 0.

Depending on the value of separate\_colour\_plane\_flag, the decoding process is structured as follows:

– If separate\_colour\_plane\_flag is equal to 0, the decoding process is invoked a single time with the current picture being the output.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), the decoding process is invoked three times. Inputs to the decoding process are all NAL units of the coded picture with identical value of colour\_plane\_id. The decoding process of NAL units with a particular value of colour\_plane\_id is specified as if only a CVS with monochrome colour format with that particular value of colour\_plane\_id would be present in the bitstream. The output of each of the three decoding processes is assigned to one of the 3 sample arrays of the current picture, with the NAL units with colour\_plane\_id equal to 0, 1 and 2 being assigned to SL, SCb and SCr, respectively.

NOTE – The variable ChromaArrayType is derived as equal to 0 when separate\_colour\_plane\_flag is equal to 1 and chroma\_format\_idc is equal to 3. In the decoding process, the value of this variable is evaluated resulting in operations identical to that of monochrome pictures (when chroma\_format\_idc is equal to 0).

The decoding process operates as follows for the current picture CurrPic:

1. The decoding of NAL units is specified in clause 8.2.
2. The processes in clause 8.3 specify the following decoding processes using syntax elements in the tile group header layer and above:

– Variables and functions relating to picture order count are derived as specified in clause 8.3.1. This needs to be invoked only for the first tile group of a picture.

– At the beginning of the decoding process for each tile group of a non-IDR picture, the decoding process for reference picture lists construction specified in clause 8.3.2 is invoked for derivation of reference picture list 0 (RefPicList[ 0 ]) and reference picture list 1 (RefPicList[ 1 ]).

– The decoding process for reference picture marking in clause 8.3.3 is invoked, wherein reference pictures may be marked as "unused for reference" or "used for long-term reference". This needs to be invoked only for the first tile group of a picture.

1. [Ed. (YK): Add herein the invocation of the decoding processes for coding tree units, scaling, transform, in-loop filtering, etc.]
2. After all tile groups of the current picture have been decoded, the current decoded picture is marked as "used for short-term reference".

## NAL unit decoding process

Inputs to this process are NAL units of the current picture and their associated non-VCL NAL units.

Outputs of this process are the parsed RBSP syntax structures encapsulated within the NAL units.

The decoding process for each NAL unit extracts the RBSP syntax structure from the NAL unit and then parses the RBSP syntax structure.

## Tile group decoding process

### Decoding process for picture order count

Output of this process is PicOrderCntVal, the picture order count of the current picture.

Each coded picture is associated with a picture order count variable, denoted as PicOrderCntVal.

When the current picture is not an IRAP picture with NoRaslOutputFlag equal to 1, the variables prevPicOrderCntLsb and prevPicOrderCntMsb are derived as follows:

* Let prevTid0Pic be the previous picture in decoding order that has TemporalId equal to 0 and that is not a RASL or RADL picture.
* The variable prevPicOrderCntLsb is set equal to tile\_group\_pic\_order\_cnt\_lsb of prevTid0Pic.
* The variable prevPicOrderCntMsb is set equal to PicOrderCntMsb of prevTid0Pic.

The variable PicOrderCntMsb of the current picture is derived as follows:

* If the current picture is an IRAP picture with NoRaslOutputFlag equal to 1, PicOrderCntMsb is set equal to 0.
* Otherwise, PicOrderCntMsb is derived as follows:

if( ( tile\_group\_pic\_order\_cnt\_lsb < prevPicOrderCntLsb ) &&  
 ( ( prevPicOrderCntLsb − tile\_group\_pic\_order\_cnt\_lsb ) >= ( MaxPicOrderCntLsb / 2 ) ) )  
 PicOrderCntMsb = prevPicOrderCntMsb + MaxPicOrderCntLsb (8‑1)  
else if( (tile\_group\_pic\_order\_cnt\_lsb > prevPicOrderCntLsb ) &&  
 ( ( tile\_group\_pic\_order\_cnt\_lsb − prevPicOrderCntLsb ) > ( MaxPicOrderCntLsb / 2 ) ) )  
 PicOrderCntMsb = prevPicOrderCntMsb − MaxPicOrderCntLsb  
else  
 PicOrderCntMsb = prevPicOrderCntMsb

PicOrderCntVal is derived as follows:

PicOrderCntVal = PicOrderCntMsb + tile\_group\_pic\_order\_cnt\_lsb (8‑2)

NOTE 1 – All IRAP pictures with NoRaslOutputFlag equal to 1 will have PicOrderCntVal equal to tile\_group\_pic\_order\_cnt\_lsb since for IRAP pictures with NoRaslOutputFlag equal to 1 PicOrderCntMsb is set equal to 0.

The value of PicOrderCntVal shall be in the range of −231 to 231 − 1, inclusive.

In one CVS, the PicOrderCntVal values for any two coded pictures shall not be the same.

The function PicOrderCnt( picX ) is specified as follows:

PicOrderCnt( picX ) = PicOrderCntVal of the picture picX (8‑3)

The function DiffPicOrderCnt( picA, picB ) is specified as follows:

DiffPicOrderCnt( picA, picB ) = PicOrderCnt( picA ) − PicOrderCnt( picB ) (8‑4)

The bitstream shall not contain data that result in values of DiffPicOrderCnt( picA, picB ) used in the decoding process that are not in the range of −215 to 215 − 1, inclusive.

NOTE 2 – Let X be the current picture and Y and Z be two other pictures in the same CVS, Y and Z are considered to be in the same output order direction from X when both DiffPicOrderCnt( X, Y ) and DiffPicOrderCnt( X, Z ) are positive or both are negative.

### Decoding process for reference picture lists construction

This process is invoked at the beginning of the decoding process for each tile group of a non-IDR picture.

Reference pictures are addressed through reference indices. A reference index is an index into a reference picture list. When decoding an I tile group, no reference picture list is used in decoding of the tile group data. When decoding a P tile group, only reference picture list 0 (i.e., RefPicList[ 0 ]), is used in decoding of the tile group data. When decoding a B tile group, both reference picture list 0 and reference picture list 1 (i.e., RefPicList[ 1 ]) are used in decoding of the tile group data.

At the beginning of the decoding process for each tile group of a non-IDR picture, the reference picture lists RefPicList[ 0 ] and RefPicList[ 1 ] are derived. The reference picture lists are used in marking of reference pictures as specified in clause 8.3.3 or in decoding of the tile group data.

NOTE 1 – For an I tile group of a non-IDR picture that it is not the first tile group of the picture, RefPicList[ 0 ] and RefPicList[ 1 ] may be derived for bitstream conformance checking purpose, but their derivation is not necessary for decoding of the current picture or pictures following the current picture in decoding order. For a P tile group that it is not the first tile group of a picture, RefPicList[ 1 ] may be derived for bitstream conformance checking purpose, but its derivation is not necessary for decoding of the current picture or pictures following the current picture in decoding order.

The reference picture lists RefPicList[ 0 ] and RefPicList[ 1 ] are constructed as follows:

for( i = 0; i < 2; i++ ) {  
 for( j = 0, k = 0, pocBase = PicOrderCntVal; j < num\_ref\_entries[ i ][ RplsIdx[ i ] ]; j++) {  
 if( st\_ref\_pic\_flag[ i ][ RplsIdx[ i ] ][ j ] ) {  
 RefPicPocList[ i ][ j ] = pocBase − DeltaPocSt[ i ][ RplsIdx[ i ] ][ j ]  
 if( there is a reference picture picA in the DPB with PicOrderCntVal equal to RefPicPocList[ i ][ j ] )  
 RefPicList[ i ][ j ] = picA  
 else  
 RefPicList[ i ][ j ] = "no reference picture" (8‑5)  
 pocBase = RefPicPocList[ i ][ j ]  
 } else {  
 if( !delta\_poc\_msb\_cycle\_lt[ i ][ k ] ) {  
 if( there is a reference picA in the DPB with PicOrderCntVal & ( MaxPicOrderCntLsb − 1 )  
 equal to poc\_lsb\_lt[ i ][ RplsIdx[ i ] ][ j ] )  
 RefPicList[ i ][ j ] = picA  
 else  
 RefPicList[ i ][ j ] = "no reference picture"  
 } else {  
 if( there is a reference picA in the DPB with PicOrderCntVal equal to  
 FullPocLt[ i ][ RplsIdx[ i ] ][ j ] )  
 RefPicList[ i ][ j ] = picA  
 else  
 RefPicList[ i ][ j ] = "no reference picture"  
 }  
 k++  
 }  
 }  
}

For each i equal to 0 or 1, the first NumRefIdxActive[ i ] entries in RefPicList[ i ] are referred to as the active entries in RefPicList[ i ], and the other entries in RefPicList[ i ] are referred to as the inactive entries in RefPicList[ i ].

NOTE 2 – It is possible that a particular picture is referred to by both an entry in RefPicList[ 0 ] and an entry in RefPicList[ 1 ]. It is also possible that a particular picture is referred to by more than one entry in RefPicList[ 0 ] or by more than one entry in RefPicList[ 1 ].

NOTE 3 – The active entries in RefPicList[ 0 ] and the active entries in RefPicList[ 1 ] collectively refer to all reference pictures that may be used for inter prediction of the current picture and one or more pictures that follow the current picture in decoding order. The inactive entries in RefPicList[ 0 ] and the inactive entries in RefPicList[ 1 ] collectively refer to all reference pictures that are *not* used for inter prediction of the current picture but may be used in inter prediction for one or more pictures that follow the current picture in decoding order.

NOTE 4 – There may be one or more entries in RefPicList[ 0 ] or RefPicList[ 1 ] that are equal to "no reference picture" because the corresponding pictures are not present in the DPB. Each inactive entry in RefPicList[ 0 ] or RefPicList[ 0 ] that is equal to "no reference picture" should be ignored. An unintentional picture loss should be inferred for each active entry in RefPicList[ 0 ] or RefPicList[ 1 ] that is equal to "no reference picture".

It is a requirement of bitstream conformance that the following constraints apply:

* For each i equal to 0 or 1, num\_ref\_entries[ i ][ RplsIdx[ i ] ] shall not be less than NumRefIdxActive[ i ].
* The picture referred to by each active entry in RefPicList[ 0 ] or RefPicList[ 1 ] shall be present in the DPB and shall have TemporalId less than or equal to that of the current picture.
* The picture referred to by each entry in RefPicList[ 0 ] or RefPicList[ 1 ] shall not be the current picture.
* An STRP entry in RefPicList[ 0 ] or RefPicList[ 1 ] of a tile group of a picture and an LTRP entry in RefPicList[ 0 ] or RefPicList[ 1 ] of the same tile group or a different tile group of the same picture shall not refer to the same picture.
* There shall be no LTRP entry in RefPicList[ 0 ] or RefPicList[ 1 ] for which the difference between the PicOrderCntVal of the current picture and the PicOrderCntVal of the picture referred to by the entry is greater than or equal to 224.
* Let setOfRefPics be the set of unique pictures referred to by all entries in RefPicList[ 0 ] and all entries in RefPicList[ 1 ]. The number of pictures in setOfRefPics shall be less than or equal to sps\_max\_dec\_pic\_buffering\_minus1 and setOfRefPics shall be the same for all tile groups of a picture.

For each current picture, let the variables maxPicOrderCnt and minPicOrderCnt be set equal to the maximum and the minimum, respectively, of the PicOrderCntVal values of the following pictures:

– The current picture.

– The previous picture in decoding order that has TemporalId equal to 0 and that is not a RASL or RADL picture. [Ed. (YK): Check this bullet item after the texts of RASL and RADL pictures are more mature.]

– The STRPs referred to by all entries in RefPicList[ 0 ] and all entries in RefPicList[ 1 ] of the current picture.

– All pictures n that have PicOutputFlag equal to 1, AuCpbRemovalTime[ n ] less than AuCpbRemovalTime[ currPic ] and DpbOutputTime[ n ] greater than or equal to AuCpbRemovalTime[ currPic ], where currPic is the current picture. [Ed. (YK): PicOutputFlag, AuCpbRemovalTime, and DpbOutputTime are to be defined.]

It is a requirement of bitstream conformance that for each current picture that is not an IRAP picture with NoRaslOutputFlag equal to 1, the value of maxPicOrderCnt − minPicOrderCnt shall be less than MaxPicOrderCntLsb / 2.

[Ed. (YK): Move the above two paragraphs to the bitstream conformance clause that is part of the HDR specification when available.]

### Decoding process for reference picture marking

This process is invoked once per picture, after decoding of a tile group header and the decoding process for reference picture list construction for the tile group as specified in clause 8.3.2, but prior to the decoding of the tile group data. This process may result in one or more reference pictures in the DPB being marked as "unused for reference" or "used for long-term reference".

A decoded picture in the DPB can be marked as "unused for reference", "used for short-term reference" or "used for long-term reference", but only one among these three at any given moment during the operation of the decoding process. Assigning one of these markings to a picture implicitly removes another of these markings when applicable. When a picture is referred to as being marked as "used for reference", this collectively refers to the picture being marked as "used for short-term reference" or "used for long-term reference" (but not both).

STRPs are identified by their PicOrderCntVal values. LTRPs are identified by the Log2( MaxLtPicOrderCntLsb ) LSBs of their PicOrderCntVal values.

If the current picture is an IRAP picture with NoRaslOutputFlag equal to 1, all reference pictures currently in the DPB (if any) are marked as "unused for reference".

Otherwise, the following applies:

* For each LTRP entry in RefPicList[ 0 ] or RefPicList[ 1 ], when the referred picture is an STRP, the picture is marked as "used for long-term reference".
* Each reference picture in the DPB that is not referred to by any entry in RefPicList[ 0 ] or RefPicList[ 1 ] is marked as "unused for reference".

### Decoding process for symmetric motion vector difference reference indices

Output of this process are RefIdxSymL0 and RefIdxSymL0 specifying the list 0 and list 1 reference picture indices for symmetric motion vector differences, i.e., when sym\_mvd\_flag is equal to 1 for a coding unit.

The variable RefIdxSymLX with X being 0 and 1 is derived as follows:

* The variable currPic specifies the current picture.
* RefIdxSymL0 is set equal to −1.
* For each index i with i = 0..NumRefIdxActive[ 0 ], the following applies:
* When all of the following conditions are true, RefIdxSymL0 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) > 0,
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) < DiffPicOrderCnt( currPic, RefPicList[ 0 ][ RefIdxSymL0 ] ) or RefIdxSymL0 is equal to −1.
* RefIdxSymL1 is set equal to −1.
* For each index i with i = 0..NumRefIdxActive[ 1 ], the following applies:
* When all of the following conditions are true, RefIdxSymL1 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) < 0,
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) > DiffPicOrderCnt( currPic, RefPicList[ 1 ][ RefIdxSymL1 ] ) or RefIdxSymL1 is equal to −1.
* When RefIdxSymL1 is equal to −1 or RefIdxSymL1 is equal to −1, the following applies:
* For each index i with i = 0..NumRefIdxActive[ 0 ], the following applies:
* When all of the following conditions are true, RefIdxSymL0 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) < 0,
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) > DiffPicOrderCnt( currPic, RefPicList[ 0 ][ RefIdxSymL0 ] ) or RefIdxSymL0 is equal to −1.
* For each index i with i = 0..NumRefIdxActive[ 1 ], the following applies:
* When all of the following conditions are true, RefIdxSymL1 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) > 0,
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) < DiffPicOrderCnt( currPic, RefPicList[ 1 ][ RefIdxSymL1 ] ) or RefIdxSymL1 is equal to −1.

## 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.

The derivation process for quantization parameters as specified in clause 8.7.1 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, and the variable treeType as inputs.

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

* If pcm\_flag[ xCb ][ yCb ] is equal to 1, the reconstructed picture is modified as follows:

SL[ xCb + i ][ yCb + j ] =   
 pcm\_sample\_luma[ ( cbHeight \* j ) + i ]  <<  ( BitDepthY − PcmBitDepthY ), (8‑6)  
 with i = 0..cbWidth − 1, j = 0..cbHeight − 1

* Otherwise, the following applies:

1. The derivation process for the luma intra prediction mode as specified in clause 8.4.2 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 8.4.4.1 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.

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

* If pcm\_flag[ xCb ][ yCb ] is equal to 1, the reconstructed picture is modified as follows:

SCb[ xCb / SubWidthC + i ][ yCb / SubHeightC + j ] =  
 pcm\_sample\_chroma[ ( cbHeight / SubWidthC \* j ) + i ]  <<  ( BitDepthC − PcmBitDepthC ),  
 with i = 0.. cbWidth / SubWidthC − 1 and j = 0.. cbHeight / SubHeightC − 1 (8‑7)

SCr[ xCb / SubWidthC + i ][ yCb / SubHeightC + j ] =  
 pcm\_sample\_chroma[ ( cbHeight / SubWidthC \* ( j + cbHeight / SubHeightC ) ) + i ]  <<  
 ( BitDepthC − PcmBitDepthC ),  
 with i = 0..cbWidth / SubWidthC − 1 and j = 0..cbHeight / SubHeightC − 1 (8‑8)

* Otherwise, the following applies:

1. The derivation process for the chroma intra prediction mode as specified in clause 8.4.3 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 8.4.4.1 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 8.4.4.1 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 |
| 81..83 | INTRA\_LT\_CCLM, INTRA\_L\_CCLM, INTRA\_T\_CCLM |

NOTE – : The intra prediction modes INTRA\_LT\_CCLM, INTRA\_L\_CCLM and INTRA\_T\_CCLM are 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 + cbHeight − 1 ) and ( xCb + cbWidth − 1, 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\_PLANAR.
* The variable availableX is equal to FALSE.
* CuPredMode[ xNbX ][ yNbX ] is not equal to MODE\_INTRA and ciip\_flag[ xNbX ][ yNbX ] is not equal to 1.
* pcm\_flag[ xNbX ][ yNbX ] is equal to 1.
* 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 variables ispDefaultMode1 and ispDefaultMode2 are defined as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_HOR\_SPLIT, ispDefaultMode1 is set equal to INTRA\_ANGULAR18 and ispDefaultMode2 is set equal to INTRA\_ANGULAR5.
* Otherwise, ispDefaultMode1 is set equal to INTRA\_ANGULAR50 and ispDefaultMode2 is set equal to INTRA\_ANGULAR63.

1. The candModeList[ x ] with x = 0..5 is derived as follows:

* If candIntraPredModeB is equal to candIntraPredModeA and candIntraPredModeA is greater than INTRA\_DC, candModeList[ x ] with x = 0..5 is derived as follows:
  + If IntraLumaRefLineIdx[ xCb ][ yCb ] is equal to 0 and IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, the following applies:

candModeList[ 0 ] = candIntraPredModeA (8‑9)

candModeList[ 1 ] = INTRA\_PLANAR (8‑10)

candModeList[ 2 ] = INTRA\_DC (8‑11)

candModeList[ 3 ] = 2 + ( ( candIntraPredModeA + 61 ) % 64 ) (8‑12)

candModeList[ 4 ] = 2 + ( ( candIntraPredModeA − 1 ) % 64 ) (8‑13)

candModeList[ 5 ] = 2 + ( ( candIntraPredModeA + 60 ) % 64 ) (8‑14)

* + Otherwise (IntraLumaRefLineIdx[ xCb ][ yCb ] is not equal to 0 or IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT), the following applies:

candModeList[ 0 ] = candIntraPredModeA (8‑15)

candModeList[ 1 ] = 2 + ( ( candIntraPredModeA + 61 ) % 64 ) (8‑16)

candModeList[ 2 ] = 2 + ( ( candIntraPredModeA − 1 ) % 64 ) (8‑17)

* + If one of the following conditions is true,
  + IntraSubPartitionsSplitType is equal to ISP\_HOR\_SPLIT and candIntraPredModeA  is less than INTRA\_ANGULAR34,
  + IntraSubPartitionsSplitType is equal to ISP\_VER\_SPLIT and candIntraPredModeA  is greater than or equal to INTRA\_ANGULAR34,
  + IntraLumaRefLineIdx[ xCb ][ yCb ] is not equal to 0,

the following applies:

candModeList[ 3 ] = 2 + ( ( candIntraPredModeA + 60 ) % 64 ) (8‑18)

candModeList[ 4 ] = 2 + ( candIntraPredModeA % 64 ) (8‑19)

candModeList[ 5 ] = 2 + ( ( candIntraPredModeA + 59 ) % 64 ) (8‑20)

* + Otherwise, the following applies:

candModeList[ 3 ] = ispDefaultMode1 (8‑21)

candModeList[ 4 ] = ispDefaultMode2 (8‑22)

candModeList[ 5 ] = INTRA\_PLANAR (8‑23)

* Otherwise if candIntraPredModeB is not equal to candIntraPredModeA and candIntraPredModeA or candIntraPredModeB is greater than INTRA\_DC, the following applies:
  + The variables minAB and maxAB are derived as follows:

minAB = Min( candIntraPredModeA, candIntraPredModeB ) (8‑24)

maxAB = Max( candIntraPredModeA, candIntraPredModeB ) (8‑25)

* + If candIntraPredModeA and candIntraPredModeB are both greater than INTRA\_DC, candModeList[ x ] with x = 0..5 is derived as follows:

candModeList[ 0 ] = candIntraPredModeA (8‑26)

candModeList[ 1 ] = candIntraPredModeB (8‑27)

* + If IntraLumaRefLineIdx[ xCb ][ yCb ] is equal to 0 and IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, the following applies:

candModeList[ 2 ] = INTRA\_PLANAR (8‑28)

candModeList[ 3 ] = INTRA\_DC (8‑29)

* + If maxAB − minAB is in the range of 2 to 62, inclusive, the following applies:

candModeList[ 4 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑30)

candModeList[ 5 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑31)

* + Otherwise, the following applies:

candModeList[ 4 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑32)

candModeList[ 5 ] = 2 + ( ( maxAB ) % 64 ) (8‑33)

* + Otherwise (IntraLumaRefLineIdx[ xCb ][ yCb ] is not equal to 0 or IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT), the following applies:
  + When IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT, and abs( candIntraPredModeB − ispDefaultMode1 ) is less than abs( candIntraPredModeA − ispDefaultMode1 ), the following applies:

candModeList[ 0 ] = candIntraPredModeB (8‑34)

candModeList[ 1 ] = candIntraPredModeA (8‑35)

* + If maxAB − minAB is equal to 1, the following applies:

candModeList[ 2 ] = 2 + ( ( minAB + 61 ) % 64 ) (8‑36)

candModeList[ 3 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑37)

candModeList[ 4 ] = 2 + ( ( minAB + 60 ) % 64 ) (8‑38)

candModeList[ 5 ] = 2 + ( maxAB % 64 ) (8‑39)

* + Otherwise if maxAB − minAB is equal to 2, the following applies:

candModeList[ 2 ] = 2 + ( ( minAB − 1 ) % 64 ) (8‑40)

candModeList[ 3 ] = 2 + ( ( minAB + 61 ) % 64 ) (8‑41)

candModeList[ 4 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑42)

candModeList[ 5 ] = 2 + ( ( minAB + 60 ) % 64 ) (8‑43)

* + Otherwise if maxAB − minAB is greater than 61, the following applies:

candModeList[ 2 ] = 2 + ( ( minAB − 1 ) % 64 ) (8‑44)

candModeList[ 3 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑45)

candModeList[ 4 ] = 2 + ( minAB % 64 ) (8‑46)

candModeList[ 5 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑47)

* + Otherwise, the following applies:

candModeList[ 2 ] = 2 + ( ( minAB + 61 ) % 64 ) (8‑48)

candModeList[ 3 ] = 2 + ( ( minAB − 1 ) % 64 ) (8‑49)

candModeList[ 4 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑50)

candModeList[ 5 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑51)

* + Otherwise (candIntraPredModeA or candIntraPredModeB is greater than INTRA\_DC), candModeList[ x ] with x = 0..5 is derived as follows:
  + If IntraLumaRefLineIdx[ xCb ][ yCb ] is equal to 0 and IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, the following applies:

candModeList[ 0 ] = candIntraPredModeA (8‑52)

candModeList[ 1 ] = candIntraPredModeB (8‑53)

candModeList[ 2 ] = 1 − minAB (8‑54)

candModeList[ 3 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑55)

candModeList[ 4 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑56)

candModeList[ 5 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑57)

* + Otherwise, if IntraLumaRefLineIdx[ xCb ][ yCb ] is not equal to 0, the following applies:

candModeList[ 0 ] = maxAB (8‑58)

candModeList[ 1 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑59)

candModeList[ 2 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑60)

candModeList[ 3 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑61)

candModeList[ 4 ] = 2 + ( maxAB % 64 ) (8‑62)

candModeList[ 5 ] = 2 + ( ( maxAB + 59 ) % 64 ) (8‑63)

* + Otherwise (IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT), the following applies:

candModeList[ 0 ] = INTRA\_PLANAR (8‑64)

candModeList[ 1 ] = maxAB (8‑65)

candModeList[ 2 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑66)

candModeList[ 3 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑67)

candModeList[ 4 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑68)

candModeList[ 5 ] = 2 + ( maxAB % 64 ) (8‑69)

* Otherwise, the following applies:
  + If IntraLumaRefLineIdx[ xCb ][ yCb ] is equal to 0 and IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, the following applies:

candModeList[ 0 ] = candIntraPredModeA (8‑70)

candModeList[ 1 ] = ( candModeList[0]  = =  INTRA\_PLANAR )  ?  INTRA\_DC  :   (8‑71)   
 INTRA\_PLANAR

candModeList[ 2 ] = INTRA\_ANGULAR50 (8‑72)

candModeList[ 3 ] = INTRA\_ANGULAR18 (8‑73)

candModeList[ 4 ] = INTRA\_ANGULAR46 (8‑74)

candModeList[ 5 ] = INTRA\_ANGULAR54 (8‑75)

* + Otherwise, if IntraLumaRefLineIdx[ xCb ][ yCb ] is not equal to 0, the following applies:

candModeList[ 0 ] = INTRA\_ANGULAR50 (8‑76)

candModeList[ 1 ] = INTRA\_ANGULAR18 (8‑77)

candModeList[ 2 ] = INTRA\_ANGULAR2 (8‑78)

candModeList[ 3 ] = INTRA\_ANGULAR34 (8‑79)

candModeList[ 4 ] = INTRA\_ANGULAR66 (8‑80)

candModeList[ 5 ] = INTRA\_ANGULAR26 (8‑81)

* + Otherwise, if IntraSubPartitionsSplitType is equal to ISP\_HOR\_SPLIT, the following applies:

candModeList[ 0 ] = INTRA\_PLANAR (8‑82)

candModeList[ 1 ] = INTRA\_ANGULAR18 (8‑83)

candModeList[ 2 ] = INTRA\_ANGULAR25 (8‑84)

candModeList[ 3 ] = INTRA\_ANGULAR10 (8‑85)

candModeList[ 4 ] = INTRA\_ANGULAR65 (8‑86)

candModeList[ 5 ] = INTRA\_ANGULAR50 (8‑87)

* + Otherwise, if IntraSubPartitionsSplitType is equal to ISP\_VER\_SPLIT, the following applies:

candModeList[ 0 ] = INTRA\_PLANAR (8‑88)

candModeList[ 1 ] = INTRA\_ANGULAR50 (8‑89)

candModeList[ 2 ] = INTRA\_ANGULAR43 (8‑90)

candModeList[ 3 ] = INTRA\_ANGULAR60 (8‑91)

candModeList[ 4 ] = INTRA\_ANGULAR3 (8‑92)

candModeList[ 5 ] = INTRA\_ANGULAR18 (8‑93)

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. When candModeList[ i ] is greater than candModeList[ j ] for i = 0..4 and for each i, j = ( i + 1 )..5, both values are swapped as follows:

( candModeList[ i ], candModeList[ j ] ) = Swap( candModeList[ i ], candModeList[ j ] ) (8‑94)

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 5, 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 are:

* 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,
* 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 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 + cbWidth / 2 ][ yCb + cbHeight / 2 ] 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 + cbWidth / 2 ][ yCb + cbHeight / 2 ] when sps\_cclm\_enabled\_flag is equal to 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| intra\_chroma\_pred\_mode[ xCb ][ yCb ] | IntraPredModeY[ xCb + cbWidth / 2 ][ yCb + cbHeight / 2 ] | | | | |
| 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 + cbWidth / 2 ][ yCb + cbHeight / 2 ] when sps\_cclm\_enabled\_flag is equal to 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| intra\_chroma\_pred\_mode[ xCb ][ yCb ] | IntraPredModeY[ xCb + cbWidth / 2 ][ yCb + cbHeight / 2 ] | | | | |
| 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 | 81 | 81 | 81 | 81 | 81 |
| 5 | 82 | 82 | 82 | 82 | 82 |
| 6 | 83 | 83 | 83 | 83 | 83 |
| 7 | 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 (8‑95)

The luma sample location is derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTb0, yTb0 ) : ( xTb0 \* 2, yTb0 \* 2 ) (8‑96)

Depending on maxTbSize, the following applies:

* If IntraSubPartSplitType is equal to NO\_ISP\_SPLIT and nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables newTbW and newTbH are derived as follows:

newTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑97)

newTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑98)

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 set equal to newTbW and the height nTbH set equal to newTbH, 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 + newTbW, yTb0 ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, 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 + newTbH ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, 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 + newTbW, yTb0 + newTbH ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, 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:
* The variables nW, nH, numPartsX and numPartsY are derived as follows:

nW = IntraSubPartitionsSplitType = = ISP\_VER\_SPLIT ? nTbW / NumIntraSubPartitions : nTbW (8‑99)

nH = IntraSubPartitionsSplitType = = ISP\_HOR\_SPLIT ? nTbH / NumIntraSubPartitions : nTbH (8‑100)

numPartsX = IntraSubPartitionsSplitType = = ISP\_VER\_SPLIT ? NumIntraSubPartitions : 1 (8‑101)

numPartsY = IntraSubPartitionsSplitType = = ISP\_HOR\_SPLIT ? NumIntraSubPartitions : 1 (8‑102)

* For xPartIdx = 0..numPartsX − 1 and yPartIdx = 0..numPartsY − 1, the following applies:

1. The general intra sample prediction process as specified in clause 8.4.4.2.1 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xTb0 + nW \* xPartIdx, yTb0 + nH \* yPartIdx ), the intra prediction mode predModeIntra, the transform block width nTbW and height nTbH set equal to nW and nH, the coding block width nCbW and height nCbH set equal to nTbW and 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 8.7.2 is invoked with the luma location ( xTbY, yTbY ) set equal to ( xTbY + nW \* xPartIdx, yTbY + nH \* yPartIdx ), the variable cIdx, the transform width nTbW and the transform height nTbH set equal to nW and nH 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 8.7.5 is invoked with the transform block location ( xTbComp, yTbComp ) set equal to ( xTb0 + nW \* xPartIdx, yTb0 + nH \* yPartIdx ), the transform block width nTbW, the transform block height nTbH set equal to nW and nH, 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 nCbW specifying the coding block width,
* a variable nCbH specifying the coding 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 variables refW and refH are derived as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0, the following applies:

refW = nTbW \* 2 (8‑103)

refH = nTbH \* 2 (8‑104)

* Otherwise ( IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 ), the following applies:

refW = nCbW \* 2 (8‑105)

refH = nCbH \* 2 (8‑106)

The variable refIdx specifying the intra prediction reference line index is derived as follows:

refIdx = ( cIdx  = =  0 )  ?  IntraLumaRefLineIdx[ xTbCmp ][ yTbCmp ]  :  0 (8‑107)

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

1. The reference sample availability marking process as specified in clause 8.4.4.2.2 is invoked with the sample location ( xTbCmp, yTbCmp ), the intra prediction reference line index refIdx, 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 − refIdx, y = −1 − refIdx..refH − 1 and x = − refIdx..refW − 1, y = −1 − refIdx as output.
2. When at least one sample refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx is marked as "not available for intra prediction", the reference sample substitution process as specified in clause 8.4.4.2.3 is invoked with the intra prediction reference line index refIdx, the reference sample width refW, the reference sample height refH, the reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, and the colour component index cIdx as inputs, and the modified reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx as output.
3. The reference sample filtering process as specified in clause 8.4.4.2.4 is invoked with the intra prediction reference line index refIdx, 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, and the colour component index cIdx as inputs, and the reference samples p[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx 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 8.4.4.2.5 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 8.4.4.2.6 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\_LT\_CCLM, INTRA\_L\_CCLM or INTRA\_T\_CCLM, the corresponding intra prediction mode process specified in clause 8.4.4.2.8 is invoked with the intra prediction mode predModeIntra, 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 8.4.4.2.7 is invoked with the intra prediction mode predModeIntra, the intra prediction reference line index refIdx, the transform block width nTbW, the transform block height nTbH, the reference sample width refW, the reference sample height refH, the coding block width nCbW and height nCbH, the colour component index cIdx, and the reference sample array p as inputs, and the modified intra prediction mode predModeIntra and the predicted sample array predSamples as outputs.

When all of the following conditions are true, the position-dependent prediction sample filtering process specified in clause 8.4.4.2.9 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:

* IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0
* refIdx is equal to 0 or cIdx is not equal to 0
* One of the following conditions is true:
* 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 refIdx specifying the intra prediction reference line index,
* 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction.

The refW + refH + 1 + ( 2 \* refIdx ) neighbouring samples refUnfilt[ x ][ y ] that are constructed samples prior to the in-loop filter process, with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, are derived as follows:

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

( xNbCmp, yNbCmp ) = ( xTbCmp + x, yTbCmp + y ) (8‑108)

* 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 ) (8‑109)

( xNbY, yNbY ) = ( cIdx  = =  0 ) ? ( xNbCmp, yNbCmp ) : ( xNbCmp << 1, yNbCmp << 1 ) (8‑110)

* 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 refIdx specifying the intra prediction reference line index,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are modified as follows:

* If all samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx 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 − refIdx ][ refH − 1 ] is marked as "not available for intra prediction", search sequentially starting from x = −1 − refIdx, y = refH − 1 to x = −1 − refIdx, y = −1 − refIdx, then from x = −refIdx, y = −1 − refIdx to x = refW − 1, y = −1 − refIdx, 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 − refIdx ][ refH − 1 ] is set equal to the value of refUnfilt[ x ][ y ].
2. For x = −1 − refIdx, y = refH − 2..−1 − refIdx, 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are marked as "available for intra prediction".

##### Reference sample filtering process

Inputs to this process are:

* a variable refIdx specifying the intra prediction reference line index,
* 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx,
* 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 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx.

The variable filterFlag is derived as follows:

* If all of the following conditions are true, filterFlag is set equal to 1:
* refIdx is equal to 0
* nTbW \* nTbH is greater than 32
* cIdx is equal to 0
* IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT
* one or more of the following conditions is true:
* predModeIntra is equal to INTRA\_PLANAR
* predModeIntra is equal to INTRA\_ANGULAR34
* predModeIntra is equal to INTRA\_ANGULAR2 and nTbH is greater than or equal to nTbW
* predModeIntra is equal to INTRA\_ANGULAR66 and nTbW is greater than or equal to nTbH
* Otherwise, filterFlag is set equal to 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 (8‑111)

p[ −1 ][ y ] = ( refUnfilt[ −1 ][ y + 1 ] + 2 \* refUnfilt[ −1 ][ y ] + refUnfilt[ −1 ][ y − 1 ] + 2 )  >>  2  
 for y = 0..refH − 2 (8‑112)

p[ −1 ][ refH − 1 ] = refUnfilt[ −1 ][ refH − 1 ] (8‑113)

p[ x ][ −1 ] = ( refUnfilt[ x − 1 ][ −1 ] + 2 \* refUnfilt[ x ][ −1 ] + refUnfilt[ x + 1 ][ −1 ] + 2 )  >>  2  
 for x = 0..refW − 2  (8‑114)

p[ refW − 1 ][ −1 ] = refUnfilt[ refW − 1 ][ −1 ] (8‑115)

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

##### 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 variables nW and nH are derived as follows:

nW = Max( nTbW, 2 ) (8‑116)

nH = Max( nTbH, 2 ) (8‑117)

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 ] = ( ( nH − 1 − y ) \* p[ x ][ −1 ] + ( y + 1 ) \* p[ −1 ][ nTbH ] ) << Log2 ( nW )  (8‑118)

predH[ x ][ y ] = ( ( nW − 1 − x ) \* p[ −1 ][ y ] + ( x + 1 ) \* p[ nTbW ][ −1 ] ) << Log2 ( nH )  (8‑119)

predSamples[ x ][ y ] = ( predV[ x ][ y ] + predH[ x ][ y ] + nW \* nH )  >>  (Log2 ( nW ) + Log2 ( nH ) + 1 ) (8‑120)

##### 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 (8‑121)

* When nTbW is greater than nTbH:

dcVal (8‑122)

* When nTbW is less than nTbH:

dcVal (8‑123)

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

predSamples[ x ][ y ] = dcVal, with x = 0.. nTbW − 1, y = 0.. nTbH − 1 (8‑124)

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

Inputs to this process are:

* the intra prediction mode predModeIntra,
* a variable refIdx specifying the intra prediction reference line index,
* 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,
* a variable nCbW specifying the coding block width,
* a variable nCbH specifying the coding block height,
* a variable cIdx specifying the colour component of the current block,
* the neighbouring samples p[ x ][ y ], with x = −1− refIdx, y = −1− refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1− refIdx.

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

The variables nW and nH are derived as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0, the following applies:

nW = nTbW (8‑125)

nH = nTbH (8‑126)

* Otherwise ( IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 ), the following applies:

nW = nCbW (8‑127)

nH = nCbH (8‑128)

The variable whRatio is set equal to Abs( Log2( nW / nH ) ).

The variable wideAngle is set equal to 0.

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

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

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\_ANGULAR2, INTRA\_ANGULAR34 or INTRA\_ANGULAR66
* refIdx is not equal to 0
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 and predModeIntra is greater than or equal to INTRA\_ANGULAR34 and nW is greater than 8
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 and predModeIntra is less than INTRA\_ANGULAR34 and nH is greater than 8.
* 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **nTbS = 2** | **nTbS = 3** | **nTbS = 4** | **nTbS = 5** | **nTbS = 6** | **nTbS = 7** |
| **intraHorVerDistThres[ nTbS ]** | 16 | 14 | 2 | 0 | 0 | 0 |

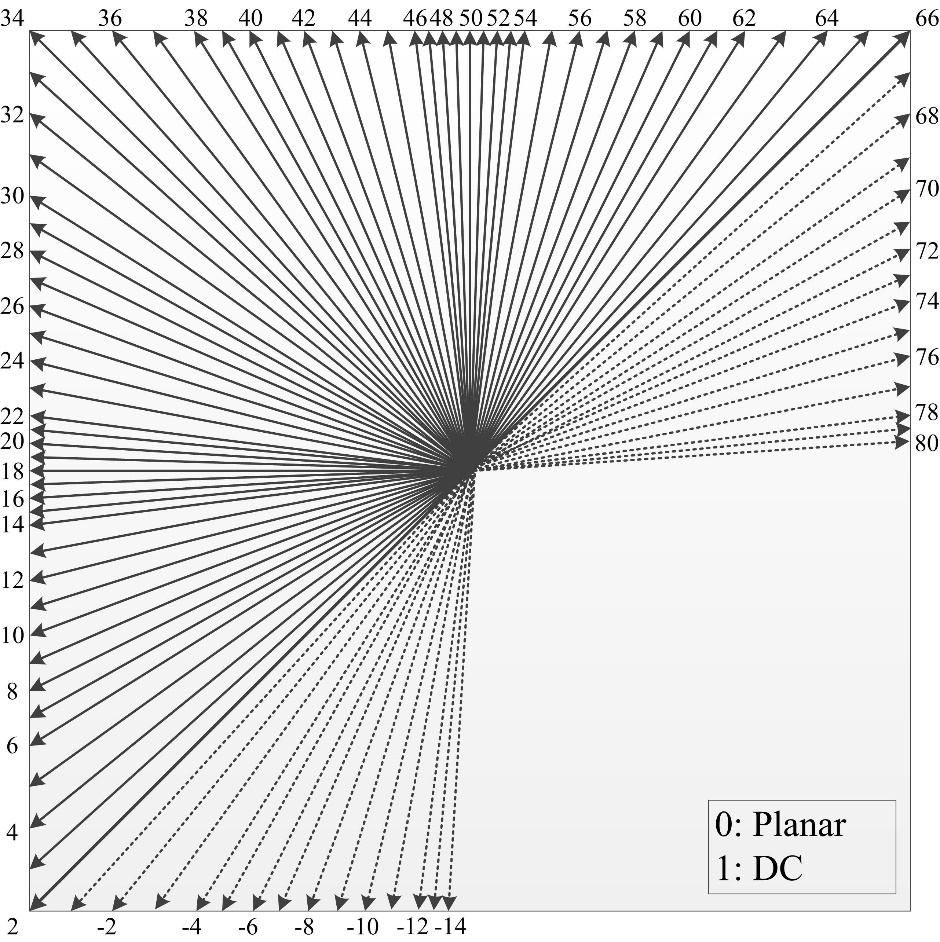


Figure 8‑1 – Intra prediction directions (informative)

Figure 8‑1 illustrates the 93 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** | **−14** | **−13** | **−12** | **−11** | **−10** | **−9** | **−8** | **−7** | **−6** | **−5** | **−4** | **−3** | **−2** | **−1** | **2** | **3** | **4** |
| **intraPredAngle** | 512 | 341 | 256 | 171 | 128 | 102 | 86 | 73 | 64 | 57 | 51 | 45 | 39 | 35 | 32 | 29 | 26 |
| **predModeIntra** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** |
| **intraPredAngle** | 23 | 20 | 18 | 16 | 14 | 12 | 10 | 8 | 6 | 4 | 3 | 2 | 1 | 0 | −1 | −2 | −3 |
| **predModeIntra** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** |
| **intraPredAngle** | −4 | −6 | −8 | −10 | −12 | −14 | −16 | −18 | −20 | −23 | −26 | −29 | −32 | −29 | −26 | −23 | −20 |
| **predModeIntra** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** |
| **intraPredAngle** | −18 | −16 | −14 | −12 | −10 | −8 | −6 | −4 | −3 | −2 | −1 | 0 | 1 | 2 | 3 | 4 | 6 |
| **predModeIntra** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** |
| **intraPredAngle** | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 23 | 26 | 29 | 32 | 35 | 39 | 45 | 51 | 57 | 64 |
| **predModeIntra** | **73** | **74** | **75** | **76** | **77** | **78** | **79** | **80** |  |  |  |  |  |  |  |  |  |
| **intraPredAngle** | 73 | 86 | 102 | 128 | 171 | 256 | 341 | 512 |  |  |  |  |  |  |  |  |  |

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

invAngle = Round (8‑129)

The interpolation filter coefficients fC[ phase ][ j ] and fG[ phase ][ j ] with phase = 0..31 and j = 0..3 are specified in Table 8‑6.

Table 8‑6 – Specification of interpolation filter coefficients fC and fG

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Fractional sample position p** | **fC interpolation filter coefficients** | | | | **fG interpolation filter coefficients** | | | |
| **fC[ p ][ 0 ]** | **fC[ p ][ 1 ]** | **fC[ p ][ 2 ]** | **fC[ p ][ 3 ]** | **fG[ p ][ 0 ]** | **fG[ p ][ 1 ]** | **fG[ p ][ 2 ]** | **fG[ p ][ 3 ]** |
| 0 | 0 | 64 | 0 | 0 | 16 | 32 | 16 | 0 |
| 1 | −1 | 63 | 2 | 0 | 15 | 29 | 17 | 3 |
| 2 | −2 | 62 | 4 | 0 | 15 | 29 | 17 | 3 |
| 3 | −2 | 60 | 7 | −1 | 14 | 29 | 18 | 3 |
| 4 | −2 | 58 | 10 | −2 | 13 | 29 | 18 | 4 |
| 5 | −3 | 57 | 12 | −2 | 13 | 28 | 19 | 4 |
| 6 | −4 | 56 | 14 | −2 | 13 | 28 | 19 | 4 |
| 7 | −4 | 55 | 15 | −2 | 12 | 28 | 20 | 4 |
| 8 | −4 | 54 | 16 | −2 | 11 | 28 | 20 | 5 |
| 9 | −5 | 53 | 18 | −2 | 11 | 27 | 21 | 5 |
| 10 | −6 | 52 | 20 | −2 | 10 | 27 | 22 | 5 |
| 11 | −6 | 49 | 24 | −3 | 9 | 27 | 22 | 6 |
| 12 | −6 | 46 | 28 | −4 | 9 | 26 | 23 | 6 |
| 13 | −5 | 44 | 29 | −4 | 9 | 26 | 23 | 6 |
| 14 | −4 | 42 | 30 | −4 | 8 | 25 | 24 | 7 |
| 15 | −4 | 39 | 33 | −4 | 8 | 25 | 24 | 7 |
| 16 | −4 | 36 | 36 | −4 | 8 | 24 | 24 | 8 |
| 17 | −4 | 33 | 39 | −4 | 7 | 24 | 25 | 8 |
| 18 | −4 | 30 | 42 | −4 | 7 | 24 | 25 | 8 |
| 19 | −4 | 29 | 44 | −5 | 6 | 23 | 26 | 9 |
| 20 | −4 | 28 | 46 | −6 | 6 | 23 | 26 | 9 |
| 21 | −3 | 24 | 49 | −6 | 6 | 22 | 27 | 9 |
| 22 | −2 | 20 | 52 | −6 | 5 | 22 | 27 | 10 |
| 23 | −2 | 18 | 53 | −5 | 5 | 21 | 27 | 11 |
| 24 | −2 | 16 | 54 | −4 | 5 | 20 | 28 | 11 |
| 25 | −2 | 15 | 55 | −4 | 4 | 20 | 28 | 12 |
| 26 | −2 | 14 | 56 | −4 | 4 | 19 | 28 | 13 |
| 27 | −2 | 12 | 57 | −3 | 4 | 19 | 28 | 13 |
| 28 | −2 | 10 | 58 | −2 | 4 | 18 | 29 | 13 |
| 29 | −1 | 7 | 60 | −2 | 3 | 18 | 29 | 14 |
| 30 | 0 | 4 | 62 | −2 | 3 | 17 | 29 | 15 |
| 31 | 0 | 2 | 63 | −1 | 3 | 17 | 29 | 15 |

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 − refIdx + x ][ −1 − refIdx ], with x = 0..nTbW + refIdx (8‑130)

* 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 − refIdx ][ −1 − refIdx + ( ( x \* invAngle + 128 )  >>  8 ) ],  
 with x = −1..( nTbH \* intraPredAngle )  >>  5 (8‑131)

ref[ ( ( nTbH \* intraPredAngle )  >>  5 ) − 1 ] = ref[ ( nTbH \* intraPredAngle )  >>  5 ] (8‑132)

ref[ nTbW + 1 + refIdx ] = ref[ nTbW + refIdx ] (8‑133)

* Otherwise,

ref[ x ] = p[ −1 − refIdx + x ][ −1 − refIdx ], with x = nTbW + 1 + refIdx..refW + refIdx (8‑134)

ref[ −1 ] = ref[ 0 ] (8‑135)

* The additional samples ref[ refW + refIdx +x ] with x = 1..( Max( 1, nTbW / nTbH ) \* refIdx + 1) are derived as follows:

ref[ refW + refIdx +x ] = p[ −1 + refW ][ −1 − refIdx ] (8‑136)

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 + refIdx ) \* intraPredAngle )  >>  5 + refIdx (8‑137)

iFact = ( ( y + 1 + refIdx ) \* intraPredAngle ) & 31 (8‑138)

* If cIdx is equal to 0, the following applies:
* The interpolation filter coefficients fT[ j ] with j = 0..3 are derived as follows:

fT[ j ] = filterFlag  ?  fG[ iFact ][ j ]  :  fC[ iFact ][ j ] (8‑139)

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

predSamples[ x ][ y ] = Clip1Y( ( (  ) + 32 )  >>  6 ) (8‑140)

* Otherwise (cIdx is not equal to 0), 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 (8‑141)

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

predSamples[ x ][ y ] = ref[ x + iIdx + 1 ] (8‑142)

* 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 − refIdx ][ −1 − refIdx + x ], with x = 0..nTbH + refIdx (8‑143)

* 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 − refIdx + ( ( x \* invAngle + 128 )  >>  8 ) ][ −1 − refIdx ],  
 with x = −1..( nTbW \* intraPredAngle )  >>  5 (8‑144)

ref[ ( ( nTbW \* intraPredAngle )  >>  5 ) − 1 ] = ref[ ( nTbW \* intraPredAngle )  >>  5 ] (8‑145)

ref[ nTbG + 1 + refIdx ] = ref[ nTbH + refIdx ] (8‑146)

* Otherwise,

ref[ x ] = p[ −1 − refIdx ][ −1 − refIdx + x ], with x = nTbH + 1 + refIdx..refH + refIdx (8‑147)

ref[ −1 ] = ref[ 0 ] (8‑148)

* The additional samples ref[ refH + refIdx +x ] with x = 1..( Max( 1, nTbW / nTbH ) \* refIdx + 1) are derived as follows:

ref[ refH + refIdx +x ] = p[ −1 + refH ][ −1 − refIdx ] (8‑149)

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 + refIdx ) \* intraPredAngle )  >>  5 (8‑150)

iFact = ( ( x + 1 + refIdx ) \* intraPredAngle ) & 31 (8‑151)

* If cIdx is equal to 0, the following applies:
* The interpolation filter coefficients fT[ j ] with j = 0..3 are derived as follows:

fT[ j ] = filterFlag  ?  fG[ iFact ][ j ]  :  fC[ iFact ][ j ] (8‑152)

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

predSamples[ x ][ y ] = Clip1Y( ( (  ) + 32 )  >>  6 ) (8‑153)

* Otherwise (cIdx is not equal to 0), 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 (8‑154)

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

predSamples[ x ][ y ] = ref[ y + iIdx + 1 ] (8‑155)

##### Specification of INTRA\_LT\_CCLM, INTRA\_L\_CCLM and INTRA\_T\_CCLM intra prediction mode

Inputs to this process are:

* the intra prediction mode predModeIntra,
* 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..2 \* nTbH − 1 and x = 0.. 2 \* 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 ) (8‑156)

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 chroma location ( xCurr, yCurr ) set equal to ( xTbC, yTbC ) and the neighbouring chroma location ( xTbC − 1, yTbC ) 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 chroma location ( xCurr, yCurr ) set equal to ( xTbC, yTbC ) and the neighbouring chroma location ( xTbC, yTbC − 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 chroma location ( xCurr, yCurr ) set equal to ( xTbC, yTbC ) and the neighbouring chroma location ( xTbC − 1, yTbC − 1 ) as inputs, and the output is assigned to availTL.
* The number of available top-right neighbouring chroma samples numTopRight is derived as follows:
* The variable numTopRight is set equal to 0 and availTR is set equal to TRUE.
* When predModeIntra is equal to INTRA\_T\_CCLM, the following applies for x = nTbW..2 \* nTbW − 1 until availTR is equal to FALSE or x is equal to 2 \* nTbW − 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 chroma location ( xCurr, yCurr ) set equal to ( xTbC , yTbC ) and the neighbouring chroma location ( xTbC + x, yTbC − 1 ) as inputs, and the output is assigned to availableTR
* When availableTR is equal to TRUE, numTopRight is incremented by one.
* The number of available left-below neighbouring chroma samples numLeftBelow is derived as follows:
* The variable numLeftBelow is set equal to 0 and availLB is set equal to TRUE.
* When predModeIntra is equal to INTRA\_L\_CCLM, the following applies for y = nTbH..2 \* nTbH − 1 until availLB is equal to FALSE or y is equal to 2 \* nTbH − 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 chroma location ( xCurr, yCurr ) set equal to ( xTbC , yTbC ) and the neighbouring chroma location ( xTbC − 1, yTbC + y ) as inputs, and the output is assigned to availableLB
* When availableLB is equal to TRUE, numLeftBelow is incremented by one.

The number of available neighbouring chroma samples on the top and top-right numTopSamp and the number of available neighbouring chroma samples on the left and left-below nLeftSamp are derived as follows:

* If predModeIntra is equal to INTRA\_LT\_CCLM, the following applies:

numSampT = availT  ?  nTbW  :  0 (8‑157)

numSampL = availL  ?  nTbH  :  0 (8‑158)

* Otherwise, the following applies:

numSampT = ( availT  &&  predModeIntra  = =  INTRA\_T\_CCLM )  ?  ( nTbW + numTopRight )  :  0 (8‑159)

numSampL = ( availL  &&  predModeIntra = =  INTRA\_L\_CCLM )  ?  ( nTbH + numLeftBelow )  :  0 (8‑160)

The variable bCTUboundary is derived as follows:

bCTUboundary = ( yTbC & ( 1 << ( CtbLog2SizeY − 1 ) − 1 )  = =  0 )  ?  TRUE  :  FALSE. (8‑161)

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

* If both numSampL and numSampT are equal to 0, the following applies:

predSamples[ x ][ y ] = 1 << ( BitDepthC − 1 ) (8‑162)

* 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 numSampL is greater than 0, the neighbouring left luma samples pY[ x ][ y ] with x = −1..−3, y = 0..2 \* numSampL − 1, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY + x , yTbY +y ).
     + When numSampT is greater than 0, the neighbouring top luma samples pY[ x ][ y ] with x = 0..2 \* numSampT − 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:
     + If sps\_cclm\_colocated\_chroma\_flag is equal to 1, the following applies:
     + pDsY[ x ][ y ] with x = 1..nTbW − 1, y = 1..nTbH − 1 is derived as follows:

pDsY[ x ][ y ] = ( pY[ 2 \* x ][ 2 \* y − 1 ] +  
  pY[ 2 \* x − 1 ][ 2 \* y ] + 4 \* pY[ 2 \* x ][ 2 \* y ] + pY[ 2 \* x + 1 ][ 2 \* y ] +  (8‑163)  
 pY[ 2 \* x ][ 2 \* y + 1 ] + 4 ) >> 3

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

pDsY[ 0 ][ y ] = ( pY[ 0 ][ 2 \* y − 1 ] +  
  pY[ −1 ][ 2 \* y ] + 4 \* pY[ 0 ][ 2 \* y ] + pY[ 1 ][ 2 \* y ] +  (8‑164)  
 pY[ 0 ][ 2 \* y + 1 ] + 4 ) >> 3

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

pDsY[ 0 ][ y ] = ( pY[ 0 ][ 2 \* y − 1 ] + 2 \* pY[ 0 ][ 2 \* y ] + pY[ 0 ][ 2 \* y + 1 ] + 2 ) >> 2 (8‑165)

* + - If availT is equal to TRUE, pDsY[ x ][ 0 ] with x = 1..nTbW − 1 is derived as follows:

pDsY[ x ][ 0 ] = ( pY[ 2 \* x ][ −1 ] +  
  pY[ 2 \* x − 1 ][ 0 ] + 4 \* pY[ 2 \* x ][ 0 ] + pY[ 2 \* x + 1 ][ 0 ] +  (8‑166)  
 pY[ 2 \* x ][ 1 ] + 4 ) >> 3

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

pDsY[ x ][ 0 ] = ( pY[ 2 \* x − 1 ][ 0 ] + 2 \* pY[ 2 \* x ][ 0 ] + pY[ 2 \* x + 1 ][ 0 ] + 2 ) >> 2 (8‑167)

* + - If availL is equal to TRUE and availT is equal to TRUE, pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = ( pY[ 0 ][ −1 ] +  
  pY[ −1 ][ 0 ] + 4 \* pY[ 0 ][ 0 ] + pY[ 1 ][ 0 ] +  (8‑168)  
 pY[ 0 ][ 1 ] + 4 ) >> 3

* + - Otherwise if availL is equal to TRUE and availT is equal to FALSE, pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = ( pY[ −1 ][ 0 ] + 2 \* pY[ 0 ][ 0 ] + pY[ 1 ][ 0 ] + 2 ) >> 2 (8‑169)

* + - Otherwise if availL is equal to FALSE and availT is equal to TRUE, pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = ( pY[ 0 ][ −1 ] + 2 \* pY[ 0 ][ 0 ] + pY[ 0 ][ 1 ] + 2 ) >> 2 (8‑170)

* + - Otherwise (availL is equal to FALSE and availT is equal to FALSE), pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = pY[ 0 ][ 0 ] (8‑171)

* + - Otherwise, the following applies:
    - 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 ] +  (8‑172)  
 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 ] +  (8‑173)  
 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 (8‑174)

* 1. When numSampL is greater than 0, the down-sampled neighbouring left luma samples pLeftDsY[ y ] with y = 0..numSampL − 1 are derived as follows:
     + If sps\_cclm\_colocated\_chroma\_flag is equal to 1, the following applies:
     + pLeftDsY[ y ] with y = 1..nTbH − 1 is derived as follows:

pLeftDsY[ y ] = ( pY[ −2 ][ 2 \* y − 1 ] +  
  pY[ −3 ][ 2 \* y ] + 4 \* pY[ −2 ][ 2 \* y ] + pY[ −1 ][ 2 \* y ] +  (8‑175)  
 pY[ −2][ 2 \* y + 1 ] + 4 ) >> 3

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

pLeftDsY[ 0 ] = ( pY[ −2 ][ −1 ] +  
  pY[ −3 ][ 0 ] + 4 \* pY[ −2 ][ 0 ] + pY[ −1 ][ 0 ] +  (8‑176)  
 pY[ −2 ][ 1 ] + 4 ) >> 3

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

pLeftDsY[ 0 ] = ( pY[ −3 ][ 0 ] + 2 \* pY[ −2 ][ 0 ] + pY[ −1 ][ 0 ] + 2 ) >> 2 (8‑177)

* + - Otherwise, the following applies:

pLeftDsY[ y ] = ( pY[ −1 ][ 2 \* y ] +  pY[ −1 ][ 2 \* y + 1 ] +   
 2\* pY[ −2 ][ 2 \* y ] + 2\*pY[ −2 ][ 2 \* y + 1 ] +  (8‑178)  
 pY[ −3 ][ 2 \* y ] +  pY[ −3 ][ 2 \* y + 1 ] + 4 ) >> 3

* 1. When numSampT is greater than 0, the down-sampled neighbouring top luma samples pTopDsY[ x ] with x = 0..numSampT − 1 are specified as follows:
     + If sps\_cclm\_colocated\_chroma\_flag is equal to 1, the following applies:
     + pTopDsY[ x ] with x = 1..numSampT − 1 is derived as follows:
     + If bCTUboundary is equal to FALSE, the following applies:

pTopDsY[ x ] = ( pY[ 2 \* x ][ −3 ] +  
 pY[ 2 \* x − 1 ][ −2 ] + 4 \* pY[ 2 \* x ][ −2 ] + pY[ 2 \* x + 1 ][ −2 ] +  (8‑179)  
 pY[ 2 \* x ][ −1 ] + 4 ) >> 3

* + - Otherwise (bCTUboundary is equal to TRUE), the following applies:

pTopDsY[ x ] = ( pY[ 2 \* x − 1 ][ −1 ] +   
 2\* pY[ 2 \* x ][ −1 ] +  (8‑180)  
 pY[ 2 \* x + 1 ][ −1 ] + 2 ) >> 2

* + - pTopDsY[ 0 ] is derived as follows:
    - If availTL is equal to TRUE and bCTUboundary is equal to FALSE, the following applies:

pTopDsY[ 0 ] = ( pY[ 0 ][ −3 ] +  
 pY[ −1 ][ −2 ] + 4 \* pY[ 0 ][ −2 ] + pY[ 1 ][ −2 ] +  (8‑181)  
 pY[ 0 ][ −1 ] + 4 ) >> 3

* + - Otherwise if availTL is equal to TRUE and bCTUboundary is equal to TRUE, the following applies:

pTopDsY[ 0 ] = ( pY[ −1 ][ −1 ] +   
 2\* pY[ 0 ][ −1 ] +  (8‑182)  
 pY[ 1 ][ −1 ] + 2 ) >> 2

* + - Otherwise if availTL is equal to FALSE and bCTUboundary is equal to FALSE, the following applies:

pTopDsY[ 0 ] = ( pY[ 0 ][ −3 ] + 2 \* pY[ 0 ][ −2 ] + pY[ 0 ][ −1 ] + 2 ) >> 2 (8‑183)

* + - Otherwise (availTL is equal to FALSE and bCTUboundary is equal to TRUE), the following applies:

pTopDsY[ 0 ] = pY[ 0 ][ −1 ] (8‑184)

* + - Otherwise, the following applies:
    - pTopDsY[ x ] with x = 1..numSampT − 1 is derived as follows:
    - If bCTUboundary is equal to FALSE, the following applies:

pTopDsY[ x ] = ( pY[ 2 \* x − 1 ][ −2 ] + pY[ 2 \* x − 1 ][ −1 ] +   
 2\* pY[ 2 \* x ][ −2 ] +  2\*pY[ 2 \* x ][ −1 ] +  (8‑185)  
 pY[ 2 \* x + 1 ][ −2 ] + pY[ 2 \* x + 1 ][ −1 ] + 4 ) >> 3

* + - Otherwise (bCTUboundary is equal to TRUE), the following applies:

pTopDsY[ x ] = ( pY[ 2 \* x − 1 ][ −1 ] +   
 2\* pY[ 2 \* x ][ −1 ] +  (8‑186)  
 pY[ 2 \* x + 1 ][ −1 ] + 2 ) >> 2

* + - pTopDsY[ 0 ] is derived as follows:
    - If availTL is equal to TRUE and bCTUboundary is equal to FALSE, the following applies:

pTopDsY[ 0 ] = ( pY[ − 1 ][ −2 ] + pY[ − 1 ][ −1 ] +   
 2\* pY[ 0 ][ −2 ] + 2\*pY[ 0 ][ −1 ] +  (8‑187)  
 pY[ 1 ][ −2 ] + pY[ 1 ][ −1 ] + 4 ) >> 3

* + - Otherwise if availTL is equal to TRUE and bCTUboundary is equal to TRUE, the following applies:

pTopDsY[ 0 ] = ( pY[ − 1 ][ −1 ] +   
 2\* pY[ 0 ][ −1 ] +  (8‑188)  
 pY[ 1 ][ −1 ] + 2 ) >> 2

* + - Otherwise if availTL is equal to FALSE and bCTUboundary is equal to FALSE, the following applies:

pTopDsY[ 0 ] = ( pY[ 0 ][ −2 ] + pY[ 0 ][ −1 ] + 1 ) >> 1 (8‑189)

* + - Otherwise (availTL is equal to FALSE and bCTUboundary is equal to TRUE), the following applies:

pTopDsY[ 0 ] = pY[ 0 ][ −1 ] (8‑190)

* 1. The variables nS, xS, yS are derived as follows:
     + If predModeIntra is equal to INTRA\_LT\_CCLM, the following applies:

nS = ( ( availL && availT ) ? Min( nTbW, nTbH ) : ( availL ? nTbH : nTbW ) ) (8‑191)

xS = 1 << ( ( ( nTbW > nTbH ) && availL && availT ) ? ( Log2( nTbW) − Log2( nTbH ) ) : 0 ) (8‑192)

yS = 1 << ( ( ( nTbH > nTbW ) && availL && availT ) ? ( Log2( nTbH) − Log2( nTbW ) ) : 0 ) (8‑193)

* + - Otherwise if predModeIntra is equal to INTRA\_L\_CCLM, the following applies:

nS = numSampL (8‑194)

xS = 1 (8‑195)

yS = 1 (8‑196)

* + - Otherwise (predModeIntra is equal to INTRA\_T\_CCLM), the following applies:

nS = numSampT (8‑197)

xS = 1 (8‑198)

yS = 1 (8‑199)

* 1. The variables minY, maxY, minC and maxC are derived as follows:
     + The variable minY is set equal to 1 << (BitDepthY) + 1 and the variable maxY is set equal to −1.
     + If availT is equal to TRUE, the variables minY, maxY, minC and maxC with x = 0..nS − 1 are derived as follows:
     + If minY is greater than pTopDsY[ x \* xS ], the following applies:

minY = pTopDsY[ x \* xS ] (8‑200)

minC = p[ x \* xS ][ −1 ] (8‑201)

* + - If maxY is less than pTopDsY[ x \* xS ], the following applies:

maxY = pTopDsY[ x \* xS ] (8‑202)

maxC = p[ x \* xS ][ −1 ]  (8‑203)

* + - If availL is equal to TRUE, the variables minY, maxY, minC and maxC with y = 0..nS − 1 are derived as follows:
    - If minY is greater than pLeftDsY[ y \* yS ], the following applies:

minY = pLeftDsY[ y \* yS ] (8‑204)

minC = p[ −1 ][ y \* yS ] (8‑205)

* + - If maxY is less than pLeftDsY[ y \* yS ], the following applies:

maxY = pLeftDsY[ y \* yS ] (8‑206)

maxC = p[ −1 ][ y \* yS ] (8‑207)

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

k = 0 (8‑208)

a = 0 (8‑209)

b = 1 << ( BitDepthC − 1) (8‑210)

* + - Otherwise, the following applies:

diff = maxY − minY (8‑211)

* + - If diff is not equal to 0, the following applies:

diffC = maxC − minC (8‑212)

x = Floor( Log2( diff ) ) (8‑213)

normDiff = ( ( diff << 4 ) >> x ) & 15 (8‑214)

x += ( normDiff  !=  0 ) ? 1 : 0 (8‑215)

y = Floor( Log2( Abs ( diffC ) ) ) + 1 (8‑216)

a = ( diffC \* ( divSigTable[ normDiff ] | 8 ) + 2y − 1 ) >> y (8‑217)

k = ( ( 3 + x − y ) < 1 ) ? 1 : 3 + x − y (8‑218)

a = ( ( 3 + x − y ) < 1 ) ? Sign( a ) \* 15 : a (8‑219)

b = minC − ( ( a \* minY ) >> k ) (8‑220)

where divSigTable[ ] is specified as follows:

divSigTable[ ] = { 0, 7, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1, 1, 1, 0 } (8‑221)

* + - Otherwise (diff is equal to 0), the following applies:

k = 0 (8‑222)

a = 0 (8‑223)

b = minC (8‑224)

* 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 ) (8‑225)

##### 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 ] (8‑226)  
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 or INTRA\_DC, the following applies:

refL[ x ][ y ] = p[ −1 ][ y ] (8‑227)

refT[ x ][ y ] = p[ x ][ −1 ] (8‑228)

wT[ y ] = 32  >>  ( ( y  <<  1 )  >>  nScale ) (8‑229)

wL[ x ] = 32  >>  ( ( x  <<  1 )  >>  nScale ) (8‑230)

wTL[ x ][ y ] = ( predModeIntra  = =  INTRA\_DC )  ?  ( ( wL[ x ]>>  4 ) + ( wT[ y ]>>  4 ) )  :  0 (8‑231)

* Otherwise, if predModeIntra is equal to INTRA\_ANGULAR18 or INTRA\_ANGULAR50, the following applies:

refL[ x ][ y ] = p[ −1 ][ y ] (8‑232)

refT[ x ][ y ] = p[ x ][ −1 ] (8‑233)

wT[ y ] = ( predModeIntra  = = INTRA\_ANGULAR18 ) ? 32  >>  ( ( y  <<  1 )  >>  nScale ) : 0 (8‑234)

wL[ x ] = ( predModeIntra  = = INTRA\_ANGULAR50 ) ? 32  >>  ( ( x  <<  1 )  >>  nScale ) : 0 (8‑235)

wTL[ x ][ y ] = ( predModeIntra  = = INTRA\_ANGULAR18 ) ? wT[ y ] : wL[ x ] (8‑236)

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

refL[ x ][ y ] = p[ −1 ][ x + y + 1 ] (8‑237)

refT[ x ][ y ] = p[ x + y + 1 ][ −1 ] (8‑238)

wT[ y ] = ( 32  >>  1 )  >>  ( ( y  <<  1 )  >>  nScale ) (8‑239)

wL[ x ] = ( 32  >>  1 )  >>  ( ( x  <<  1 )  >>  nScale ) (8‑240)

wTL[ x ][ y ] = 0 (8‑241)

* 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 8.4.4.2.7 depending on intraPredMode:

dXPos[ y ] = ( ( y + 1 ) \* invAngle + 2 )  >>  2  
dXFrac[ y ] = dXPos[ y ] & 63 (8‑242)  
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 (8‑243)

refT[ x ][ y ] = ( dX[ x ][ y ] < refW − 1 )  ?  mainRef[ dX[ x ][ y ] + ( dXFrac[ y ]  >>  5 ) ]  :  0 (8‑244)

wT[ y ] = ( dX[ x ][ y ] < refW − 1 )  ?  32  >>  ( ( y  <<  1 )  >>  nScale )  :  0 (8‑245)

wL[ x ] = 0 (8‑246)

wTL[ x ][ y ] = 0 (8‑247)

* 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 8.4.4.2.7 depending on intraPredMode:

dYPos[ x ] = ( ( x + 1 ) \* invAngle + 2 )  >>  2  
dYFrac[ x ] = dYPos[ x ] & 63 (8‑248)  
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 )  ?  sideRef[ dY[ x ][ y ] + ( dYFrac[ x ]  >>  5 ) ]  :  0 (8‑249)

refT[ x ][ y ] = 0 (8‑250)

wT[ y ] = 0 (8‑251)

wL[ x ] = ( dY[ x ][ y ] < refH − 1 )  ?  32  >>  ( ( x  <<  1 )  >>  nScale )  :  0 (8‑252)

wTL[ x ][ y ] = 0 (8‑253)

* 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 ] + (8‑254) ( 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,
* 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.

The derivation process for quantization parameters as specified in clause 8.7.1 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, and the variable treeType as inputs.

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

1. The variable dmvrFlag is set equal to 0.
2. The motion vector components and reference indices of the current coding unit are derived as follows:

* If merge\_triangle\_flag[ xCb ][ yCb ], inter\_affine\_flag[ xCb ][ yCb ] and merge\_subblock\_flag[ xCb ][ yCb ] are all equal to 0, the following applies:
* The derivation process for motion vector components and reference indices as specified in clause 8.5.2.1 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 luma motion vectors mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], the reference indices refIdxL0 and refIdxL1 and the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], and the bi-prediction weight index gbiIdx as outputs.
* When all of the following conditions are true, dmvrFlag is set equal to 1:
* sps\_dmvr\_enabled\_flag is equal to 1
* merge\_flag[ xCb ][ yCb ] is equal to 1
* both predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ] are equal to 1
* mmvd\_flag[ xCb ][ yCb ] is equal to 0
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0 ]) is equal to DiffPicOrderCnt( RefPicList[ 1 ][ refIdxL1 ], currPic )
* cbHeight is greater than or equal to 8
* cbHeight\*cbWidth is greater than or equal to 64
* If dmvrFlag is equal to 1, the following applies:
* For X being 0 and 1, 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 8.5.7.2 with X and refIdxLX as inputs.
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the subblock width sbWidth and the subblock height sbHeight are derived as follows:

numSbX = ( cbWidth > 16 )  ?  ( cbWidth >> 4 ) : 1 (8‑255)

numSbY = ( cbHeight > 16 )  ?  ( cbHeight >> 4 ) : 1 (8‑256)

sbWidth = ( cbWidth > 16 )  ?  16 : cbWidth (8‑257)

sbHeight = ( cbHeight > 16 )  ?  16 : cbHeight (8‑258)

* For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:
* The luma motion vectors mvLX[ xSbIdx ][ ySbIdx ] and the prediction list utilization flags predFlagLX[ xSbIdx ][ ySbIdx ] with X equal to 0 and 1, and the luma location ( xSb[xSbIdx][ySbIdx], ySb[xSbIdx][ySbIdx] ) specifying the top-left sample of the coding subblock relative to the top‑left luma sample of the current picture are derived as follows:

mvLX[ xSbIdx ][ ySbIdx ] = mvLX[ 0 ][ 0 ] (8‑259)

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLX[ 0 ][ 0 ] (8‑260)

xSb[ xSbIdx ][ ySbIdx ] =  xCb + xSbIdx \* sbWidth (8‑261)

ySb[ xSbIdx ][ ySbIdx ] =  yCb + ySbIdx \* sbHeight (8‑262)

* The decoder side motion vector refimenent process specified in clause  8.5.3.1 is invoked with xSb[ xSbIdx ][ ySbIdx ], ySb[ xSbIdx ][ ySbIdx ], sbWidth, sbHeight, the motion vectors mvLX[ xSbIdx ][ ySbIdx ] and the reference picture array refPicLXL as inputs and delta motion vectors dMvLX[ xSbIdx ][ ySbIdx ] as outputs with X equal to 0 and 1.
* The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLX[ xSbIdx ][ ySbIdx ] and refIdxLX as inputs, and mvCLX[ xSbIdx ][ ySbIdx ] as outputs with X equal to 0 and 1.
* Otherwise (dmvrFlag is equal to 0), the following applies:
* When treeType is equal to SINGLE\_TREE and predFlagLX[ 0 ][0 ], with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLX[ 0 ][ 0 ] and refIdxLX as inputs, and mvCLX[ 0 ][ 0 ] as output.
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are both set equal to 1.
* Otherwise, if merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1, inter\_affine\_flag[ xCb ][ yCb ] and merge\_subblock\_flag[ xCb ][ yCb ] are both equal to 0, the derivation process for triangle motion vector components and reference indices as specified in clause 8.5.4.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the luma motion vectors mvLA and mvLB, the chroma motion vectors mvCLA and mvCLB, the reference indices refIdxLA and refIdxLB and the prediction list flags predFlagLA and predFlagLB as outputs.
* Otherwise (inter\_affine\_flag[ xCb ][ yCb ] or merge\_subblock\_flag[ xCb ][ yCb ] is equal to 1), the derivation process for subblock motion vector components and reference indices as specified in clause 8.5.5.1 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 number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the prediction list utilization flags predFlagLX[ xSbIdx ][ ySbIdx ], 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, and with X being 0 or 1, and the bi-prediction weight index gbiIdx as outputs.

1. The arrays of luma and chroma motion vectors after decoder side motion vector refinement, refMvLX[ xSbIdx ][ ySbIdx ] and refMvCLX[ xSbIdx ][ ySbIdx ], with X being 0 and 1, are derived as follows for xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1:

* If dmvrFlag is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with refMvLX[ xSbIdx ][ ySbIdx ] and refIdxLX as inputs, and refMvCLX[ xSbIdx ][ ySbIdx ] as output and the input refMvLX[ xSbIdx ][ ySbIdx ] is derived as follows;

refMvLX[ xSbIdx ][ ySbIdx ] = mvLX[ xSbIdx ][ ySbIdx ] + dMvLX[ xSbIdx ][ ySbIdx ] (8‑263)

* Otherwise (dmvrFlag is equal to 0), the following applies:

refMvLX[ xSbIdx ][ ySbIdx ] = mvLX[ xSbIdx ][ ySbIdx ] (8‑264)

refMvCLX [ xSbIdx ][ ySbIdx ] = mvCLX[ xSbIdx ][ ySbIdx ] (8‑265)

NOTE – The array refMvLX is stored in MvDmvrLX and used in the derivation process for collocated motion vectors in clause 8.5.2.12. The array of non-refine luma motion vectors MvLX is used in the spatial motion vector prediction and deblocking boundary strength derivation processes.

1. When ciip\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for intra prediction mode in combined merge and intra prediction as specified in 8.5.6 is invoked.
2. The prediction samples of the current coding unit are derived as follows:

* If merge\_triangle\_flag[ xCb ][ yCb ] is equal to 0, the prediction samples of the current coding unit are derived as follows:
  + - * The decoding process for inter blocks as specified in clause 8.5.7.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the 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 ], and the refined luma motion vectors refMvL0[ xSbIdx ][ ySbIdx ] and refMvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the generalization bi-prediction weight index gbiIdx, and the variable cIdx set equal to 0 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array predSamplesL of prediction luma samples as outputs.
      * The decoding process for inter blocks as specified in clause 8.5.7.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], and the refined chroma motion vectors refMvCL0[ xSbIdx ][ ySbIdx ] and refMvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the generalization bi-prediction weight index gbiIdx, and the variable cIdx set equal to 1 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth / 2)x(cbHeight / 2) array predSamplesCb of prediction chroma samples for the chroma components Cb as outputs.
      * The decoding process for inter blocks as specified in clause 8.5.7.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], and the refined chroma motion vectors refMvCL0[ xSbIdx ][ ySbIdx ] and refMvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the generalization bi-prediction weight index gbiIdx, and the variable cIdx set equal to 2 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth / 2)x(cbHeight / 2) array predSamplesCr of prediction chroma samples for the chroma components Cr as outputs.
* Otherwise (merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1), the decoding process for triangular inter blocks as specified in clause 8.5.8.1 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 mvLA and mvLB, the chroma motion vectors mvCLA and mvCLB, the reference indices refIdxLA and refIdxLB, and the prediction list flags predFlagLA and predFlagLB 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 predSamplesCb and predSamplesCr of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

1. The variables NumSbX[ xCb ][ yCb ] and NumSbY[ xCb ][ yCb ] are set equal to numSbX and numSbY, respectively.
2. 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 8.5.9 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 cIdxset equal to 0 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 8.5.9 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 cIdxset equal to 1 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 8.5.9 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 cIdxset equal to 2 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 8.7.5 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 set equal to 0, 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 8.7.5 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 set equal to 1, 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 8.7.5 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 set equal to 2, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to predSamplesCr and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to resSamplesCr 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 luma motion vectors in 1/16 fractional-sample accuracy mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ],
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ],
* the bi-prediction weight index gbiIdx.

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

For the derivation of the variables mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], refIdxL0 and refIdxL1, as well as predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], 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 8.5.2.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight inputs, and the output being the luma motion vectors mvL0[ 0 ][ 0 ], mvL1[ 0 ][ 0 ], the reference indices refIdxL0, refIdxL1, the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], and the bi-prediction weight index gbiIdx.
* Otherwise, the following applies:
* For X being replaced by either 0 or 1 in the variables predFlagLX[ 0 ][0 ], mvLX[ 0 ][0 ] and refIdxLX, in PRED\_LX, and in the syntax elements ref\_idx\_lX and MvdLX, the following ordered steps apply:

1. The variables refIdxLX and predFlagLX[ 0 ][0 ] are derived as follows:

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

refIdxLX = ref\_idx\_lX[ xCb ][ yCb ] (8‑266)

predFlagLX[ 0 ][0 ] = 1 (8‑267)

* Otherwise, the variables refIdxLX and predFlagLX[ 0 ][0 ] are specified by:

refIdxLX = −1 (8‑268)

predFlagLX[ 0 ][0 ] = 0 (8‑269)

1. The variable mvdLX is derived as follows:

mvdLX[ 0 ] = MvdLX[ xCb ][ yCb ][ 0 ] (8‑270)

mvdLX[ 1 ] = MvdLX[ xCb ][ yCb ][ 1 ] (8‑271)

1. When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for luma motion vector prediction in clause 8.5.2.8 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[ 0 ][ 0 ] is equal to 1, the luma motion vector mvLX[ 0 ][ 0 ] is derived as follows:

uLX[ 0 ] = ( mvpLX[ 0 ] + mvdLX[ 0 ] + 218 ) % 218 (8‑272)

mvLX[ 0 ][ 0 ][ 0 ] = ( uLX[ 0 ] >= 217 ) ? ( uLX[ 0 ] − 218 ) : uLX[ 0 ] (8‑273)

uLX[ 1 ] = ( mvpLX[ 1 ] + mvdLX[ 1 ] + 218 ) % 218 (8‑274)

mvLX[ 0 ][ 0 ][ 1 ] = ( uLX[ 1 ] >= 217 ) ? ( uLX[ 1 ] − 218 ) : uLX[ 1 ] (8‑275)

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

* The bi-prediction weight index gbiIdx is set equal to gbi\_idx[ xCb ][ yCb ].

When all of the following conditions are true, refIdxL1 is set equal to −1, predFlagL1 is set equal to 0, and gbiIdx is set equal to 0:

* predFlagL0[ 0 ][ 0 ] is equal to 1.
* predFlagL1[ 0 ][ 0 ] is equal to 1.
* cbWidth is equal to 4.
* cbHeight is equal to 4.

The updating process for the history-based motion vector predictor list as specified in clause 8.5.2.16 is invoked with luma motion vectors mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], reference indices refIdxL0 and refIdxL1, prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], and bi-prediction weight index gbiIdx.

#### 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 luma motion vectors in 1/16 fractional-sample accuracy mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ],
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ],
* the bi-prediction weight index gbiIdx.

The bi-prediction weight index gbiIdx is set equal to 0.

The variables xSmr, ySmr, smrWidth, smrHeight, and smrNumHmvpCand are derived as follows:

xSmr = IsInSmr[ xCb ][ yCb ]  ?  SmrX[ xCb ][ yCb ]  :  xCb (8‑276)

ySmr = IsInSmr[ xCb ][ yCb ]  ?  SmrY[ xCb ][ yCb ]  :  yCb (8‑277)

smrWidth = IsInSmr[ xCb ][ yCb ]  ?  SmrW[ xCb ][ yCb ]  :  cbWidth (8‑278)

smrHeight = IsInSmr[ xCb ][ yCb ]  ?  SmrH[ xCb ][ yCb ]  :  cbHeight (8‑279)

smrNumHmvpCand = IsInSmr[ xCb ][ yCb ]  ?  NumHmvpSmrCand  :  NumHmvpCand (8‑280)

The motion vectors mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], the reference indices refIdxL0 and refIdxL1 and the prediction utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ] are derived by the following ordered steps:

1. The derivation process for merging candidates from neighbouring coding units as specified in clause 8.5.2.3 is invoked with the luma coding block location ( xCb, yCb ) set equal to ( xSmr, ySmr ), the luma coding block width cbWidth, and the luma coding block height cbHeight set equal to smrWidth and smrHeight 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, and the bi-prediction weight indices gbiIdxA0,gbiIdxA1, gbiIdxB0,gbiIdxB1, gbiIdxB2.
2. The reference indices, refIdxLXCol, with X being 0 or 1, and the bi-prediction weight index gbiIdxCol for the temporal merging candidate Col are set equal to 0.
3. The derivation process for temporal luma motion vector prediction as specified in in clause 8.5.2.11 is invoked with the luma location ( xCb, yCb ) set equal to ( xSmr, ySmr ), the luma coding block width cbWidth, the luma coding block height cbHeight set equal to smrWidth and smrHeight 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 (8‑281)

predFlagL0Col = availableFlagL0Col (8‑282)

predFlagL1Col = 0 (8‑283)

1. When tile\_group\_type is equal to B, the derivation process for temporal luma motion vector prediction as specified in clause 8.5.2.11 is invoked with the luma location ( xCb, yCb ) set equal to ( xSmr, ySmr ), the luma coding block width cbWidth, the luma coding block height cbHeight set equal to smrWidth and smrHeight 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 (8‑284)

predFlagL1Col = availableFlagL1Col (8‑285)

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 (8‑286)if( availableFlagA0 )  
 mergeCandList[ i++ ] = A0if( 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 numCurrMergeCand is less than (MaxNumMergeCand − 1) and smrNumHmvpCand is greater than 0, the following applies:

* The derivation process of history-based merging candidates as specified in 8.5.2.6 is invoked with mergeCandList, isInSmr set equal to IsInSmr[ xCb ][ yCb ], and numCurrMergeCand as inputs, and modified mergeCandList and numCurrMergeCand as outputs.
* numOrigMergeCand is set equal to numCurrMergeCand.

1. When numCurrMergeCand is less than MaxNumMergeCand and greater than 1, the following applies:

* The derivation process for pairwise average merging candidate specified in clause 8.5.2.4 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 refIdxL0avgCand and refIdxL1avgCand, the prediction list utilization flags predFlagL0avgCand and predFlagL1avgCand and the motion vectors mvL0avgCand and mvL1avgCand of candidate avgCand being added into mergeCandList. The bi-prediction weight index gbiIdx of candidate avgCand being added into mergeCandList is set equal to 0.
* numOrigMergeCand is set equal to numCurrMergeCand.

1. The derivation process for zero motion vector merging candidates specified in clause 8.5.2.5 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 bi-prediction weight index gbiIdx of every new candidate zeroCandm being added into mergeCandList is set equal to 0. The number of candidates being added, numZeroMergeCand, is set equal to ( numCurrMergeCand − numOrigMergeCand ). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand − 1, inclusive.
2. 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:

refIdxLX = refIdxLXN (8‑287)

predFlagLX[ 0 ][ 0 ] = predFlagLXN (8‑288)

mvLX[ 0 ][ 0 ][ 0 ] = mvLXN[ 0 ] (8‑289)

mvLX[ 0 ][ 0 ][ 1 ] = mvLXN[ 1 ] (8‑290)

gbiIdx = gbiIdxN (8‑291)

1. When mmvd\_flag[ xCb ][ yCb ] is equal to 1, the following applies:

* The derivation process for merge motion vector difference as specified in 8.5.2.7 is invoked with the luma location ( xCb, yCb ), the luma motion vectors mvL0[ 0 ][ 0 ], mvL1[ 0 ][ 0 ], the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ] as inputs, and the motion vector differences mMvdL0 and mMvdL1 as outputs.
* The motion vector difference mMvdLX is added to the merge motion vectors mvLX for X being 0 and 1 as follows:

mvLX[ 0 ][ 0 ][ 0 ] += mMvdLX[ 0 ] (8‑292)

mvLX[ 0 ][ 0 ][ 1 ] += mMvdLX[ 1 ] (8‑293)

#### 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,
* the bi-prediction weight indices gbiIdxA0,gbiIdxA1, gbiIdxB0,gbiIdxB1, and gbiIdxB2.

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, and gbiIdxA1 is set equal to 0.
* Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvLXA1 = MvLX[ xNbA1 ][ yNbA1 ] (8‑294)

refIdxLXA1 = RefIdxLX[ xNbA1 ][ yNbA1 ] (8‑295)

predFlagLXA1 = PredFlagLX[ xNbA1 ][ yNbA1 ] (8‑296)

gbiIdxA1 = GbiIdx[ xNbA1 ][ yNbA1 ] (8‑297)

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, and gbiIdxB1 is set equal to 0:
  + 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 ] (8‑298)

refIdxLXB1 = RefIdxLX[ xNbB1 ][ yNbB1 ] (8‑299)

predFlagLXB1 = PredFlagLX[ xNbB1 ][ yNbB1 ] (8‑300)

gbiIdxB1 = GbiIdx[ xNbB1 ][ yNbB1 ] (8‑301)

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, and gbiIdxB0 is set equal to 0:
  + 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.
  + availableA1 is equal to TRUE, the luma locations ( xNbA1, yNbA1 ) and ( xNbB0, yNbB0 ) have the same motion vectors and the same reference indices and merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1.
* Otherwise, availableFlagB0 is set equal to 1 and the following assignments are made:

mvLXB0 = MvLX[ xNbB0 ][ yNbB0 ] (8‑302)

refIdxLXB0 = RefIdxLX[ xNbB0 ][ yNbB0 ] (8‑303)

predFlagLXB0 = PredFlagLX[ xNbB0 ][ yNbB0 ] (8‑304)

gbiIdxB0 = GbiIdx[ xNbB0 ][ yNbB0 ] (8‑305)

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, and gbiIdxA0 is set equal to 0:
  + 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.
  + availableB1 is equal to TRUE, the luma locations ( xNbB1, yNbB1 ) and ( xNbA0, yNbA0 ) have the same motion vectors and the same reference indices and merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1.
  + availableB0 is equal to TRUE, the luma locations ( xNbB0, yNbB0 ) and ( xNbA0, yNbA0 ) have the same motion vectors and the same reference indices and merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1.
* Otherwise, availableFlagA0 is set equal to 1 and the following assignments are made:

mvLXA0 = MvLX[ xNbA0 ][ yNbA0 ] (8‑306)

refIdxLXA0 = RefIdxLX[ xNbA0 ][ yNbA0 ] (8‑307)

predFlagLXA0 = PredFlagLX[ xNbA0 ][ yNbA0 ] (8‑308)

gbiIdxA0 = GbiIdx[ xNbA0 ][ yNbA0 ] (8‑309)

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, and gbiIdxB2 is set equal to 0:
  + 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.
  + availableB0 is equal to TRUE, the luma locations ( xNbB0, yNbB0 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices and merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1.
  + availableA0 is equal to TRUE, the luma locations ( xNbA0, yNbA0 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices and merge\_triangle\_flag[ xCb ][ yCb ] is equal to 1.
  + availableFlagA0 + availableFlagA1 + availableFlagB0 + availableFlagB1 is equal to 4 and merge\_triangle\_flag[ xCb ][ yCb ] is equal to 0.
* Otherwise, availableFlagB2 is set equal to 1 and the following assignments are made:

mvLXB2 = MvLX[ xNbB2 ][ yNbB2 ] (8‑310)

refIdxLXB2 = RefIdxLX[ xNbB2 ][ yNbB2 ] (8‑311)

predFlagLXB2 = PredFlagLX[ xNbB2 ][ yNbB2 ] (8‑312)

gbiIdxB2 = GbiIdx[ xNbB2 ][ yNbB2 ] (8‑313)

#### Derivation process for pairwise average merging candidate

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.

Outputs of this process are:

* the merging candidate list mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the reference indices refIdxL0avgCand and refIdxL1avgCand of candidate avgCand added into mergeCandList during the invocation of this process,
* the prediction list utilization flags predFlagL0avgCand and predFlagL1avgCand of candidate avgCand added into mergeCandList during the invocation of this process,
* the motion vectors in 1/16 fractional-sample accuracy mvL0avgCand and mvL1avgCand of candidate avgCand added into mergeCandList during the invocation of this process.

The variable numRefLists is derived as follows:

numRefLists  =  ( tile\_group\_type = = B )  ?  2 : 1 (8‑314)

The following assignments are made, with p0Cand being the candidate at position 0 and p1Cand being the candidate at position 1 in the merging candidate list mergeCandList:

p0Cand = mergeCandList[ 0 ] (8‑315)

p1Cand = mergeCandList[ 1 ] (8‑316)

The candidate avgCand is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to avgCand, and the reference indices, the prediction list utilization flags and the motion vectors of avgCand are derived as follows and numCurrMergeCand is incremented by 1:

* For each reference picture list LX with X ranging from 0 to ( numRefLists − 1 ), the following applies:
  + If predFlagLXp0Cand is equal to 1 and predFlagLXp1Cand is equal to 1, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], and mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = refIdxLXp0Cand (8‑317)

predFlagLXavgCand = 1 (8‑318)

mvLXavgCand[ 0 ] = ( mvLXp0Cand[ 0 ] + mvLXp1Cand[ 0 ] ) / 2 (8‑319)

mvLXavgCand[ 1 ] = ( mvLXp0Cand[ 1 ] + mvLXp1Cand[ 1 ] ) / 2 (8‑320)

* + Otherwise, if predFlagLXp0Cand is equal to 1 and predFlagLXp1Cand is equal to 0, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = refIdxLXp0Cand (8‑321)

predFlagLXavgCand = 1 (8‑322)

mvLXavgCand[ 0 ] = mvLXp0Cand[ 0 ] (8‑323)

mvLXavgCand[ 1 ] = mvLXp0Cand[ 1 ] (8‑324)

* + Otherwise, if predFlagLXp0Cand is equal to 0 and predFlagLXp1Cand is equal to 1, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = refIdxLXp1Cand (8‑325)

predFlagLXavgCand = 1 (8‑326)

mvLXavgCand[ 0 ] = mvLXp1Cand[ 0 ] (8‑327)

mvLXavgCand[ 1 ] = mvLXp1Cand[ 1 ] (8‑328)

* + Otherwise, if predFlagLXp0Cand is equal to 0 and predFlagLXp1Cand is equal to 0, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = −1 (8‑329)

predFlagLXavgCand = 0 (8‑330)

mvLXavgCand[ 0 ] = 0 (8‑331)

mvLXavgCand[ 1 ] = 0 (8‑332)

* When numRefLists is equal to 1, the following applies:

refIdxL1avgCand = −1 (8‑333)

predFlagL1avgCand = 0 (8‑334)

#### 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 tile\_group\_type is equal to P, numRefIdx is set equal to NumRefIdxActive[ 0 ].
* Otherwise (tile\_group\_type is equal to B), numRefIdx is set equal to Min( NumRefIdxActive[ 0 ], NumRefIdxActive[ 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 tile\_group\_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 (8‑335)

refIdxL1zeroCandm = −1 (8‑336)

predFlagL0zeroCandm = 1 (8‑337)

predFlagL1zeroCandm = 0 (8‑338)

mvL0zeroCandm[ 0 ] = 0 (8‑339)

mvL0zeroCandm[ 1 ] = 0 (8‑340)

mvL1zeroCandm[ 0 ] = 0 (8‑341)

mvL1zeroCandm[ 1 ] = 0 (8‑342)

numCurrMergeCand = numCurrMergeCand + 1 (8‑343)

* + Otherwise (tile\_group\_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 (8‑344)

refIdxL1zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑345)

predFlagL0zeroCandm = 1 (8‑346)

predFlagL1zeroCandm = 1 (8‑347)

mvL0zeroCandm[ 0 ] = 0 (8‑348)

mvL0zeroCandm[ 1 ] = 0 (8‑349)

mvL1zeroCandm[ 0 ] = 0 (8‑350)

mvL1zeroCandm[ 1 ] = 0 (8‑351)

numCurrMergeCand = numCurrMergeCand + 1 (8‑352)

1. The variable zeroIdx is incremented by 1.

#### Derivation process for history-based merging candidates

Inputs to this process are:

* a merge candidate list mergeCandList,
* a variable isInSmr specifying whether the current coding unit is inside a shared mergign candidate region,
* the number of available merging candidates in the list numCurrMergeCand.

Outputs to this process are:

* the modified merging candidate list mergeCandList,
* the modified number of merging candidates in the list numCurrMergeCand.

The variables isPrunedA1 and isPrunedB1 are both set equal to FALSE.

The array smrHmvpCandList and the variable smrNumHmvpCand are derived as follows:

smrHmvpCandList = isInSmr  ?  HmvpSmrCandList  :  HmvpCandList (8‑353)

smrNumHmvpCand = isInSmr  ?  NumHmvpSmrCand  :  NumHmvpCand (8‑354)

For each candidate in smrHmvpCandList[ hMvpIdx ] with index hMvpIdx = 1..smrNumHmvpCand, the following ordered steps are repeated until numCurrMergeCand is equal to ( MaxNumMergeCand − 1):

1. The variable sameMotion is derived as follows:
   * If all of the following conditions are true for any merging candidate N with N being A1 or B1, sameMotion and isPrunedN are both set equal to TRUE:
   * hMvpIdx is less than or equal to 2.
   * The candidate smrHmvpCandList[ smrNumHmvpCand − hMvpIdx] is equal to the merging candidate N.
   * isPrunedN is equal to FALSE.
   * Otherwise, sameMotion is set equal to FALSE.
2. When sameMotion is equal to FALSE, the candidate smrHmvpCandList[ smrNumHmvpCand − hMvpIdx] is added to the merging candidate list as follows:

mergeCandList[ numCurrMergeCand++ ] = smrHmvpCandList[ smrNumHmvpCand − hMvpIdx ] (8‑355)

#### Derivation process for merge motion vector difference

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,
* reference indices refIdxL0 and refIdxL1,
* prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are the luma merge motion vector differences in 1/16 fractional-sample accuracy mMvdL0 and mMvdL1.

The variable currPic specifies the current picture.

The luma merge motion vector differences mMvdL0 and mMvdL1 are derived as follows:

* If both predFlagL0 and predFlagL1 are equal to 1, the following applies:

currPocDiffL0  =  DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0 ] ) (8‑356)

currPocDiffL1  =  DiffPicOrderCnt( currPic, RefPicList[ 1 ][ refIdxL1 ] ) (8‑357)

* If currPocDiffL0 is equal to currPocDiffL1, the following applies:

mMvdL0[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑358)

mMvdL0[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑359)

mMvdL1[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑360)

mMvdL1[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑361)

* Otherwise, if Abs( currPocDiffL0 ) is greater than or equal to Abs( currPocDiffL1 ), the following applies:

td = Clip3( −128, 127, currPocDiffL0 ) (8‑362)

tb = Clip3( −128, 127, currPocDiffL1 ) (8‑363)

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑364)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑365)

mMvdL0[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑366)

mMvdL0[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑367)

mMvdL1[ 0 ] = Clip3( −215, 215 − 1, Sign( distScaleFactor \* mMvdL0[ 0 ] )  \*   (8‑368)  
 ( ( Abs( distScaleFactor \* mMvdL0[ 0 ] ) + 127 ) >> 8 ) )

mMvdL1[ 1 ] = Clip3( −215, 215 − 1, Sign( distScaleFactor \* mMvdL0[ 1 ] )  \*   (8‑369)  
 ( ( Abs( distScaleFactor \* mMvdL0[ 1 ] ) + 127 ) >> 8 ) )

* Otherwise (Abs( currPocDiffL0 ) is less than Abs( currPocDiffL1 )), the following applies:

td = Clip3( −128, 127, currPocDiffL1 ) (8‑370)

tb = Clip3( −128, 127, currPocDiffL0 ) (8‑371)

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑372)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑373)

mMvdL1[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑374)

mMvdL1[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑375)

mMvdL0[ 0 ] = Clip3( −215, 215 − 1, Sign( distScaleFactor \* mMvdL1[ 0 ] )  \*   (8‑376)  
 ( ( Abs( distScaleFactor \* mMvdL1[ 0 ] ) + 127 ) >> 8 ) )

mMvdL0[ 1 ] = Clip3( −215, 215 − 1, Sign( distScaleFactor \* mMvdL1[ 1 ] )  \*   (8‑377)  
 ( ( Abs( distScaleFactor \* mMvdL1[ 1 ] ) + 127 ) >> 8 ) )

* Otherwise ( predFlagL0 or predFlagL1 are equal to 1 ), the following applies for X being 0 and 1:

mMvdLX[ 0 ] = ( predFlagLX = = 1 )  ?  MmvdOffset[ xCb ][ yCb ][ 0 ]  :  0 (8‑378)

mMvdLX[ 1 ] = ( predFlagLX = = 1 )  ?  MmvdOffset[ xCb ][ yCb ][ 1 ]  :  0 (8‑379)

#### 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 8.5.2.9 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 ] ] (8‑380)

#### 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 8.5.2.10 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 8.5.2.14 is invoked 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 8.5.2.11 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 8.5.2.14 is invoked 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:

numCurrMvpCand = 0  
if( availableFlagLXA ) {  
 mvpListLX[ numCurrMvpCand++ ] = mvLXA  
 if( availableFlagLXB && ( mvLXA != mvLXB ) )  
 mvpListLX[ numCurrMvpCand++ ] = mvLXB (8‑381)  
} else if( availableFlagLXB )  
 mvpListLX[ numCurrMvpCand++ ] = mvLXB  
if( numCurrMvpCand < 2 && availableFlagLXCol )  
 mvpListLX[ numCurrMvpCand++ ] = mvLXCol

1. When numCurrMvpCand is less than 2 and NumHmvpCand is greater than 0, the following applies for i= 1..Min( 4, NumHmvpCand ) until numCurrMvpCand is equal to 2:

* For each reference picture list LY with Y = X..( 1 − X ), the following applies until numCurrMvpCand is equal to 2:
* When the reference picture corresponding to the reference index of the history-based motion vector predictor candidate HmvpCandList[ i − 1 ] in the reference picture list LY is the same as the reference picture corresponding to reference index refIdxLX in the reference picture list LX, the following applies:
* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to the LY motion vector of the candidate HmvpCandList[ i − 1 ], rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded LY motion vector of the candidate HmvpCandList[ i − 1 ] as output is assigned to mvpListLX[ numCurrMvpCand++ ].

1. When numCurrMvpCand is less than 2, the following applies for until numCurrMvpCand is equal to 2:

mvpListLX[ numCurrMvpCand ][ 0 ] = 0 (8‑382)

mvpListLX[ numCurrMvpCand ][ 1 ] = 0 (8‑383)

numCurrMvpCand++ (8‑384)

#### 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.

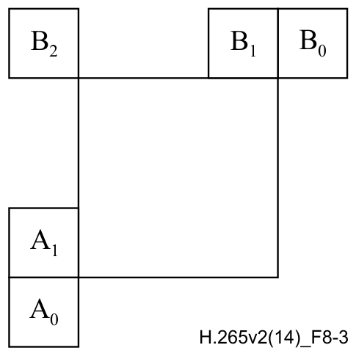


Figure 8‑2 – Spatial motion vector neighbours (informative)

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( RefPicList[ X ][ RefIdxLX[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = MvLX[ xNbAk ][ yNbAk ] (8‑385)

* Otherwise, when PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = MvLY[ xNbAk ][ yNbAk ] (8‑386)

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 ] (8‑387)

refIdxA = RefIdxLX[ xNbAk ][ yNbAk ] (8‑388)

refPicListA = RefPicList[ X ] (8‑389)

* 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 ] (8‑390)

refIdxA = RefIdxLY[ xNbAk ][ yNbAk ] (8‑391)

refPicListA = RefPicList[ Y ] (8‑392)

* When availableFlagLXA is equal to 1, DiffPicOrderCnt( refPicListA[ refIdxA ], RefPicList[ X ][ refIdxLX ] ) is not equal to 0, mvLXA is derived as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑393)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑394)

mvLXA = Clip3( −131072, 131071, Sign( distScaleFactor \* mvLXA ) \*   
 ( ( Abs( distScaleFactor \* mvLXA ) + 127 )  >>  8 ) ) (8‑395)

where td and tb are derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, refPicListA[ refIdxA ] ) ) (8‑396)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicList[ X ][ refIdxLX ] ) ) (8‑397)

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( RefPicList[ X ][ RefIdxLX[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLX[ xNbBk ][ yNbBk ] (8‑398)

refIdxB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑399)

* Otherwise, when PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[ xNbBk ][ yNbBk ] (8‑400)

refIdxB = RefIdxLY[ xNbBk ][ yNbBk ] (8‑401)

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 (8‑402)

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 ] (8‑403)

refIdxB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑404)

refPicListB = RefPicList[ X ] (8‑405)

* 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 ] (8‑406)

refIdxB = RefIdxLY[ xNbBk ][ yNbBk ] (8‑407)

refPicListB = RefPicList[ Y ] (8‑408)

* When availableFlagLXB is equal to 1, DiffPicOrderCnt( refPicListB[ refIdxB ], RefPicList[ X ][ refIdxLX ] ) is not equal to 0, mvLXB is derived as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑409)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑410)

mvLXB = Clip3( −131072, 131071, Sign( distScaleFactor \* mvLXB ) \*  
 ( ( Abs( distScaleFactor \* mvLXB ) + 127 )  >>  8 ) ) (8‑411)

where td and tb are derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, refPicListB[ refIdxB ] ) ) (8‑412)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicList[ X ][ refIdxLX ] ) ) (8‑413)

#### 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 tile\_group\_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 (tile\_group\_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 (8‑414)

yColBr = yCb + cbHeight (8‑415)

* 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 8.5.2.12 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 ) (8‑416)

yColCtr = yCb + ( cbHeight  >>  1 ) (8‑417)

* 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 8.5.2.12 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 ], MvDmvrL0[ 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 ], MvDmvrL1[ 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 or IBC prediction mode, or its reference picture is ColPic, 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 tile group 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, refPicList[ listCol ][ refIdxCol ] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the tile group containing coding block colCb in the collocated picture specified by ColPic, and the following applies:

colPocDiff = DiffPicOrderCnt( ColPic, refPicList[ listCol ][ refIdxCol ] ) (8‑418)

currPocDiff = DiffPicOrderCnt( currPic, RefPicList[ X ][ refIdxLX ] ) (8‑419)

* + - * The temporal motion buffer compression process for collocated motion vectors as specified in clause 8.5.2.15 is invoked with mvCol as input, and the modified mvCol as output.
      * If RefPicList[ X ][ refIdxLX ] is a long-term reference picture, or colPocDiff is equal to currPocDiff, mvLXCol is derived as follows:

mvLXCol = mvCol (8‑420)

* + - * Otherwise, mvLXCol is derived as a scaled version of the motion vector mvCol as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑421)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑422)

mvLXCol =  Clip3( −131072, 131071, Sign( distScaleFactor \* mvCol ) \*   
 ( ( Abs( distScaleFactor \* mvCol ) + 127 )  >>  8 ) ) (8‑423)

where td and tb are derived as follows:

td = Clip3( −128, 127, colPocDiff ) (8‑424)

tb = Clip3( −128, 127, currPocDiff ) (8‑425)

#### Derivation process for chroma motion vectors

Input to this process are:

* a luma motion vector in 1/16 fractional-sample accuracy mvLX,
* the reference index refIdxLX.

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.

The chroma motion vector mvCLX, is derived as follows:

mvCLX[ 0 ] = mvLX[ 0 ] \* 2 / SubWidthC (8‑426)

mvCLX[ 1 ] = mvLX[ 1 ] \* 2 / SubHeightC (8‑427)

#### 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 = ( rightShift = = 0 )  ?  0  :  ( 1  <<  ( rightShift − 1 ) ) (8‑428)

mvX[ 0 ] = ( mvX[ 0 ] >= 0 ? ( mvX[ 0 ] + offset ) >> rightShift :    
 − ( ( − mvX[ 0 ] + offset ) >> rightShift ) ) << leftShift (8‑429)

mvX[ 1 ] = ( mvX[ 1 ] >= 0 ? ( mvX[ 1 ] + offset ) >> rightShift :    
 − ( ( − mvX[ 1 ] + offset ) >> rightShift ) ) << leftShift (8‑430)

#### Temporal motion buffer compression process for collocated motion vectors

Input to this process is a motion vector mv.

Outputs of this process is the rounded motion vector mv.

For each motion vector component compIdx being 0 and 1, mv[ compIdx ] is modifed as follows:

s = mv[ compIdx ] >> 17 (8‑431)

f = Floor( Log2( ( mv[ compIdx ] ^ s ) | 31 ) ) − 4 (8‑432)

mask = ( −1 << f ) >> 1 (8‑433)

round = ( 1 << f ) >> 2 (8‑434)

mv[ compIdx ] = ( mv[ compIdx ] + round ) & mask (8‑435)

NOTE – This process enables storage of collocated motion vectors using a bit reduced representation. Each signed 18-bit motion vector component can be represented in a mantissa plus exponent format with a 6-bit signed mantissa and a 4-bit exponent.

#### Updating process for the history-based motion vector predictor candidate list

Inputs to this process are:

* luma motion vectors in 1/16 fractional-sample accuracy mvL0 and mvL1,
* reference indices refIdxL0 and refIdxL1,
* prediction list utilization flags predFlagL0 and predFlagL1,
* bi-prediction weight index gbiIdx.

The MVP candidate hMvpCand consists of the luma motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0 and predFlagL1, and the bi-prediction weight index gbiIdx.

The candidate list HmvpCandList is modified using the candidate hMvpCand by the following ordered steps:

1. The variable identicalCandExist is set equal to FALSE and the variable removeIdx is set equal to 0.
2. When NumHmvpCand is greater than 0, for each index hMvpIdx with hMvpIdx = 0..NumHmvpCand − 1, the following steps apply until identicalCandExist is equal to TRUE:
   * When hMvpCand is equal to HmvpCandList[ hMvpIdx ], identicalCandExist is set equal to TRUE and removeIdx is set equal to hMvpIdx.
3. The candidate list HmvpCandList is updated as follows:
   * If identicalCandExist is equal to TRUE or NumHmvpCand is equal to MaxNumMergeCand − 1, the following applies:
   * For each index i with i = ( removeIdx + 1 )..( NumHmvpCand − 1 ), HmvpCandList[ i − 1] is set equal to HmvpCandList[ i ].
   * HmvpCandList[ NumHmvpCand − 1 ] is set equal to mvCand.
   * Otherwise (identicalCandExist is equal to FALSE and NumHmvpCand is less than MaxNumMergeCand − 1), the following applies:
   * HmvpCandList[ NumHmvpCand++ ] is set equal to mvCand.

### Decoder side motion vector refinement 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,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0 and mvL1,
* the selected luma reference picture sample arrays refPicL0L and refPicL1L.

Outputs of this process are:

* delta luma motion vectors dMvL0 and dMvL1.

The variable subPelFlag is set to 0. and the variables srRange, offsetH0, offsetH1, offsetV0, and offsetV1 are all set equal to 2.

Both components of the delta luma motion vectors dMvL0 and dMvL1 are set equal to zero and modified as follows:

* For each X being 0 or 1, the ( sbWidth + 2 \* srRange ) x ( sbHeight + 2 \* srRange ) array predSamplesLXL of prediction luma sample values is derived by invoking the fractional sample bilinear interpolation process specified in 8.5.3.2.1 with the luma location ( xSb, ySb ), the prediction block width set equal to ( sbWidth + 2 \* srRange ), the prediction block height set equal to ( sbHeight + 2 \* srRange ), the reference picture sample array refPicLXL, the motion vector mvLX and the refinement search range srRange as inputs.
* The list sadList[ i ] with i = 0..8 is derived by invoking the sum of absolute differences calculation process specified in 8.5.3.3 with sbWidth, sbHeight, offsetH0, offsetH1, offsetV0, offsetV1, predSamplesL0L and predSamplesL1L as inputs.
* When sadList[ 4 ] is greater than or equal to sbHeight \* sbWidth, the following applies:
* The variable bestIdx is derived by invoking the array entry selection process specified in clause 8.5.3.4 with the list sadList[ i ] with i = 0..8 as input.
* If bestIdx is equal to 4, subPelFlag is set equal to 1.
* Otherwise, the following applies:

dX = bestIdx % 3 − 1 (8‑436)

dY = bestIdx / 3 − 1 (8‑437)

dMvL0[ 0 ] += 16 \* dX (8‑438)

dMvL0[ 1 ] += 16 \* dY (8‑439)

offsetH0 += dX (8‑440)

offsetV0 += dY (8‑441)

offsetH1 −= dX (8‑442)

offsetV1 −= dY (8‑443)

* The list sadList[ i ] with i = 0..8 is modifed by invoking the sum of absolute differences calculation process specified in 8.5.3.3 with sbWidth, sbHeight, offsetH0, offsetH1, offsetV0, offsetV1, predSamplesL0L and predSamplesL1L as inputs.
* The variable bestIdx is modified by invoking the array entry selection process specified in clause 8.5.3.4 with the list sadList[ i ] with i = 0..8 as input.
* If bestIdx is equal to 4, subPelFlag is set equal to 1
* Otherwise (bestIdx is not equal to 4), the following applies:

dMvL0[ 0 ] += 16 \* ( bestIdx % 3 − 1 ) (8‑444)

dMvL0[ 1 ] += 16 \* ( bestIdx / 3 − 1 ) (8‑445)

* When subPelFlag is equal to 1, the parametric motion vector refinement process specified in clause 8.5.3.5 is invoked with the list sadList[ i ] with i = 0..8, and the delta motion vector dMvL0 as inputs and the modified dMvL0 as output.
* The delta motion vector dMvL1 is derived as follows:

dMvL1[ 0 ] = −dMvL0[ 0 ] (8‑446)

dMvL1[ 1 ] = −dMvL0[ 1 ] (8‑447)

#### Fractional sample bilinear interpolation process

##### General

Inputs to this process are:

* a luma location ( xSb, ySb ) specifying the top-left sample of the current subblock relative to the top‑left luma sample of the current picture,
* a variable pbWidth specifying the width of the current prediction block in luma samples,
* a variable pbHeight specifying the height of the current prediction block in luma samples,
* a luma motion vector mvLX given in 1/16-luma-sample units,
* the selected reference picture sample array refPicLXL,
* the refinement search range srRange.

Output of this process is:

* a ( pbWidth ) x ( pbHeight ) array predSamplesLXL of luma prediction 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 array refPicLXL.

For each luma sample location ( xL = 0..pbWidth − 1, yL = 0..pbHeight − 1 ) inside the luma prediction sample array predSamplesLXL, the corresponding luma prediction 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 − srRange (8‑448)

yIntL = ySb + ( mvLX[ 1 ]  >>  4 ) + yL − srRange (8‑449)

xFracL = mvLX[ 0 ] & 15 (8‑450)

yFracL = mvLX[ 1 ] & 15 (8‑451)

* The luma prediction sample value predSamplesLXL[ xL ][ yL ] is derived by invoking the luma sample bilinear interpolation process specified in clause 8.5.3.2.2 with ( xIntL, yIntL ), ( xFracL, yFracL ) and refPicLXL as inputs.

##### Luma sample bilinear 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, shift3, shift4, offset1, offset2 and offset3 are derived as follows:

shift1 = BitDepthY − 6 (8‑452)

offset1 = 1 << ( shift1 − 1 ) (8‑453)

shift2 = 4 (8‑454)

offset2 = 1 << ( shift2 − 1 ) (8‑455)

shift3 = 10 − BitDepthY (8‑456)

offset3 = 1 << ( shift3 − 1 ) (8‑457)

shift4 = BitDepthY − 10 (8‑458)

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 fbL[ p ] for each 1/16 fractional sample position p equal to xFracL or  yFracL are specified in Table 8‑7.

The luma locations in full-sample units ( xInti, yInti ) are derived as follows for i = 0..1:

xInti = sps\_ref\_wraparound\_enabled\_flag ?  
 ClipH( ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY, picW, ( xIntL + i ) ) : (8‑459)  
 Clip3( 0, picW − 1, xIntL + i )

yInti = Clip3( 0, picH − 1, yIntL + i ) (8‑460)

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 = BitDepthY <= 10  ?  (refPicLXL[ xInt0 ][ yInt0 ] << shift3 ) :   
 ( ( refPicLXL[ xInt0 ][ yInt0 ] + offset3 ) >> shift4 ) (8‑461)

– Otherwise, if xFracL is not equal to 0 and yFracL is equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = (  + offset1 )  >>  shift1 (8‑462)

– Otherwise, if xFracL is equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = (  + offset1 )  >>  shift1 (8‑463)

– 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..1, is derived as follows:

temp[ n ] = (  + offset1 )  >>  shift1 (8‑464)

* The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL = (  + offset2 )  >>  shift2 (8‑465)

Table 8‑7 – Specification of the luma bilinear interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p.

|  |  |  |
| --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | |
| **fbL[ p ][ 0 ]** | **fbL[ p ][ 1 ]** |
| 1 | 15 | 1 |
| 2 | 14 | 2 |
| 3 | 13 | 3 |
| 4 | 12 | 4 |
| 5 | 11 | 5 |
| 6 | 10 | 6 |
| 7 | 9 | 7 |
| 8 | 8 | 8 |
| 9 | 7 | 9 |
| 10 | 6 | 10 |
| 11 | 5 | 11 |
| 12 | 4 | 12 |
| 13 | 3 | 13 |
| 14 | 2 | 14 |
| 15 | 1 | 15 |

#### Sum of absolute differences calculation process

Inputs to this process are:

* two variables nSbW and nSbH specifying the width and the height of the current subblock,
* two ( nSbW + 4 ) x ( nSbH + 4 ) array predSamplesL0 and predSamplesL1,
* four variables offsetH0, offsetH1, offsetV0, offsetV1 specifying the offset postion of the search center.

Outputs of this process is:

* the list sadList[ i ] specifying the sum of absolute differences for each position i= 0..8.

For each i = 0..8, sadList[ i ] is derived as follows using the prediction sample arrays predSamplesL0 and predSamplesL1, the offsets dX[ i ] and dY[ i ] specified in Table 8‑8, offsetH0, offsetH1, offsetV0, and offsetV1:

sadList[ i ] =  (8‑466)

with

pL0i[x][y] = predSamplesL0[ x + dX[ i ] + offsetH0 ][ 2 \* y + dY[ i ] + offsetV0 ] (8‑467)

pL1i[x][y] = predSamplesL1[ x − sX[ i ] + offsetH1 ][ 2 \* y − dY[ i ] + offsetV1 ] (8‑468)

Table 8‑8 – Specification of the search position offsets dX[ i ] and dY[ i ]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **dX[ i ]** | −1 | 0 | 1 | −1 | 0 | 1 | −1 | 0 | 1 |
| **dY[ i ]** | −1 | −1 | −1 | 0 | 0 | 0 | 1 | 1 | 1 |

#### Array entry selection process

Inputs to this process is a list of sum of absolute differences sadList[ i ] with i = 0..8.

Output of this process is a variable bestIdx specifying the index i of sadList[ i ] with the lowest value.

The following steps are applied to derive the variable bestIdx:

* If sadList[ 1 ] is less than or equal to sadList[ 7 ] and sadList[ 3 ] is less than or equal to sadList[ 5 ], the following applies:

tempIdx = 0 (8‑469)

bestIdx = ( sadList[ 1 ] <= sadList[ 3 ] )  ?  1 : 3 (8‑470)

* Otherwise, if sadList[ 1 ] is greater than sadList[ 7 ] and sadList[ 3 ] is less than or equal to sadList[ 5 ], the following applies:

tempIdx = 6 (8‑471)

bestIdx = ( sadList[ 7 ] <= sadList[ 3 ] )  ?  7 : 3 (8‑472)

* Otherwise, if sadList[ 1 ] is less than or equal to sadList[ 7 ] and sadList[ 3 ] is greater than sadList[ 5 ], the following applies:

tempIdx = 2 (8‑473)

bestIdx = ( sadList[ 1 ] <= sadList[ 5 ] )  ?  1 : 5 (8‑474)

* Otherwise, if sadList[ 1 ] is greater than sadList[ 7 ] and sadList[ 3 ] is greater than sadList[ 5 ], the following applies:

tempIdx = 8 (8‑475)

bestIdx = ( sadList[ 7 ] <= sadList[ 5 ] )  ?  7 : 5 (8‑476)

* When sadList[ 4 ] is less than or equal to sadList[ bestIdx ], bestIdx is set equal to 4.
* When sadList[ tempIdx ] is less than sadList[ bestIdx ], bestIdx is set equal to tempIdx.

#### Parametric motion vector refinement process

Inputs to this process are:

* a list of sum of absolute differences sadList[ i ] with i = 0..8,
* the delta luma motion vector dMvL0 given in 1/16-luma-sample units.

Output of this process is the modified delta luma motion vector dMvL0.

The variable dX is derived as follows:

* If sadList[ 3 ] + sadList[ 5 ] is equal to sadList[ 4 ] << 1,dX is set equal to 0.
* Otherwise, (sadList[ 3 ] + sadList[ 5 ] is greater than sadList[ 4 ] << 1 ), the following applies:

dX = ( ( sadList[ 3 ] − sadList[ 5 ] ) << 3 ) / ( sadList[ 3 ] + sadList[ 5 ] − ( sadList[ 4 ] << 1 ) ) (8‑477)

The variable dY is derived as follows:

* If sadList[ 1 ] + sadList[ 7 ] is equal to sadList[ 4 ] << 1,dY is set equal to 0.
* Otherwise (sadList[ 1 ] + sadList[ 7 ] is greater than sadList[ 4 ] << 1), the following applies:

dY = ( ( sadList[ 1 ] − sadList[ 7 ] ) << 3 ) / ( sadList[ 1 ] + sadList[ 7 ] − ( sadList[ 4 ] << 1 ) ) (8‑478)

The delta luma motion vector dMvL0 is modified as follows:

dMvL0[ 0 ] += dX (8‑479)

dMvL0[ 1 ] += dY (8‑480)

NOTE – dX is constrained to be between −8 and 8 since all values of sadList[ i ] with i = 0..8 are greater than 0 and the smallest value is sadList[ 4 ]. This allows the division to be performed with up to 4 quotient bits and can be implemented using compares, shifts, and subtractions.

### Derivation process for triangle 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 luma motion vectors in 1/16 fractional-sample accuracy mvLA and mvLB,
* the chroma motion vectors in 1/32 fractional-sample accuracy mvCLA and mvCLB,
* the reference indices refIdxLA and refIdxLB,
* the prediction list flags predFlagLA and predFlagLB.

The derivation process for luma motion vectors for triangle merge mode as specified in clause 8.5.4.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight as inputs, and the output being the luma motion vectors mvLA, mvLB, the reference indices refIdxLA, refIdxLB and the prediction list flags predFlagLA and predFlagLB.

The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLA and refIdxLA as input, and the output being mvCLA.

The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLB and refIdxLB as input, and the output being mvCLB.

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

This process is only invoked when merge\_triangle\_flag[ xCb ][ yCb ] 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 luma motion vectors in 1/16 fractional-sample accuracy mvLA and mvLB,
* the reference indices refIdxLA and refIdxLB,
* the prediction list flags predFlagLA and predFlagLB.

The motion vectors mvLA and mvLB, the reference indices refIdxLA and refIdxLB and the prediction flags predFlagLA and predFlagLB are derived by the following ordered steps:

1. The derivation process for merging candidates from neighbouring coding units as specified in clause 8.5.2.3 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 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 reference indices for the temporal merging candidate, refIdxLXColC0 and refIdxLXColC1, with X being 0 or 1, are set equal to 0.
3. The derivation process for temporal triangle merging candidates as specified in clause 8.5.4.3 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and the variables refIdxL0ColC0 and refIdxL0ColC1 as inputs, and the output being the availability flag availableFlagL0ColC0 and availableFlagL0ColC1, and the temporal motion vector mvL0ColC0 and mvL0ColC1. The variables availableFlagColC0, availableFlagColC1, predFlagL0ColC0, predFlagL0ColC1, predFlagL1ColC0 and predFlagL1ColC1 are derived as follows:

availableFlagColC0 = availableFlagL0ColC0 (8‑481)

predFlagL0ColC0 = availableFlagL0ColC0 (8‑482)

predFlagL1ColC0 = 0 (8‑483)

availableFlagColC1 = availableFlagL0ColC1 (8‑484)

predFlagL0ColC1 = availableFlagL0ColC1 (8‑485)

predFlagL1ColC1 = 0 (8‑486)

1. When tile\_group\_type is equal to B, the derivation process for temporal triangle merging candidates as specified in clause 8.5.4.3 is invoked with the luma location ( xCb, yCb ), the the luma coding block width cbWidth, the luma coding block height cbHeight and the variables refIdxL1ColC0 and refIdxL1ColC1 as inputs, and the output being the availability flag availableFlagL1ColC0 and availableFlagL1ColC1, and the temporal motion vector mvL1ColC0 and mvL1ColC1. The variables availableFlagColC0, availableFlagColC1, predFlagL1ColC0 and predFlagL1ColC1 are derived as follows:

availableFlagColC0 = availableFlagL0ColC0  | |  availableFlagL1ColC0 (8‑487)

predFlagL1ColC0 = availableFlagL1ColC0 (8‑488)

availableFlagColC1 = availableFlagL0ColC1  | |  availableFlagL1ColC1 (8‑489)

predFlagL1ColC1 = availableFlagL1ColC1 (8‑490)

1. The derivation process for uni-prediction triangle merging candidates specified in clause 8.5.4.4 is invoked with the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1, availableFlagB2, availableFlagColC0 and availableFlagColC1, the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1, refIdxLXB2, refIdxLXColC0 and refIdxLXColC1, the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1, predFlagLXB2, predFlagLXColC0 and predFlagLXColC1, the motion vectors mvLXA0, mvLXA1, mvLXB0, mvLXB1, mvLXB2, mvLXColC0 and mvLXColC1, with X being 0 or 1 as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices refIdxL0uniCandk and refIdxL1uniCandk, the prediction list utilization flags predFlagL0uniCandk and predFlagL1uniCandk and the motion vectors mvL0uniCandk and mvL1uniCandk of every candidate uniCandk in the merging candidate list mergeCandList. When numCurrMergeCand is greater than 0, k ranges from 0 to numCurrMergeCand − 1, inclusive. The variable numOrigMergeCand is set equal to numCurrMergeCand.
2. The derivation process for zero motion vector triangle merging candidates specified in clause 8.5.4.6 is invoked with the 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 ). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand − 1, inclusive.
3. The following applies:
   * The variables m and n, being the candidates in the merging candidate list mergeCandList, are derived using merge\_triangle\_idx0[ xCb ][ yCb ] and as merge\_triangle\_idx1[ xCb ][ yCb ] follows:

m = merge\_triangle\_idx0[ xCb ][ yCb ] (8‑491)

n = merge\_triangle\_idx1[ xCb ][ yCb ] + ( merge\_triangle\_idx1[ xCb ][ yCb ] >= m ) ? 1 : 0 (8‑492)

* + The following assignments are made with M and N being the candidates at position m and n in the merging candidate list mergeCandList ( M = mergeCandList[ m ] and N = mergeCandList[ n ] ):

predFlagLA = ( predFlagL0M = = 1 ) ? 0 : 1 (8‑493)

predFlagLB = ( predFlagL0N = = 1 ) ? 0 : 1 (8‑494)

refIdxLA = ( predFlagL0M = = 1 ) ? refIdxL0M : refIdxL1M (8‑495)

refIdxLB = ( predFlagL0N = = 1 ) ? refIdxL0N : refIdxL1N (8‑496)

mvLA[ 0 ] = ( predFlagL0M = = 1 ) ? mvL0M[ 0 ] : mvL1M[ 0 ] (8‑497)

mvLA[ 1 ] = ( predFlagL0M = = 1 ) ? mvL0M[ 1 ] : mvL1M[ 1 ] (8‑498)

mvLB[ 0 ] = ( predFlagL0N = = 1 ) ? mvL0N[ 0 ] : mvL1N[ 0 ] (8‑499)

mvLB[ 1 ] = ( predFlagL0N = = 1 ) ? mvL0N[ 1 ] : mvL1N[ 1 ] (8‑500)

#### Derivation process for temporal triangle 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,
* reference indices refIdxLXC0 and refIdxLXC1, with X being 0 or 1.

Outputs of this process are:

* the motion vector prediction mvLXColC0 and mvLXColC1 in 1/16 fractional-sample accuracy,
* the availability flag availableFlagLXColC0 and availableFlagLXColC1.

The variable currCb specifies the current luma coding block at luma location ( xCb, yCb ).

The variables mvLXColC0, mvLXColC1, availableFlagLXColC0 and availableFlagLXColC1 are derived as follows:

* If tile\_group\_temporal\_mvp\_enabled\_flag is equal to 0, both components of mvLXColC0 and mvLXColC1 are set equal to 0 and availableFlagLXColC0 and availableFlagLXColC1 are set equal to 0.
* Otherwise (tile\_group\_temporal\_mvp\_enabled\_flag is equal to 1), the following ordered steps apply:

1. The bottom right collocated motion vector mvLXColC0 is derived as follows:

xColBr = xCb + cbWidth (8‑501)

yColBr = yCb + cbHeight (8‑502)

* 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 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLXC0 and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXColC0 and availableFlagLXColC0.
* Otherwise, both components of mvLXColC0 are set equal to 0 and availableFlagLXColC0 is set equal to 0.

1. The central collocated motion vector mvLXColC1 is derived as follows:

xColCtr = xCb + ( cbWidth  >>  1 ) (8‑503)

yColCtr = yCb + ( cbHeight  >>  1 ) (8‑504)

* 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 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLXC1 and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXColC1 and availableFlagLXColC1.

#### Derivation process for uni-prediction triangle merging candidates

Inputs to this process are:

* the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1, availableFlagB2, availableFlagColC0 and availableFlagColC1,
* the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1, refIdxLXB2, refIdxLXColC0 and refIdxLXColC1,
* the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1, predFlagLXB2, predFlagLXColC0 and predFlagLXColC1,
* the motion vectors mvLXA0, mvLXA1, mvLXB0, mvLXB1, mvLXB2, mvLXColC0 and mvLXColC1, with X being 0 or 1.

Outputs of this process are:

* the merging candidate list mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the reference indices refIdxL0uniCandk and refIdxL1uniCandk of every candidate uniCandk included into mergeCandList during the invocation of this process,
* the prediction list utilization flags predFlagL0uniCandk and predFlagL1uniCandk of every candidate uniCandk included into mergeCandList during the invocation of this process,
* the motion vectors in 1/16 fractional-sample accuracy mvL0uniCandk and mvL1uniCandk of every candidate uniCandk included into mergeCandList during the invocation of this process.

The following steps are performed:

* The variable currPic specifies the current picture.
* The two components of the variable zeroMv, representing a motion vector, are derived as follows:

zeroMv[ 0 ] = 0 (8‑505)

zeroMv[ 1 ] = 0 (8‑506)

* The variable numCurrMergeCand is set equal to 0.
* For N being equal to each of A1, B1, B0, A0, B2, ColC0 and ColC1 the following applies:
  + When numCurrMergeCand is greater than 0, the flags uniqueFlagm with m = 0..numCurrMergeCand − 1 are derived by invoking the comparison process for uni-prediction luma motion vectors specified in clause 8.5.4.5 with the reference indices refIdxLXN and refIdxLXuniCandm, the prediction list utilization flags predFlagLXN and predFlagLXuniCandm, the motion vectors mvLXN and mvLXuniCandm with X being 0 or 1, and the prediction list flag predFlag set to 0 as inputs.
  + For k equal to numCurrMergeCand, when all of the following conditions are true:
  + numCurrMergeCand is less than 5
  + availableFlagN is equal to 1 and predFlagL0N + predFlagL1N is equal to 1
  + the flags uniqueFlagm are all equal to 1 for m = 0..k − 1 when k > 0

the candidate uniCandk is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to uniCandk, and the reference indices, the prediction list utilization flags, the motion vectors of uniCandk and the new value of numCurrMergeCand are derived as follows:

refIdxL0uniCandk = refIdxL0N (8‑507)

refIdxL1uniCandk = refIdxL1N (8‑508)

predFlagL0uniCandk = predFlagL0N (8‑509)

predFlagL1uniCandk = predFlagL1N (8‑510)

mvL0uniCandk = mvL0N (8‑511)

mvL1uniCandk = mvL1N (8‑512)

numCurrMergeCand = numCurrMergeCand + 1 (8‑513)

* For N being equal to each of A1, B1, B0, A0, B2, ColC0 and ColC1 the following applies:
  + When numCurrMergeCand is greater than 0, the flags uniqueFlagm with m = 0..numCurrMergeCand − 1 are derived by invoking the comparison process for uni-prediction luma motion vectors specified in clause 8.5.4.5 with the reference indices refIdxLXN and refIdxLXuniCandm, the prediction list utilization flags predFlagLXN and predFlagLXuniCandm, the motion vectors mvLXN and mvLXuniCandm with X being 0 or 1, and the prediction list flag predFlag set to 0 as inputs.
  + For k equal to numCurrMergeCand, when all of the following conditions are true:
  + numCurrMergeCand is less than 5
  + availableFlagN is equal to 1 and predFlagL0N + predFlagL1N is equal to 2
  + the flags uniqueFlagm are all equal to 1 for m = 0..k − 1 when k > 0

the candidate uniCandk is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to uniCandk, and the reference indices, the prediction list utilization flags, the motion vectors of uniCandk and the new value of numCurrMergeCand are derived as follows:

refIdxL0uniCandk = refIdxL0N (8‑514)

refIdxL1uniCandk = −1 (8‑515)

predFlagL0uniCandk = predFlagL0N (8‑516)

predFlagL1uniCandk = 0 (8‑517)

mvL0uniCandk = mvL0N (8‑518)

mvL1uniCandk = zeroMv (8‑519)

numCurrMergeCand = numCurrMergeCand + 1 (8‑520)

* For N being equal to each of A1, B1, B0, A0, B2, ColC0 and ColC1 the following applies:
  + When numCurrMergeCand is greater than 0, the flags uniqueFlagm with m = 0 .. numCurrMergeCand − 1 are derived by invoking the comparison process for uni-prediction luma motion vectors specified in clause 8.5.4.5 with the reference indices refIdxLXN and refIdxLXuniCandm, the prediction list utilization flags predFlagLXN and predFlagLXuniCandm, the motion vectors mvLXN and mvLXuniCandm with X being 0 or 1, and the prediction list flag predFlag set to 1 as inputs.
  + For k equal to numCurrMergeCand, when all of the following conditions are true:
  + numCurrMergeCand is less than 5
  + availableFlagN is equal to 1 and predFlagL0N + predFlagL1N is equal to 2
  + the flags uniqueFlagm are all equal to 1 for m = 0 .. k − 1 when k > 0

the candidate uniCandk with k equal to numCurrMergeCand is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to uniCandk, and the reference indices, the prediction list utilization flags, the motion vectors of uniCandk and the new value of numCurrMergeCand are derived as follows:

refIdxL0uniCandk = −1 (8‑521)

refIdxL1uniCandk = refIdxL1N (8‑522)

predFlagL0uniCandk = 0 (8‑523)

predFlagL1uniCandk = predFlagL1N (8‑524)

mvL0uniCandk = zeroMv (8‑525)

mvL1uniCandk = mvL1N (8‑526)

numCurrMergeCand = numCurrMergeCand + 1 (8‑527)

* For N being equal to each of A1, B1, B0, A0, B2, ColC0 and ColC1 the following applies:
  + The variable tempMv is derived as follows:
  + If DiffPicOrderCnt( RefPicList[ 0 ][ refIdxL0N ], RefPicList[ 1 ][ refIdxL1N ] ) is not equal to 0:

tempMv[ 0 ] = ( mvL0N[ 0 ] + Clip3( −131072, 131071, Sign( distScaleFactor \* mvL1N[ 0 ] ) \*  (8‑528)  
 ( ( Abs( distScaleFactor \* mvL1N[ 0 ] ) + 127 )  >>  8 ) ) + 1 ) >> 1

tempMv[ 1 ] = ( mvL0N[ 1 ] + Clip3( −131072, 131071, Sign( distScaleFactor \* mvL1N[ 1 ] ) \*  (8‑529)  
 ( ( Abs( distScaleFactor \* mvL1N[ 1 ] ) + 127 )  >>  8 ) ) + 1 ) >> 1

where distScaleFactor is derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0N ]  ) ) (8‑530)

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑531)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicList[ 1 ][ refIdxL1N ] ) ) (8‑532)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑533)

* + Otherwise ( DiffPicOrderCnt( RefPicList[ 0 ][ refIdxL0N ], RefPicList[ 1 ][ refIdxL1N ] ) is equal to 0 ):

tempMv[ 0 ] = ( ( mvL0N[ 0 ] + mvL1N[ 0 ] + 1 ) >> 1 ) (8‑534)

tempMv[ 1 ] = ( ( mvL0N[ 1 ] + mvL1N[ 1 ] + 1 ) >> 1 ) (8‑535)

* + When numCurrMergeCand is greater than 0, the flags uniqueFlagm with m = 0 .. numCurrMergeCand − 1 are derived by invoking the comparison process for uni-prediction luma motion vectors specified in clause 8.5.4.5 with the reference indices refIdxLXN and refIdxLXuniCandm, the prediction list utilization flags predFlagLXN and predFlagLXuniCandm, the motion vectors mvLXN set equal to tempMv and mvLXuniCandm with X being 0 or 1, and the prediction list flag predFlag set to 0 as inputs.
  + For k equal to numCurrMergeCand, when all of the following conditions are true:
  + numCurrMergeCand is less than 5
  + availableFlagN is equal to 1 and predFlagL0N + predFlagL1N is equal to 2
  + the flags uniqueFlagm are all equal to 1 for m = 0..k − 1 when k > 0

the candidate uniCandk is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to uniCandk, and the reference indices, the prediction list utilization flags, the motion vectors of uniCandk and the new value of numCurrMergeCand are derived as follows:

refIdxL0uniCandk = refIdxL0N (8‑536)

refIdxL1uniCandk = −1 (8‑537)

predFlagL0uniCandk = 1 (8‑538)

predFlagL1uniCandk = 0 (8‑539)

mvL0uniCandk = tempMv (8‑540)

mvL1uniCandk = zeroMv (8‑541)

numCurrMergeCand = numCurrMergeCand + 1 (8‑542)

#### Comparison process for uni-prediction luma motion vectors

Inputs to this process, with X being equal to 0 or 1, are:

* the reference indices refIdxLXN and refIdxLXuniCandm,
* the prediction list utilization flags predFlagLXN and predFlagLXuniCandm,
* the motion vectors mvLXN and mvLXuniCandm,
* the prediction list flag predFlag.

Output of this process is:

* the flag uniqueFlagm.

The prediction list flags predFlagN and predFlagCand are derived as follows:

predFlagN = ( predFlagL0N + predFlagL1N = = 2 ) ? predFlag : ( ( predFlagL0N = = 1 ) ? 0 : 1 ) (8‑543)

predFlagCand = ( predFlagL0uniCandm = = 1 ) ? 0 : 1 (8‑544)

The variables refPicPocN and refPicPocCand are derived as follows:

refPicPocN = ( predFlagN = = 0 ) ? RefPicList[ 0 ][ refIdxL0N ] : RefPicList[ 1 ][ refIdxL1N ] (8‑545)

refPicPocCand = ( predFlagCand = = 0 ) ? RefPicList[ 0 ][ refIdxL0uniCandm ] (8‑546)  
 : RefPicList[ 1 ][ refIdxL1uniCandm ]

The motion vectors mvN and mvCand are derived as follows:

mvN = ( predFlagN = = 0 ) ? mvL0N : mvL1N (8‑547)

mvCand = ( predFlagCand = = 0 ) ? mvL0uniCandm : mvL1uniCandm (8‑548)

The flag uniqueFlagm is derived as follows:

uniqueFlagm = ( refPicPocN != refPicPocCand | | mvN != mvCand ) ? 1 : 0 (8‑549)

#### Derivation process for zero motion vector triangle merging candidates

Inputs to this process are:

* a merging candidate list 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 variables numRefIdx is derived as follows:

* numRefIdx is set equal to Min( NumRefIdxActive[ 0 ], NumRefIdxActive[ 1 ] ).

When numCurrMergeCand is less than 5, the variable numInputMergeCand is set equal to numCurrMergeCand, the variables zeroIdx is set equal to 0 and the following ordered steps are repeated until numCurrMergeCand is equal to 5:

1. 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 (8‑550)

refIdxL1zeroCandm = −1 (8‑551)

predFlagL0zeroCandm = 1 (8‑552)

predFlagL1zeroCandm = 0 (8‑553)

mvL0zeroCandm[ 0 ] = 0 (8‑554)

mvL0zeroCandm[ 1 ] = 0 (8‑555)

mvL1zeroCandm[ 0 ] = 0 (8‑556)

mvL1zeroCandm[ 1 ] = 0 (8‑557)

numCurrMergeCand = numCurrMergeCand + 1 (8‑558)

1. 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 = −1 (8‑559)

refIdxL1zeroCandm = zeroIdx (8‑560)

predFlagL0zeroCandm = 0 (8‑561)

predFlagL1zeroCandm = 1 (8‑562)

mvL0zeroCandm[ 0 ] = 0 (8‑563)

mvL0zeroCandm[ 1 ] = 0 (8‑564)

mvL1zeroCandm[ 0 ] = 0 (8‑565)

mvL1zeroCandm[ 1 ] = 0 (8‑566)

numCurrMergeCand = numCurrMergeCand + 1 (8‑567)

1. zeroIdx = ( zeroIdx + 1 ) % numRefIdx

### Derivation process for subblock 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 number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the prediction list utilization flag arrays predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbX − 1,
* the luma subblock motion vector arrays in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the chroma subblock motion vector arrays in 1/32 fractional-sample accuracy mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the bi-prediction weight index gbiIdx.

For the derivation of the variables mvL0[ xSbIdx ][ ySbIdx ], mvL1[ xSbIdx ][ ySbIdx ], mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, numSbX, numSbY, predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the following applies:

* If merge\_subblock\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for motion vectors and reference indices in subblock merge mode as specified in 8.5.5.2 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, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the reference indices refIdxL0, refIdxL1, the prediction list utilization flag arrays predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the luma subblock motion vector arrays mvL0[ xSbIdx ][ ySbIdx ] and mvL0[ xSbIdx ][ ySbIdx ], and the chroma subblock motion vector arrays mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], with xSbIdx = 0.. numSbX − 1, ySbIdx = 0 .. numSbY − 1, and the bi-prediction weight index gbiIdx as outputs.
* Otherwise (merge\_subblock\_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:
* For the derivation of the number of control point motion vectors numCpMv, the control point motion vectors cpMvLX[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1, refIdxLX, predFlagLX[ 0 ][ 0 ], the following applies:

1. The number of control point motion vectors numCpMv is set equal to MotionModelIdc[ xCb ][ yCb ] + 1.
2. 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 ] (8‑568)

predFlagLX[ 0 ][ 0 ] = 1 (8‑569)

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

refIdxLX = −1 (8‑570)

predFlagLX[ 0 ][ 0 ] = 0 (8‑571)

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 ] (8‑572)

mvdCpLX[ cpIdx ][ 1 ] = MvdCpLX[ xCb ][ yCb ][ cpIdx ][ 1 ] (8‑573)

1. When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in clause 8.5.5.7 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[ 0 ][ 0 ] 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 (8‑574)

cpMvLX[ cpIdx ][ 0 ] = (uLX[ cpIdx ][ 0 ] >= 217 ) ? (uLX[ cpIdx ][ 0 ] − 218 ) :   
 uLX[ cpIdx ][ 0 ] (8‑575)

uLX[ cpIdx ][ 1 ] = ( mvpCpLX[ cpIdx ][ 1 ] + mvdCpLX[ cpIdx ][ 1 ] + 218 ) % 218 (8‑576)

cpMvLX[ cpIdx ][ 1 ] = (uLX[ cpIdx ][ 1 ] >= 217 ) ? (uLX[ cpIdx ][ 1 ] − 218 ) :   
 uLX[ cpIdx ][ 1 ] (8‑577)

* The variables numSbX and numSbY are derived as follows:

numSbX = ( cbWidth >> 2 ) (8‑578)

numSbY = ( cbHeight >> 2 ) (8‑579)

* For xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, the following applies:

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLX[ 0 ][ 0 ] (8‑580)

* When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.5.5.9 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 cpMvLX[ cpIdx ] with cpIdx being 0..2, the reference index refIdxLX and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY as inputs, the luma motion vector array mvLX[ xSbIdx ][ ySbIdx ] and the chroma motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 as outputs.
* The bi-prediction weight index gbiIdx is set equal to gbi\_idx[ xCb ][ yCb ].

#### Derivation process for motion vectors and reference indices in subblock 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 luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flag arrays predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ],
* the luma subblock motion vector arrays in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the chroma subblock motion vector arrays in 1/32 fractional-sample accuracy mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the bi-prediction weight index gbiIdx.

The variables numSbX, numSbY and the subblock merging candidate list, subblockMergeCandList are derived by the following ordered steps:

1. When sps\_sbtmvp\_enabled\_flag is equal to 1, the following applies:

* 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, and gbiIdxA1 is set equal to 0.
    - Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvLXA1 = MvLX[ xNbA1 ][ yNbA1 ] (8‑581)

refIdxLXA1 = RefIdxLX[ xNbA1 ][ yNbA1 ] (8‑582)

predFlagLXA1 = PredFlagLX[ xNbA1 ][ yNbA1 ] (8‑583)

* The derivation process for subblock-based temporal merging candidates as specified in clause 8.5.5.3 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight , the availability flag availableFlagA1, the reference index refIdxLXA1, the prediction list utilization flag predFlagLXA1, and the motion vector mvLXA1 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.

1. When sps\_affine\_enabled\_flag is equal to 1, the sample locations ( xNbA0, yNbA0 ), ( xNbA1, yNbA1 ), ( xNbA2, yNbA2 ), ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ), ( xNbB2, yNbB2 ), ( xNbB3, yNbB3 ), and the variables numSbX and numSbY are derived as follows:

( xA0, yA0 ) = ( xCb − 1, yCb + cbHeight ) (8‑584)

( xA1, yA1 ) = ( xCb − 1, yCb + cbHeight − 1 ) (8‑585)

( xA2, yA2 ) = ( xCb − 1, yCb ) (8‑586)

( xB0, yB0 ) = ( xCb + cbWidth , yCb − 1 ) (8‑587)

( xB1, yB1 ) = ( xCb + cbWidth − 1, yCb − 1 ) (8‑588)

( xB2, yB2 ) = ( xCb − 1, yCb − 1 ) (8‑589)

( xB3, yB3 ) = ( xCb, yCb − 1 ) (8‑590)

numSbX = cbWidth >> 2 (8‑591)

numSbY = cbHeight >> 2 (8‑592)

1. When sps\_affine\_enabled\_flag is equal to 1, the variable availableFlagA is set equal to FALSE and the following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* 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 ( xNbAk, yNbAk ) as inputs, and the output is assigned to the block availability flag availableAk.
* When availableAk is equal to TRUE and MotionModelIdc[ xNbAk ][ yNbAk ] is greater than 0 and availableFlagA is equal to FALSE, the following applies:
  + - The variable availableFlagA is set equal to TRUE, motionModelIdcA is set equal to MotionModelIdc[ xNbAk ][ yNbAk ], ( xNb, yNb ) is set equal to ( CbPosX[ xNbAk ][ yNbAk ], CbPosY[ xNbAk ][ yNbAk ] ), nbW is set equal to CbWidth[ xNbAk ][ yNbAk ], nbH is set equal to CbHeight[ xNbAk ][ yNbAk ], numCpMv is set equal to MotionModelIdc[ xNbAk ][ yNbAk ] + 1, and gbiIdxA is set equal to GbiIdx[ xNbAk ][ yNbAk ].
    - For X being replaced by either 0 or 1, the following applies:
      * When PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1, the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 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 ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvLXA[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

predFlagLXA = PredFlagLX[ xNbAk ][ yNbAk ] (8‑593)

refIdxLXA = RefIdxLX[ xNbAk ][ yNbAk ] (8‑594)

1. When sps\_affine\_enabled\_flag is equal to 1, the variable availableFlagB is set equal to FALSE and 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 MotionModelIdc[ xNbBk ][ yNbBk ] is greater than 0 and availableFlagB is equal to FALSE, the following applies:
  + - The variable availableFlagB is set equal to TRUE, motionModelIdcB is set equal to MotionModelIdc[ xNbBk ][ yNbBk ], ( xNb, yNb ) is set equal to ( CbPosX[ xNbAB ][ yNbBk ], CbPosY[ xNbBk ][ yNbBk ] ), nbW is set equal to CbWidth[ xNbBk ][ yNbBk ], nbH is set equal to CbHeight[ xNbBk ][ yNbBk ], numCpMv is set equal to MotionModelIdc[ xNbBk ][ yNbBk ] + 1, and gbiIdxB is set equal to GbiIdx[ xNbBk ][ yNbBk ].
    - For X being replaced by either 0 or 1, the following applies:
      * When PredFlagLX[ xNbBk ][ yNbBk ] is equal to TRUE, the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 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 ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvLXB[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

predFlagLXB = PredFlagLX[ xNbBk ][ yNbBk ] (8‑595)

refIdxLXB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑596)

1. When sps\_affine\_enabled\_flag is equal to 1, the derivation process for constructed affine control point motion vector merging candidates as specified in clause 8.5.5.6 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the availability flags availableA0, availableA1, availableA2, availableB0, availableB1, availableB2, availableB3 as inputs, and the availability flags availableFlagConstK, the reference indices refIdxLXConstK, prediction list utilization flags predFlagLXConstK, motion model indices motionModelIdcConstK and cpMvpLXConstK[ cpIdx ] with X being 0 or 1, K = 1..6, cpIdx = 0..2 as outputs and gbiIdxConstK is set equal to 0 with K = 1..6..
2. The initial subblock merging candidate list, subblockMergeCandList, is constructed as follows:

i = 0  
if( availableFlagSbCol )  
 subblockMergeCandList[ i++ ] = SbCol  
if( availableFlagA && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = A  
if( availableFlagB && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Bif( availableFlagConst1 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const1 (8‑597)if( availableFlagConst2 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const2if( availableFlagConst3 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const3  
if( availableFlagConst4 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const4if( availableFlagConst5 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const5  
if( availableFlagConst6 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const6

1. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the subblockMergeCandList.
2. When numCurrMergeCand is less than MaxNumSubblockMergeCand, the following is repeated until numCurrMrgeCand is equal to MaxNumSubblockMergeCand, with mvZero[0] and mvZero[1] both being equal to 0:

* The reference indices, the prediction list utilization flags and the motion vectors of zeroCandm with m equal to ( numCurrMergeCand − numOrigMergeCand ) are derived as follows:

refIdxL0ZeroCandm = 0 (8‑598)

predFlagL0ZeroCandm = 1 (8‑599)

cpMvL0ZeroCandm[ 0 ] = mvZero (8‑600)

cpMvL0ZeroCandm[ 1 ] = mvZero (8‑601)

cpMvL0ZeroCandm[ 2 ] = mvZero (8‑602)

refIdxL1ZeroCandm = ( tile\_group\_type = = B ) ? 0 : −1 (8‑603)

predFlagL1ZeroCandm = ( tile\_group\_type = = B ) ? 1 : 0 (8‑604)

cpMvL1ZeroCandm[ 0 ] = mvZero (8‑605)

cpMvL1ZeroCandm[ 1 ] = mvZero (8‑606)

cpMvL1ZeroCandm[ 2 ] = mvZero (8‑607)

motionModelIdcZeroCandm = 1 (8‑608)

gbiIdxZeroCandm = 0 (8‑609)

* The candidate zeroCandm with m equal to ( numCurrMergeCand − numOrigMergeCand ) is added at the end of subblockMergeCandList and numCurrMergeCand is incremented by 1 as follows:

subblockMergeCandList[ numCurrMergeCand++ ] =  zeroCandm (8‑610)

The variables refIdxL0, refIdxL1, predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], mvL0[ xSbIdx ][ ySbIdx ], mvL1[ xSbIdx ][ ySbIdx ], mvCL0[ xSbIdx ][ ySbIdx ], and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1 are derived as follows:

* If subblockMergeCandList[ merge\_subblock\_idx[ xCb ][ yCb ] ] is equal to SbCol, the bi-prediction weight index gbiIdx is set equal to 0 and the following applies with X being 0 or 1:

refIdxLX = refIdxLXSbCol (8‑611)

* + - * For xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, the following applies:

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLXSbCol[ xSbIdx ][ ySbIdx ] (8‑612)

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = mvLXSbCol[ xSbIdx ][ ySbIdx ][ 0 ] (8‑613)

mvLX[ xSbIdx ][  ySbIdx ][ 1 ] = mvLXSbCol[ xSbIdx ][ ySbIdx ][ 1 ] (8‑614)

* + - * When predFlagLX[ xSbIdx ][ ySbIdx ], is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLX[ xSbIdx ][ ySbIdx ] and refIdxLX as inputs, and the output being mvCLX[ xSbIdx ][ ySbIdx ].
      * The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[ x ][ y ] = 0  (8‑615)

* Otherwise (subblockMergeCandList[ merge\_subblock\_idx[ xCb ][ yCb ] ] is not equal to SbCol), the following applies with X being 0 or 1:
  + - * The following assignments are made with N being the candidate at position merge\_subblock\_idx[ xCb ][ yCb ] in the subblock merging candidate list subblockMergeCandList ( N = subblockMergeCandList[ merge\_subblock\_idx[ xCb ][ yCb ] ] ):

refIdxLX = refIdxLXN (8‑616)

predFlagLX[ 0][ 0 ] = predFlagLXN (8‑617)

cpMvLX[ 0 ] = cpMvLXN[ 0 ] (8‑618)

cpMvLX[ 1 ] = cpMvLXN[ 1 ] (8‑619)

cpMvLX[ 2 ] = cpMvLXN[ 2 ] (8‑620)

numCpMv = motionModelIdxN + 1 (8‑621)

gbiIdx = gbiIdxN (8‑622)

* + - * For xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, the following applies:

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLX[ 0 ][ 0 ] (8‑623)

* + - * When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.5.5.9 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 cpMvLX[ cpIdx ] with cpIdx being 0..2, and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY as inputs, the luma subblock motion vector array mvLX[ xSbIdx ][ ySbIdx ] and the chroma subblock motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 as outputs.
      * The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[ x ][ y ] = numCpMv − 1 (8‑624)

#### 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 flag availableFlagA1 of the neighbouring coding unit,
* the reference index refIdxLXA1 of the neighbouring coding unit,
* the prediction list utilization flag predFlagLXA1 of the neighbouring coding unit,
* the motion vector in 1/16 fractional-sample accuracy mvLXA1 of the neighbouring coding unit.

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 one or more of the following conditions is true, availableFlagSbCol is set equal to 0.
* tile\_group\_temporal\_mvp\_enabled\_flag is equal to 0.
* sps\_sbtmvp\_flag is equal to 0.
* cbWidth is less than 8.
* cbHeight is less than 8.
* 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 (8‑625)

yCtb = ( yCb  >>  CtuLog2Size )  <<  CtuLog2Size (8‑626)

xCtr = xCb + ( cbWidth / 2 ) (8‑627)

yCtr = yCb + ( cbHeight / 2 ) (8‑628)

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 8.5.5.4 is invoked with the location ( xCtb, yCtb ), the location ( xColCtrCb, yColCtrCb ), the availability flag availableFlagA1, and the prediction list utilization flag predFlagLXA1, and the reference index refIdxLXA1, and the motion vector mvLXA1, 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  =  cbWidth >> 3 (8‑629)

numSbY  =  cbHeight >> 3 (8‑630)

sbWidth  =  cbWidth / numSbX (8‑631)

sbHeight  =  cbHeight / numSbY (8‑632)

refIdxLXSbCol  =  0 (8‑633)

* 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 (8‑634)

ySb  =  yCb + ySbIdx \* sbHeight (8‑635)

* The location ( xColSb, yColSb ) of the collocated subblock inside ColPic is derived as follows.

xColSb = Clip3( xCtb,   
 Min( CurPicWidthInSamplesY − 1, xCtb + ( 1  <<  CtbLog2SizeY ) + 3 ), (8‑636)  
 xSb + ( tempMv[0]  >>  4 ) )

yColSb = Clip3( yCtb,   
 Min( CurPicHeightInSamplesY − 1, yCtb + ( 1  <<  CtbLog2SizeY ) − 1 ), (8‑637)  
 ySb + ( tempMv[1]  >>  4 ) )

* The variable currCb specifies the luma coding block covering the current coding subblock 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 8.5.2.12 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 8.5.2.12 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 for X being 0 and 1:

mvLXSbCol[ xSbIdx ][ ySbIdx ] = ctrMvLX (8‑638)

predFlagLXSbCol[ xSbIdx ][ ySbIdx ] = ctrPredFlagLX (8‑639)

#### 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 flag availableFlagA1 of the neighbouring coding unit,
* the reference index refIdxLXA1 of the neighbouring coding unit,
* the prediction list utilization flag predFlagLXA1 of the neighbouring coding unit,
* the motion vector in 1/16 fractional-sample accuracy mvLXA1 of the neighbouring coding unit.

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 variable tempMv is set as follows:

tempMv[ 0 ] = 0 (8‑640)

tempMv[ 1 ] = 0 (8‑641)

The variable currPic specifies the current picture.

When availableFlagA1 is equal to TRUE, the following applies:

* If all of the following conditions are true, tempMV is set equal to mvL1A1:
  + predFlagL1A1N is equal to 1,
  + DiffPicOrderCnt(ColPic, RefPicList[ 1 ][refIdxL1A1]) is equal to 0,
  + DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current tile group,
  + tile\_group\_type is equal to B,
  + collocated\_from\_l0\_flag is equal to 0.
* Otherwise if all of the following conditions are true, tempMV is set equal to mvL0A1:
  + predFlagL0A1 is equal to 1,
  + DiffPicOrderCnt(ColPic, RefPicList[ 0 ][refIdxL0A1]) is equal to 0.

The location ( xColCb, yColCb ) of the collocated block inside ColPic is derived as follows.

xColCb = Clip3( xCtb,   
 Min( CurPicWidthInSamplesY − 1, xCtb + ( 1  <<  CtbLog2SizeY ) + 3 ), (8‑642)  
 xColCtrCb + ( tempMv[0]  >>  4 ) )

yColCb = Clip3( yCtb,   
 Min( CurPicHeightInSamplesY − 1, yCtb + ( 1  <<  CtbLog2SizeY ) − 1 ), (8‑643)  
 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  8.5.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL0, and sbFlag set equal to 1 as inputs and the output being assigned to ctrMvL0 and ctrPredFlagL0.
  + The derivation process for temporal motion vector prediction in subclause  8.5.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL1, and sbFlag set equal to 1 as inputs and the output being assigned to ctrMvL1 and ctrPredFlagL1.
* Otherwise, the following applies:

ctrPredFlagL0 = 0 (8‑644)

ctrPredFlagL1 = 0 (8‑645)

#### 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 variable isCTUboundary is derived as follows:

* If all the following conditions are true, isCTUboundary is set equal to TRUE:
* ( ( yNb + nNbH ) % CtbSizeY ) is equal to 0
* yNb + nNbH is equal to yCb
* Otherwise, isCTUboundary is set equal to FALSE.

The variables log2NbW and log2NbH are derived as follows:

log2NbW = Log2( nNbW ) (8‑646)

log2NbH = Log2( nNbH ) (8‑647)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

* If isCTUboundary is equal to TRUE, the following applies:

mvScaleHor = MvLX[ xNb ][ yNb + nNbH − 1 ][ 0 ] << 7 (8‑648)

mvScaleVer = MvLX[ xNb ][ yNb + nNbH − 1 ][ 1 ] << 7 (8‑649)

dHorX = ( MvLX[ xNb + nNbW − 1 ][ yNb + nNbH − 1 ][ 0 ] − MvLX[ xNb ][ yNb + nNbH − 1 ][ 0 ] )    
 << ( 7 − log2NbW ) (8‑650)

dVerX = ( MvLX[ xNb + nNbW − 1 ][ yNb + nNbH − 1 ][ 1 ] − MvLX[ xNb ][ yNb + nNbH − 1 ][ 1 ] )    
 << ( 7 − log2NbW ) (8‑651)

* Otherwise (isCTUboundary is equal to FALSE), the following applies:

mvScaleHor = CpMvLX[ xNb ][ yNb ][ 0 ][ 0 ] << 7 (8‑652)

mvScaleVer = CpMvLX[ xNb ][ yNb ][ 0 ][ 1 ] << 7 (8‑653)

dHorX = ( CpMvLX[ xNb + nNbW − 1 ][ yNb ][ 1 ][ 0 ] − CpMvLX[ xNb ][ yNb ][ 0 ][ 0 ] )   
 << ( 7 − log2NbW ) (8‑654)

dVerX = ( CpMvLX[ xNb + nNbW − 1 ][ yNb ][ 1 ][ 1 ] − CpMvLX[ xNb ][ yNb ][ 0 ][ 1 ] )   
 << ( 7 − log2NbW ) (8‑655)

The variables dHorY and dVerY are derived as follows:

* If isCTUboundary is equal to FALSE and MotionModelIdc[ xNb ][ yNb ] is equal to 2, the following applies:

dHorY = ( CpMvLX[ xNb ][ yNb + nNbH − 1 ][ 2 ][ 0 ] − CpMvLX[ xNb ][ yNb ][ 2 ][ 0 ] )   
 << ( 7 − log2NbH ) (8‑656)

dVerY = ( CpMvLX[ xNb ][ yNb + nNbH − 1 ][ 2 ][ 1 ] − CpMvLX[ xNb ][ yNb ][ 2 ][ 1 ] )   
 << ( 7 − log2NbH ) (8‑657)

* Otherwise (isCTUboundary is equal to TRUE or MotionModelIdc[ xNb ][ yNb ] is equal to 1), the following applies,

dHorY = − dVerX (8‑658)

dVerY = dHorX (8‑659)

The luma affine control point motion vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 are derived as follows:

* When isCTUboundary is equal to TRUE, yNb is set equal to yCb.
* 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 ) ) (8‑660)

cpMvLX[ 0 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb − xNb ) + dVerY \* ( yCb − yNb ) ) 8‑661)

cpMvLX[ 1 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb + cbWidth − xNb ) + dHorY \* ( yCb − yNb ) ) (8‑662)

cpMvLX[ 1 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb + cbWidth − xNb ) + dVerY \* ( yCb − yNb ) ) (8‑663)

* 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 ) ) (8‑664)

cpMvLX[ 2 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb − xNb ) + dVerY \* ( yCb + cbHeight − yNb ) ) (8‑665)

* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked 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.
* The motion vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 are clipped as follows:

cpMvLX[ cpIdx ][ 0 ] = Clip3( −217, 217 − 1, cpMvLX[ cpIdx ][ 0 ] ) (8‑666)

cpMvLX[ cpIdx ][ 1 ] = Clip3( −217, 217 − 1, cpMvLX[ cpIdx ][ 1 ] ) (8‑667)

#### Derivation process for constructed affine control point motion vector merging 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 availability flags availableA0, availableA1, availableA2, availableB0, availableB1, availableB2, availableB3,
* the sample locations ( xNbA0, yNbA0 ), ( xNbA1, yNbA1 ), ( xNbA2, yNbA2 ), ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ), ( xNbB2, yNbB2 ) and ( xNbB3, yNbB3 ).

Output of this process are:

* the availability flag of the constructed affine control point motion vector merging candidiates availableFlagConstK, with K = 1..6,
* the reference indices refIdxLXConstK, with K = 1..6, X being 0 or 1,
* the prediction list utilization flags predFlagLXConstK, with K = 1..6, X being 0 or 1,
* the affine motion model indices motionModelIdcConstK, with K = 1..6,
* the constructed affine control point motion vectors cpMvLXConstK[ cpIdx ] with cpIdx = 0..2, K = 1..6 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLXCorner[ 0 ], reference index refIdxLXCorner[ 0 ], prediction list utilization flag predFlagLXCorner[ 0 ] and the availability flag availableFlagCorner[ 0 ] with X being 0 and 1 are derived as follows:

* The availability flag availableFlagCorner[ 0 ] is set equal to FALSE.
* The following applies for ( xNbTL, yNbTL ) with TL being replaced by B2, B3, and A2:
  + - * When availableTL is equal to TRUE and availableFlagCorner[ 0 ] is equal to FALSE, the following applies with X being 0 and 1:

refIdxLXCorner[ 0 ] = RefIdxLX[ xNbTL ][ yNbTL ] (8‑668)

predFlagLXCorner[ 0 ] = PredFlagLX[ xNbTL ][ yNbTL ] (8‑669)

cpMvLXCorner[ 0 ] = MvLX[ xNbTL ][ yNbTL ] (8‑670)

availableFlagCorner[ 0 ] = TRUE (8‑671)

The second (top-right) control point motion vector cpMvLXCorner[ 1 ], reference index refIdxLXCorner[ 1 ], prediction list utilization flag predFlagLXCorner[ 1 ] and the availability flag availableFlagCorner[ 1 ] with X being 0 and 1 are derived as follows

* The availability flag availableFlagCorner[ 1 ] is set equal to FALSE.
* The following applies for ( xNbTR, yNbTR ) with TR being replaced by B1 and B0:
  + - * When availableTR is equal to TRUE and availableFlagCorner[ 1 ] is equal to FALSE, the following applies with X being 0 and 1:

refIdxLXCorner[ 1 ] = RefIdxLX[ xNbTR ][ yNbTR ] (8‑672)

predFlagLXCorner[ 1 ] = PredFlagLX[ xNbTR ][ yNbTR ] (8‑673)

cpMvLXCorner[ 1 ] = MvLX[ xNbTR ][ yNbTR ] (8‑674)

availableFlagCorner[ 1 ] = TRUE (8‑675)

The third (bottom-left) control point motion vector cpMvLXCorner[ 2 ], reference index refIdxLXCorner[ 2 ], prediction list utilization flag predFlagLXCorner[ 2 ] and the availability flag availableFlagCorner[ 2 ] with X being 0 and 1 are derived as follows:

* The availability flag availableFlagCorner[ 2 ] is set equal to FALSE.
* The following applies for ( xNbBL, yNbBL ) with BL being replaced by A1 and A0:
  + - * When availableBL is equal to TRUE and availableFlagCorner[ 2 ] is equal to FALSE, the following applies with X being 0 and 1:

refIdxLXCorner[ 2 ] = RefIdxLX[ xNbBL ][ yNbBL ] (8‑676)

predFlagLXCorner[ 2 ] = PredFlagLX[ xNbBL ][ yNbBL ] (8‑677)

cpMvLXCorner[ 2 ] = MvLX[ xNbBL ][ yNbBL ] (8‑678)

availableFlagCorner[ 2 ] = TRUE (8‑679)

The fourth (collocated bottom-right) control point motion vector cpMvLXCorner[ 3 ], reference index refIdxLXCorner[ 3 ], prediction list utilization flag predFlagLXCorner[ 3 ] and the availability flag availableFlagCorner[ 3 ] with X being 0 and 1 are derived as follows:

* The reference indices for the temporal merging candidate, refIdxLXCorner[ 3 ], with X being 0 or 1, are set equal to 0.
* The variables mvLXCol and availableFlagLXCol, with X being 0 or 1, are derived as follows:
* If tile\_group\_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 (tile\_group\_temporal\_mvp\_enabled\_flag is equal to 1), the following applies:

xColBr = xCb + cbWidth (8‑680)

yColBr = yCb + cbHeight (8‑681)

* 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 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLXCorner[ 3 ] 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.
* The variables availableFlagCorner[ 3 ], predFlagL0Corner[ 3 ], cpMvL0Corner[ 3 ] and predFlagL1Corner[ 3 ] are derived as follows:

availableFlagCorner[ 3 ] = availableFlagL0Col (8‑682)

predFlagL0Corner[ 3 ] = availableFlagL0Col (8‑683)

cpMvL0Corner[ 3 ] = mvL0Col (8‑684)

predFlagL1Corner[ 3 ] = 0 (8‑685)

* When tile\_group\_type is equal to B, the variables availableFlagCorner[ 3 ], predFlagL1Corner[ 3 ] and cpMvL1Corner[ 3 ] are derived as follows:

availableFlagCorner[ 3 ] = availableFlagL0Col  | |  availableFlagL1Col (8‑686)

predFlagL1Corner[ 3 ] = availableFlagL1Col (8‑687)

cpMvL1Corner[ 3 ] = mvL1Col (8‑688)

When sps\_affine\_type\_flag is equal to 1, the first four constructed affine control point motion vector merging candidates ConstK with K = 1..4 including the availability flags availableFlagConstK, the reference indices refIdxLXConstK, the prediction list utilization flags predFlagLXConstK, the affine motion model indices motionModelIdcConstK, and the constructed affine control point motion vectors cpMvLXConstK[ cpIdx ] with cpIdx = 0..2 and X being 0 or 1 are derived as follows:

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 1 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 1 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 1 ]
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 2 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst1 = 1 (8‑689)

refIdxLXConst1 = refIdxLXCorner[ 0 ] (8‑690)

cpMvLXConst1[ 0 ] = cpMvLXCorner[ 0 ] (8‑691)

cpMvLXConst1[ 1 ] = cpMvLXCorner[ 1 ] (8‑692)

cpMvLXConst1[ 2 ] = cpMvLXCorner[ 2 ] (8‑693)

* The variables availableFlagConst1 and motionModelIdcConst1 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst1 is set equal to TRUE and motionModelIdcConst1 is set equal to 2.
      * Otherwise, availableFlagConst1 is set equal to FALSE and motionModelIdcConst1 is set equal to 0.

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 1 ] is equal to TRUE and availableFlagCorner[ 3 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 1 ] is equal to 1
* predFlagLXCorner[ 3 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 1 ]
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 3 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst2 = 1 (8‑694)

refIdxLXConst2 = refIdxLXCorner[ 0 ] (8‑695)

cpMvLXConst2[ 0 ] = cpMvLXCorner[ 0 ] (8‑696)

cpMvLXConst2[ 1 ] = cpMvLXCorner[ 1 ] (8‑697)

cpMvLXConst2[ 2 ] = cpMvLXCorner[ 3 ] + cpMvLXCorner[ 0 ] − cpMvLXCorner[ 1 ] (8‑698)

* The variables availableFlagConst2 and motionModelIdcConst2 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst2 is set equal to TRUE and motionModelIdcConst2 is set equal to 2.
      * Otherwise, availableFlagConst2 is set equal to FALSE and motionModelIdcConst2 is set equal to 0.

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE and availableFlagCorner[ 3 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* predFlagLXCorner[ 3 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 2 ]
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 3 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst3 = 1 (8‑699)

refIdxLXConst3 = refIdxLXCorner[ 0 ] (8‑700)

cpMvLXConst3[ 0 ] = cpMvLXCorner[ 0 ] (8‑701)

cpMvLXConst3[ 1 ] = cpMvLXCorner[ 3 ] + cpMvLXCorner[ 0 ] − cpMvLXCorner[ 2 ] (8‑702)

cpMvLXConst3[ 2 ] = cpMvLXCorner[ 2 ] (8‑703)

* The variables availableFlagConst3 and motionModelIdcConst3 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst3 is set equal to TRUE and motionModelIdcConst3 is set equal to 2.
      * Otherwise, availableFlagConst3 is set equal to FALSE and motionModelIdcConst3 is set equal to 0.

1. When availableFlagCorner[ 1 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE and availableFlagCorner[ 3 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 1 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* predFlagLXCorner[ 3 ] is equal to 1
* refIdxLXCorner[ 1 ] is equal to refIdxLXCorner[ 2 ]
* refIdxLXCorner[ 1 ] is equal to refIdxLXCorner[ 3 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst4 = 1 (8‑704)

refIdxLXConst4 = refIdxLXCorner[ 1 ] (8‑705)

cpMvLXConst4[ 0 ] = cpMvLXCorner[ 1 ] + cpMvLXCorner[ 2 ] − cpMvLXCorner[ 3 ] (8‑706)

cpMvLXConst4[ 1 ] = cpMvLXCorner[ 1 ] (8‑707)

cpMvLXConst4[ 2 ] = cpMvLXCorner[ 2 ] (8‑708)

* The variables availableFlagConst4 and motionModelIdcConst4 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst4 is set equal to TRUE and motionModelIdcConst4 is set equal to 2.
      * Otherwise, availableFlagConst4 is set equal to FALSE and motionModelIdcConst4 is set equal to 0.

The last two constructed affine control point motion vector merging candidates ConstK with K = 5..6 including the availability flags availableFlagConstK, the reference indices refIdxLXConstK, the prediction list utilization flags predFlagLXConstK, the affine motion model indices motionModelIdcConstK, and the constructed affine control point motion vectors cpMvLXConstK[ cpIdx ] with cpIdx = 0..2 and X being 0 or 1 are derived as follows:

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 1 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 1 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 1 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst5 = 1 (8‑709)

refIdxLXConst5 = refIdxLXCorner[ 0 ] (8‑710)

cpMvLXConst5[ 0 ] = cpMvLXCorner[ 0 ] (8‑711)

cpMvLXConst5[ 1 ] = cpMvLXCorner[ 1 ] (8‑712)

* The variables availableFlagConst5 and motionModelIdcConst5 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst5 is set equal to TRUE and motionModelIdcConst5 is set equal to 1.
      * Otherwise, availableFlagConst5 is set equal to FALSE and motionModelIdcConst5 is set equal to 0.

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 2 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following applies:
      * The second control point motion vector cpMvLXCorner[ 1 ] is derived as follows:

cpMvLXCorner[ 1 ][ 0 ] = ( cpMvLXCorner[ 0 ][ 0 ] << 7 ) +    
 ( ( cpMvLXCorner[ 2 ][ 1 ] − cpMvLXCorner[ 0 ][ 1 ] ) (8‑713)  
  << ( 7 + Log2( cbHeight / cbWidth ) ) )

cpMvLXCorner[ 1 ][ 1 ] = ( cpMvLXCorner[ 0 ][ 1 ] << 7 ) +    
 ( ( cpMvLXCorner[ 2 ][ 0 ] − cpMvLXCorner[ 0 ][ 0 ] ) (8‑714)  
  << ( 7 + Log2( cbHeight / cbWidth ) ) )

* + - * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLXCorner[ 1 ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLXCorner[ 1 ] as output.
      * The following assignments are made:

predFlagLXConst6 = 1 (8‑715)

refIdxLXConst6 = refIdxLXCorner[ 0 ] (8‑716)

cpMvLXConst6[ 0 ] = cpMvLXCorner[ 0 ] (8‑717)

cpMvLXConst6[ 1 ] = cpMvLXCorner[ 1 ] (8‑718)

* The variables availableFlagConst6 and motionModelIdcConst6 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst6 is set equal to TRUE and motionModelIdcConst6 is set equal to 1.
      * Otherwise, availableFlagConst6 is set equal to FALSE and motionModelIdcConst6 is set equal to 0.

#### 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.

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 variables availableFlagA and availableFlagB are both set equal to FALSE.
3. The sample locations ( xNbA0, yNbA0 ), ( xNbA1, yNbA1 ), ( xNbA2, yNbA2 ), ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ), and ( xNbB2, yNbB2 ) are derived as follows:

( xA0, yA0 ) = ( xCb − 1, yCb + cbHeight ) (8‑719)

( xA1, yA1 ) = ( xCb − 1, yCb + cbHeight − 1 ) (8‑720)

( xB0, yB0 ) = ( xCb + cbWidth , yCb − 1 ) (8‑721)

( xB1, yB1 ) = ( xCb + cbWidth − 1, yCb − 1 ) (8‑722)

( xB2, yB2 ) = ( xCb − 1, yCb − 1 ) (8‑723)

1. The following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* 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 ( xNbAk, yNbAk ) as inputs, and the output is assigned to the block availability flag availableAk.
* When availableAk is equal to TRUE and MotionModelIdc[ xNbAk ][ yNbAk ] is greater than 0 and availableFlagA is equal to FALSE, the following applies:
  + - The variable ( xNb, yNb ) is set equal to ( CbPosX[ xNbAk ][ yNbAk ], CbPosY[ xNbAk ][ yNbAk ] ), nbW is set equal to CbWidth[ xNbAk ][ yNbAk ],and nbH is set equal to CbHeight[ xNbAk ][ yNbAk ].
    - If PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1 and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagA is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 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 ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), 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.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLX[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ 0 ] (8‑724)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ 1 ] (8‑725)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ 2 ] (8‑726)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑727)

* + - Otherwise if PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagA is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 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 ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), 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.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLY[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLY[ 0 ] (8‑728)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLY[ 1 ] (8‑729)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLY[ 2 ] (8‑730)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑731)

1. 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 MotionModelIdc[ xNbBk ][ yNbBk ] is greater than 0 and availableFlagB is equal to FALSE, the following applies:
  + - The variable ( xNb, yNb ) is set equal to ( CbPosX[ xNbBk ][ yNbBk ], CbPosY[ xNbBk ][ yNbBk ] ), nbW is set equal to CbWidth[ xNbBk ][ yNbBk ],and nbH is set equal to CbHeight[ xNbBk ][ yNbBk ].
    - If PredFlagLX[ xNbBk ][ yNbBk ] is equal to 1 and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagB is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 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 ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), 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.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLX[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ 0 ] (8‑732)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ 1 ] (8‑733)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ 2 ] (8‑734)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑735)

* + - Otherwise if PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagB is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 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 ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), 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.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLY[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLY[ 0 ] (8‑736)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLY[ 1 ] (8‑737)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLY[ 2 ] (8‑738)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑739)

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 8.5.5.8 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 as inputs, and the availability flag availableConsFlagLX, the availability flags availableFlagLX[ cpIdx ] and cpMvpLX[ cpIdx ] with cpIdx = 0..numCpMv − 1 as outputs.
* When availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ 0 ] (8‑740)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ 1 ] (8‑741)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ 2 ] (8‑742)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑743)

1. The following applies for cpIdx = 0..numCpMv − 1:

* When numCpMvpCandLX is less than 2 and availableFlagLX[ cpIdx ] is equal to 1, the following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ cpIdx ] (8‑744)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ cpIdx ] (8‑745)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ cpIdx ] (8‑746)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑747)

1. When numCpMvpCandLX is less than 2, the following applies:

* The derivation process for temporal luma motion vector prediction as specified in clause 8.5.2.11 is with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX as inputs, and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
* When availableFlagLXCol is equal to 1, the following applies:
* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked 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.
* The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvLXCol (8‑748)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvLXCol (8‑749)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvLXCol (8‑750)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑751)

1. When numCpMvpCandLX is less than 2, the following is repeated until numCpMvpCandLX is equal to 2, with mvZero[0] and mvZero[1] both being equal to 0:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvZero (8‑752)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvZero (8‑753)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvZero (8‑754)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑755)

The affine control point motion vector predictor cpMvpLX with X being 0 or 1 is derived as follows:

cpMvpLX = cpMvpListLX[ mvp\_lX\_flag[ xCb ][ yCb ] ] (8‑756)

#### 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 availability flags availableFlagLX[ cpIdx ] with cpIdx = 0..2 and 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( RefPicList[ X ][ RefIdxLX[ xNbTL ][ yNbTL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 0 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 0 ] = MvLX[ xNbTL ][ yNbTL ] (8‑757)

* Otherwise, when PredFlagLY[ xNbTL ][ yNbTL ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbTL ][ yNbTL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 0 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 0 ] = MvLY[ xNbTL ][ yNbTL ] (8‑758)

* When availableFlagLX[ 0 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ 0 ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 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[ 1 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbTR ][ yNbTR ] is equal to 1, and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbTR ][ yNbTR ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 1 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 1 ] = MvLX[ xNbTR ][ yNbTR ] (8‑759)

* Otherwise, when PredFlagLY[ xNbTR ][ yNbTR ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbTR ][ yNbTR ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 1 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 1 ] = MvLY[ xNbTR ][ yNbTR ] (8‑760)

* When availableFlagLX[ 1 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ 1 ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 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( RefPicList[ X ][ RefIdxLX[ xNbBL ][ yNbBL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 2 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 2 ] = MvLX[ xNbBL ][ yNbBL ] (8‑761)

* Otherwise, when PredFlagLY[ xNbBL ][ yNbBL ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbBL ][ yNbBL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 2 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 2 ] = MvLY[ xNbBL ][ yNbBL ] (8‑762)

* When availableFlagLX[ 2 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ 2 ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvLX[ 2 ] as output.

The variable availableConsFlagLX is derived as follows:

* + 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 1, availableConsFlagLX is set equal to 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 cpMvLX[ cpIdx ], with cpIdx = 0..numCpMv − 1 and X being 0 or 1,
* the reference index refIdxLX and X being 0 or 1,
* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY.

Outputs of this process are:

* the luma subblock motion vector array mvLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1,
* the chroma subblock motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1.

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

CpMvLX[ x ][ y ][ 0 ] = cpMvLX[ 0 ] (8‑763)

CpMvLX[ x ][ y ][ 1 ] = cpMvLX[ 1 ] (8‑764)

CpMvLX[ x ][ y ][ 2 ] = cpMvLX[ 2 ] (8‑765)

The variables log2CbW and log2CbH are derived as follows:

log2CbW = Log2( cbWidth ) (8‑766)

log2CbH = Log2( cbHeight ) (8‑767)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

mvScaleHor = cpMvLX[ 0 ][ 0 ] << 7 (8‑768)

mvScaleVer = cpMvLX[ 0 ][ 1 ] << 7 (8‑769)

dHorX = ( cpMvLX[ 1 ][ 0 ] − cpMvLX[ 0 ][ 0 ] ) << ( 7 − log2CbW ) (8‑770)

dVerX = ( cpMvLX[ 1 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 − log2CbW ) (8‑771)

The variables dHorY and dVerY are derived as follows:

* If numCpMv is equal to 3, the following applies:

dHorY = ( cpMvLX[ 2 ][ 0 ] − cpMvLX[ 0 ][ 0 ] ) << ( 7 − log2CbH ) (8‑772)

dVerY = ( cpMvLX[ 2 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 − log2CbH ) (8‑773)

* Otherwise ( numCpMv is equal to 2), the following applies:

dHorY = − dVerX (8‑774)

dVerY = dHorX (8‑775)

For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:

* The luma motion vector mvLX[ xSbIdx ][ ySbIdx ] is derived as follows :

xPosCb = 2 + ( xSbIdx << 2 ) (8‑776)

yPosCb = 2 + ( ySbIdx << 2 ) (8‑777)

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = ( mvScaleHor + dHorX \* xPosCb + dHorY \* yPosCb ) (8‑778)

mvLX[ xSbIdx ][ ySbIdx ][ 1 ] = ( mvScaleVer + dVerX \* xPosCb + dVerY \* yPosCb ) (8‑779)

* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked 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 motion vectors mvLX[ xSbIdx ][ ySbIdx ] are clipped as follows:

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = Clip3( −217, 217 − 1, mvLX[ xSbIdx ][ ySbIdx ][ 0 ] ) (8‑780)

mvLX[ xSbIdx ][ ySbIdx ][ 1 ] = Clip3( −217, 217 − 1, mvLX[ xSbIdx ][ ySbIdx ][ 1 ] ) (8‑781)

For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:

* The average luma motion vector mvAvgLX is derived as follows:

mvAvgLX = mvLX[ ( xSbIdx >> 1 << 1) ][ (ySbIdx>>1<<1) ] +  (8‑782) mvLX[ ( xSbIdx >> 1 << 1 ) + 1 ][ (ySbIdx>>1<<1) + 1 ]

mvAvgLX[ 0 ] = mvAvgLX[ 0 ] >= 0 ? ( mvAvgLX[ 0 ] + 1 ) >> 1 : (8‑783) −( ( −mvAvgLX[ 0 ] + 1 ) >> 1 )

mvAvgLX[ 1 ] = mvAvgLX[ 1 ] >= 0 ? ( mvAvgLX[ 1 ] + 1 ) >> 1 : (8‑784) −( ( −mvAvgLX[ 1 ] + 1 ) >> 1 )

* The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvAvgLX and refIdxLX as inputs, and the chroma motion vector mvCLX[ xSbIdx ][ ySbIdx ] as output.

[Ed. (BB): This way four 2x2 chroma subblocks (4x4 chroma block) share the same motion vector which is derived from the average of two 4x4 luma subblock motion vectors. In the decoding process motion compensation is still performed on 2x2 chroma blocks which is however a motion compensation on a chroma 4x4 block because all chroma MVs inside a 4x4 chroma block are the same. I would prefer an editorial change that makes it more clear that affine chroma MC is performed on 4x4 chroma blocks.]

### Derivation process for intra prediction mode in combined merge and intra prediction

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 intra prediction mode 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 and ciip\_flag[ xNbX ][ yNbX ] is not equal to 1.
* X is equal to B and yCb − 1 is less than ( ( yCb  >>  CtbLog2SizeY )  <<  CtbLog2SizeY ).
* Otherwise, candIntraPredModeX is derived as follows:
* If IntraPredModeY[ xNbX ][ yNbX ] is greater than INTRA\_ANGULAR34, candIntraPredModeX is set equal to INTRA\_ANGULAR50.
* Otherwise, if IntraPredModeY[ xNbX ][ yNbX ] is less than or equal to INTRA\_ANGULAR34 and IntraPredModeY[ xNbX ][ yNbX ] is greater than INTRA\_DC, candIntraPredModeX is set equal to INTRA\_ANGULAR18.
* Otherwise, candIntraPredModeX is set equal to IntraPredModeY[ xNbX ][ yNbX ].

1. The candidate mode list 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 (8‑785)

candModeList[ 1 ] = INTRA\_DC (8‑786)

candModeList[ 2 ] = INTRA\_ANGULAR50 (8‑787)

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

candModeList[ 0 ] = candIntraPredModeA (8‑788)

candModeList[ 1 ] = INTRA\_PLANAR (8‑789)

candModeList[ 2 ] = INTRA\_DC (8‑790)

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

candModeList[ 0 ] = candIntraPredModeA (8‑791)

candModeList[ 1 ] = candIntraPredModeB (8‑792)

* 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. The following process is applied to update candModeList:

* If cbHeight is greater than 2 \* cbWidth and one of candModeList[ x ], with x = 0..2 is equal to INTRA\_ANGULAR50, the coressponding candModeList[ x ] is replaced with candIntraPredModeC, which is derived as follows:
* If none of candModeList[ x ], with x = 0..2 is equal to INTRA\_PLANAR, candIntraPredModeC is set equal to INTRA\_PLANAR.
* Otherwise, if none of candModeList[ x ], x = 0..2 is equal to INTRA\_DC, candIntraPredModeC is set equal to INTRA\_DC.
* Otherwise, if none of candModeList[ x ], with x = 0..2 is equal to INTRA\_ANGULAR50, candIntraPredModeC is set equal to INTRA\_ANGULAR50.
* Otherwise (none of candModeList[ x ], x = 0..2 is equal to INTRA\_ANGULAR18). candIntraPredModeC is set equal to INTRA\_ANGULAR18.
* Otherwise, if cbWidth is greater than 2 \* cbHeight and if one of candModeList[ x ], with x = 0..2 is equal to INTRA\_ANGULAR18, the coressponding candModeList[ x ] is replaced with candIntraPredModeC, which is derived as follows:
* If none of candModeList[ x ], with x = 0..2 is equal to INTRA\_PLANAR, candIntraPredModeC is set equal to INTRA\_PLANAR.
* Otherwise, if none of candModeList[ x ], x = 0..2 is equal to INTRA\_DC, candIntraPredModeC is set equal to INTRA\_DC.
* Otherwise, if none of candModeList[ x ], with x = 0..2 is equal to INTRA\_ANGULAR50, candIntraPredModeC is set equal to INTRA\_ANGULAR50.
* Otherwise (none of candModeList[ x ], x = 0..2 is equal to INTRA\_ANGULAR18). candIntraPredModeC is set equal to INTRA\_ANGULAR18.

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

* If ciip\_luma\_mpm\_flag[ xCb ][ yCb ] is equal to 1, IntraPredModeY[ xCb ][ yCb ] is set equal to candModeList[ intra\_luma\_mpm\_idx[ xCb ][ yCb ] ].
* Otherwise, IntraPredModeY[ xCb ][ yCb ] is set to equal to candIntraPredModeC, which is derived as follows:
* If none of candModeList[ x ], with x = 0..2 is equal to INTRA\_PLANAR, candIntraPredModeC is set equal to INTRA\_PLANAR.
* Otherwise, if none of candModeList[ x ], x = 0..2 is equal to INTRA\_DC, candIntraPredModeC is set equal to INTRA\_DC.
* Otherwise, if none of candModeList[ x ], with x = 0..2 is equal to INTRA\_ANGULAR50, candIntraPredModeC is set equal to INTRA\_ANGULAR50.
* Otherwise (none of candModeList[ x ], x = 0..2 is equal to INTRA\_ANGULAR18). candIntraPredModeC is set equal to INTRA\_ANGULAR18.

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 ].

### Decoding process for inter blocks

#### General

This process is invoked when decoding a coding unit 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,
* variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,
* the motion vectors mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* the refined motion vectors refMvL0[ xSbIdx ][ ySbIdx ] and refMvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and 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, and ySbIdx = 0 .. numSbY − 1,
* the bi-prediction weight index gbiIdx,
* a variable cIdx specifying the colour component index of the current block.

Outputs of this process are:

* an array predSamples of prediction samples.

Let predSamplesL0L, predSamplesL1L and predSamplesIntraL be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, predSamplesL0Cb, predSamplesL1Cb, predSamplesL0Cr and predSamplesL1Cr, predSamplesIntraCb, and predSamplesIntraCr be (cbWidth / 2)x(cbHeight / 2) arrays of predicted chroma sample values.

The width and the height of the current coding sublock subCbWidth, subCbHeight in luma samples are derived as follows:

sbWidth  =  cbWidth / numSbX (8‑793)

sbHeight  =  cbHeight / numSbY (8‑794)

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 ) (8‑795)

* The variable currPic specifies the current picture and the variable bdofFlag is derived as follows:
* If all of the following conditions are true, bdofFlag is set equal to TRUE.
* sps\_bdof\_enabled\_flag is equal to 1.
* predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] are both equal to 1.
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0 ] ) \* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ refIdxL1 ] ) is less than 0.
* MotionModelIdc[ xCb ][ yCb ] is equal to 0.
* merge\_subblock\_flag[ xCb ][ yCb ] is equal to 0.
* GbiIdx[ xCb ][ yCb ] is equal to 0.
* cIdx is equal to 0.
* Otherwise, bdofFlag is set equal to FALSE.
* For X being each of 0 and 1, when predFlagLX[ xSbIdx ][ ySbIdx ] 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 8.5.7.2 with X and refIdxLX as inputs.
  + The motion vector offset mvOffset is set equal to refMvLX[ xSbIdx ][ xSbIdx ] − mvLX[ xSbIdx ][ ySbIdx ].
  + When one or more of the following conditions is true, mvOffset[ 0 ] is set equal to 0:
    - xSb is not equal to xCb and mvOffset[ 0 ] is less than 0
    - ( xSb + sbWidth ) is not equal to ( xCb + cbWidth) and mvOffset[ 0 ] is greater than 0
  + When one or more of the following conditions is true, mvOffset[ 1 ] is set equal to 0:
    - ySb is not equal to yCb and mvOffset[ 1 ] is less than 0
    - ( ySb + sbHeight ) is not equal to ( yCb + cbHeight ) and mvOffset[ 1 ] is greater than 0
  + If cIdx is equal to 0, the following applies:
    - The array predSamplesLXL is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth, the coding subblock height sbHeight in luma samples, the luma motion vector offset mvOffset, the refined luma motion vector refMvLX[ xSb ][ xSb ], the reference array refPicLXL, bdofFlag, and cIdx as inputs.
  + Otherwise if cIdx is equal to 1, the following applies:
    - The array predSamplesLXCb is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth / 2, the coding subblock height sbHeight / 2, the chroma motion vector offset mvOffset, the refined chroma motion vector refMvLX[ xSb ][ xSb ], the reference array refPicLXCb, bdofFlag, and cIdx as inputs.
  + Otherwise (cIdx is equal to 2), the following applies:
    - The array predSamplesLXCr is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth / 2, the coding subblock height sbHeight / 2, the chroma motion vector offset mvOffset, the refined chroma motion vector refMvLX[ xSb ][ xSb ], the reference array refPicLXCr, bdofFlag, and cIdx as inputs.
* If bdofFlag is equal to TRUE, the following applies:
* The variable shift is set equal to Max( 2, 14 − BitDepthY ).
* The variables cuDiffThres, bdofBlkDiffThres, and cuSumDiff are derived as follows:

cuDiffThres = ( 1 << ( BitDepthY − 8 + shift ) ) \* cbWidth\*cbHeight (8‑796)

bdofBlkDiffThres = 1 << ( BitDepthY − 3 + shift ) (8‑797)

cuSumDiff = 0 (8‑798)

* For xIdx = 0..(sbWidth >> 2 ) − 1 and yIdx = 0..( sbHeight >> 2 ) − 1, the variables bdofBlkSumDiff and the bidirectional optical flow utilization flag bdofUtilizationFlag[ xIdx ][ yIdx ] are derived as follows:

bdofBlkSumDiff = ( predSamplesL0L[ ( xIdx << 2 ) + 1 + i ][ ( yIdx << 2 ) + 1 + j ] −  (8‑799)  
 predSamplesL1L[ ( xIdx << 2 ) + 1 + i ][ ( yIdx << 2 ) + 1 + j] )

bdofUtilizationFlag[ xIdx ][ yIdx ] =  bdofBlkSumDiff >= bdofBlkDiffThres (8‑800)

cuSumDiff += bdofBlkSumDiff (8‑801)

* When cuSumDiff is less than cuDiffThres, bdofFlag is set equal to FALSE.
* The array predSamples of prediction samples is derived as follows:
* If cIdx is equal to 0, the prediction samples inside the current luma coding subblock, predSamples[ xL + xSb ][ yL + ySb ] with xL = 0..sbWidth − 1 and yL = 0..sbHeight − 1, are derived as follows:
* If bdofFlag is equal to TRUE, the bidirectional optical flow sample prediction process as specified in clause 8.5.7.4 is invoked with nCbW set equal to the luma coding subblock width sbWidth, nCbH set equal to the luma coding subblock height sbHeight and the sample arrays predSamplesL0L and predSamplesL1L, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1 and bdofUtilizationFlag[ xIdx ][ yIdx ] with xIdx = 0..( sbWidth >> 2 ) − 1, yIdx = 0..( sbHeight >> 2 ) − 1as inputs, and predSamples[ xL + xSb ][ yL + ySb ] as outputs.
* Otherwise (bdofFlag is equal to FALSE), the weighted sample prediction process as specified in clause 8.5.7.5 is invoked with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays predSamplesL0L and predSamplesL1L, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, gbiIdx, and cIdx as inputs, and predSamples[ xL + xSb ][ yL + ySb ] as outputs.
* Otherwise, if cIdx is equal to 1, the prediction samples inside the current chroma component Cb coding block, predSamples[ 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 8.5.7.5 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[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, gbiIdx, and cIdx as inputs.
* Otherwise (cIdx is equal to 2), the prediction samples inside the current chroma component Cr coding block, predSamples[ 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 8.5.7.5 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[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, gbiIdx, and cIdx as inputs.
* When cIdx 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 ] (8‑802)

MvL1[ xSb + x ][ ySb + y ] = mvL1[ xSbIdx ][ ySbIdx ] (8‑803)

MvDmvrL0[ xSb + x ][ ySb + y ] = refMvL0[ xSbIdx ][ ySbIdx ] (8‑804)

MvDmvrL1[ xSb + x ][ ySb + y ] = refMvL1[ xSbIdx ][ ySbIdx ] (8‑805)

RefIdxL0[ xSb + x ][ ySb + y ] = refIdxL0 (8‑806)

RefIdxL1[ xSb + x ][ ySb + y ] = refIdxL1 (8‑807)

PredFlagL0[ xSb + x ][ ySb + y ] = predFlagL0[ xSbIdx ][ ySbIdx ] (8‑808)

PredFlagL1[ xSb + x ][ ySb + y ] = predFlagL1[ xSbIdx ][ ySbIdx ] (8‑809)

GbiIdx[ xSb + x ][ ySb + y ] = gbiIdx (8‑810)

When ciip\_flag[ xCb ][ yCb ] is equal to 1, the array predSamples of prediction samples is modifed as follows:

* If cIdx is equal to 0, the following applies:
* The general intra sample prediction process as specified in clause 8.4.4.2.1 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb, yCb ), the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], the transform block width nTbW and height nTbH set equal to cbWidth and cbHeight, the coding block width nCbW and height nCbH set equal to cbWidth and cbHeight, and the variable cIdx as inputs, and the output is assigned to the (cbWidth)x(cbHeight) array predSamplesIntraL.
* The weighted sample prediction process for combined merge and intra prediction as specified in clause 8.5.7.6 is invoked with the coding block width cbWidth, the coding block height cbHeight, the sample arrays predSamplesInter and predSamplesIntra set equal to predSamples and predSamplesIntraL, respectively, the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the colour component index cIdx as inputs, and the output is assigend to the (cbWidth)x(cbHeight) array predSamples.
* Otherwise, if cIdx is equal to 1, the following applies:
* The general intra sample prediction process as specified in clause 8.4.4.2.1 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb / 2, yCb / 2 ), the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], the transform block width nTbW and height nTbH set equal to cbWidth / 2 and cbHeight / 2, the coding block width nCbW and height nCbH set equal to cbWidth / 2 and cbHeight / 2, and the variable cIdx as inputs, and the output is assigned to the (cbWidth / 2)x(cbHeight / 2) array predSamplesIntraCb.
* The weighted sample prediction process for combined merge and intra prediction as specified in clause 8.5.7.6 is invoked with the coding block width cbWidth / 2, the coding block height cbHeight / 2, the sample arrays predSamplesInter and predSamplesIntra set equal to predSamplesCb and predSamplesIntraCb, respectively, the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the colour component index cIdx as inputs, and the output is assigend to the (cbWidth / 2)x(cbHeight / 2) array predSamples.
* Otherwise (cIdx is equal to 2), the following applies:
* The general intra sample prediction process as specified in clause 8.4.4.2.1 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb / 2, yCb / 2 ), the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], the transform block width nTbW and height nTbH set equal to cbWidth / 2 and cbHeight / 2, the coding block width nCbW and height nCbH set equal to cbWidth / 2 and cbHeight / 2, and the variable cIdx as inputs, and the output is assigned to the (cbWidth / 2)x(cbHeight / 2) array predSamplesIntraCr.
* The weighted sample prediction process for combined merge and intra prediction as specified in clause 8.5.7.6 is invoked with the coding block width cbWidth / 2, the coding block height cbHeight / 2, the sample arrays predSamplesInter and predSamplesIntra set equal to predSamplesCr and predSamplesIntraCr, respectively, the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the colour component index cIdx as inputs, and the output is assigend to the (cbWidth / 2)x(cbHeight / 2) array predSamples.

#### Reference picture selection process

Inputs to this process are:

* a value X representing a reference list being equal to either 0 or 1,
* 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 RefPicList[ X ][ refIdxLX ], where X is the value of X that this process is invoked for, 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 8.8 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,
* a variable sbHeight specifying the height of the current coding subblock,
* a motion vector offset mvOffset,
* a refined motion vector refMvLX,
* the selected reference picture sample array refPicLX,
* the bidirectional optical flow flag bdofFlag,
* a variable cIdx specifying the colour component index of the current block.

Outputs of this process are:

* an (sbWidth +bdofOffset)x(sbHeight +bdofOffset) array predSamplesLX of prediction sample values.

The bidirectional optical flow boundary offset bdofOffset is derived as follows:

bdofOffset = bdofFlag ? 2 : 0 (8‑811)

* If cIdx is equal to 0, the following applies:
  + 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 refPicLX.
  + For each luma sample location ( xL = 0..sbWidth − 1 +bdofOffset, yL = 0..sbHeight − 1 +bdofOffset ) inside the prediction luma sample array predSamplesLX, the corresponding prediction luma sample value predSamplesLX[ xL ][ yL ] is derived as follows:
* The variables xIntL, yIntL, xFracL and yFracL are derived as follows:

xIntL = xSb + ( refMvLX[ 0 ]  >>  4 ) + xL (8‑812)

yIntL = ySb + ( refMvLX[ 1 ]  >>  4 ) + yL (8‑813)

xFracL = refMvLX[ 0 ] & 15 (8‑814)

yFracL = refMvLX[ 1 ] & 15 (8‑815)

* + If bdofFlag is equal to TRUE and one or more of the following conditions are true, the prediction luma sample value predSamplesLX[ xL ][ yL ] is derived by invoking the luma integer sample fetching process as specified in clause 8.5.7.3.3 with ( xIntL, yIntL ), ( xFracL, yFracL ) and refPicLX as inputs:
    - bdofFlag is equal to TRUE.
    - xL is equal to 0.
    - xL is equal to sbWidth + 1.
    - yL is equal to 0.
    - yL is equal to sbHeight + .
  + Otherwise, the following applies:
    - The motion vector mvLX is set equal to ( refMvLX − mvOffset ).

The prediction luma sample value predSamplesLX[ xL ][ yL ] is derived by invoking the luma sample 8-tap interpolation filtering process as specified in clause 8.5.7.3.2 with ( xIntL, yIntL ), ( xFracL, yFracL ), refPicLX, and padVal as inputs.

* Otherwise (cIdx is not equal to 0), the following applies:
  + 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 refPicLX.
  + For each chroma sample location ( xC = 0..sbWidth − 1, yC = 0.. sbHeight − 1 ) inside the prediction chroma sample arrays predSamplesLX, the corresponding prediction chroma sample value predSamplesLX[ xC ][ yC ] is derived as follows:
* The variables xIntC, yIntC, xFracC and yFracC are derived as follows: [Ed. (SL): Shall we make it consistent: /2 or /SubWidthC and /SubHeightC?]

xIntC = ( xSb / SubWidthC ) + ( mvLX[ 0 ]  >>  5 ) + xC (8‑816)

yIntC = ( ySb / SubHeightC ) + ( mvLX[ 1 ]  >>  5 ) + yC (8‑817)

xFracC = mvLX[ 0 ] & 31 (8‑818)

yFracC = mvLX[ 1 ] & 31 (8‑819)

* + The motion vector mvLX is set equal to ( refMvLX − mvOffset ).
  + The list padVal[ dir ] is derived as follows for dir = 0..1:
    - The variable disp is derived as follows:

disp = ( refMvLX[ dir ]  >>  4) − ( mvLX[ dir ]  >>  4) + ( dir = = 0 ? xC : yC) (8‑820)

* + - If disp is less than 0, padVal[ dir ] is set equal to disp.
    - Otherwise, if disp is greater than ( dir  = = 0  ?  sbWidth / SubWidthC : sbHeight / SubWidthC ) − 1, padVal[ dir ] is set equal to disp − ( ( dir  = =  0  ?  sbWidth / SubWidthC : sbHeight / SubWidthC ) − 1).
    - Otherwise, padVal[ dir ] is set equal to 0.
  + The prediction sample value predSamplesLX[ xC ][ yC ] is derived by invoking the process specified in clause 8.5.7.3.4 with ( xIntC, yIntC ), ( xFracC, yFracC ), refPicLX, and padVal as inputs.

##### Luma sample 8-tap interpolation filtering 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,

– a list padVal[ dir ] with dir = 0,1 specifying reference sample padding direction and amount.

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 8‑9.

The luma interpolation filter coefficients fPadL[ 0 ] are set equal to fL[ xFracL ] and fPadL[ 1 ] are set equal to fL[ yFracL ] and modified depending on padVal[ dir ] as follows for dir being equal to 0 and 1:

– If padVal[ dir ] is equal to − 2, fPadL[ dir ] is modified as follows:

fPadL[ dir ][ 2 ] = fPadL[ xFracL ][ 2 ] + fPadL[ xFracL ][ 1 ] + fPadL[ xFracL ][ 0 ] (8‑821)

fPadL[ dir ][ 0 ] = 0 (8‑822)

fPadL[ dir ][ 1 ] = 0 (8‑823)

– Otherwise, if padVal[ dir ] is equal to − 1, fPadL[ dir ] is modified as follows:

fPadL[ dir ][ 1 ] = fPadL[ dir ][ 1 ] + fPadL[ dir ][ 0 ] (8‑824)

fPadL[ dir ][ 0 ] = 0 (8‑825)

– Otherwise, if padVal[ dir ] is equal to 1, fPadL[ dir ] is modified as follows:

fPadL[ dir ][ 6 ] = fPadL[ dir ][ 6 ] + fPadL[ dir ][ 7 ] (8‑826)

fPadL[ dir ][ 7 ] = 0 (8‑827)

– Otherwise, if padVal[ dir ] is equal to 2, fPadL[ dir ] is modified as follows:

fPadL[ dir ][ 5 ] = fPadL[ dir ][ 5 ] + fPadL[ dir ][ 6 ] + fPadL[ dir ][ 7 ] (8‑828)

fPadL[ dir ][ 6 ] = 0 (8‑829)

fPadL[ dir ][ 7 ] = 0 (8‑830)

The luma locations in full-sample units ( xInti, yInti ) are derived as follows for i = 0..7:

xInti = sps\_ref\_wraparound\_enabled\_flag ?  
 ClipH( ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY, picW, xIntL + i − 3 ) : (8‑831)   
 Clip3( 0, picW − 1, xIntL + i − 3 )

yInti = Clip3( 0, picH − 1, yIntL + i − 3 ) (8‑832)

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[ xInt3 ][ yInt3 ] << shift3 (8‑833)

– Otherwise, if xFracL is not equal to 0 and yFracL is equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL =   >>  shift1 (8‑834)

– Otherwise, if xFracL is equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL =   >>  shift1 (8‑835)

– 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:

temp[ n ] =   >>  shift1 (8‑836)

* The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL =   >>  shift2 (8‑837)

Table 8‑9 – 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 |

##### Luma integer sample fetching process

Inputs to this process are:

* a luma location in full-sample units ( xIntL, yIntL ),
* the luma reference sample array refPicLXL,

Output of this process is a predicted luma sample value predSampleLXL

The variable shift 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 locations in full-sample units ( xInt, yInt ) are derived as follows:

xInt = sps\_ref\_wraparound\_enabled\_flag ? (8‑838)  
 ClipH( ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY, picW, xIntL ) :  
 Clip3( 0, picW − 1, xIntL )

yInt = Clip3( 0, picH − 1, yIntL ) (8‑839)

The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL = refPicLXL[ xInt ][ yInt ] << shift3 (8‑840)

##### 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 chroma interpolation filter coefficients fC[ p ] for each 1/32 fractional sample position p equal to xFracC or  yFracC are specified in Table 8‑10.

The chroma interpolation filter coefficients fPadC[ 0 ] are set equal to fC[ xFracL ] and fPadC[ 1 ] are set equal to fC[ yFracL ] and modified depending on padVal[ dir ] as follows for dir being equal to 0 and 1:

– If padVal[ dir ] is equal to − 1, fPadC[ dir ] is modified as follows:

fPadC[ dir ][ 1 ] = fPadC[ dir ][ 1 ] + fPadC[ dir ][ 0 ] (8‑841)

fPadC[ dir ][ 0 ] = 0 (8‑842)

– Otherwise, if padVal[ dir ] is equal to 1, fPadC[ dir ] is modified as follows:

fPadC[ dir ][ 2 ] = fPadC[ dir ][ 2 ] + fPadC[ dir ][ 3 ] (8‑843)

fPadC[ dir ][ 3 ] = 0 (8‑844)

The variable xOffset is set equal to ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY ) / SubWidthC.

The chroma locations in full-sample units ( xInti, yInti ) are derived as follows for i = 0..3:

xInti = sps\_ref\_wraparound\_enabled\_flag ? ClipH( xOffset, picWC, xIntC + i − 1 ) : (8‑845)  
  Clip3( 0, picWC − 1, xIntC + i − 1 )

yInti = Clip3( 0, picHC − 1, yIntC + i − 1 ) (8‑846)

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[ xInt1 ][ yInt1 ] << shift3 (8‑847)

– Otherwise if xFracC is not equal to 0 and yFracC is equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC =   >>  shift1 (8‑848)

– Otherwise if xFracC is equal to 0 and yFracC is not equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC =   >>  shift1 (8‑849)

– 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:

temp[ n ] =   >>  shift1 (8‑850)

* The predicted chroma sample value predSampleLXC is derived as follows:

predSampleLXC =( fPadC[ 1][ 0 ] \* temp[ 0 ] +  
  fPadC[ 1 ][ 1 ] \* temp[ 1 ] +  
  fPadC[ 1 ][ 2 ] \* temp[ 2 ] + (8‑851)  
  fPadC[ 1 ][ 3 ] \* temp[ 3 ] ) >> shift2

Table 8‑10 – 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 |

#### Bidirectional optical flow 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 + 2)x(nCbH + 2) luma prediction sample arrays predSamplesL0 and predSamplesL1,
* the prediction list utilization flags predFlagL0 and predFlagL1,
* the reference indices refIdxL0 and refIdxL1,
* the bidirectional optical flow utilization flags bdofUtilizationFlag[ xIdx ][ yIdx ] with xIdx = 0..( nCbW >> 2 ) − 1, yIdx = 0..( nCbH >> 2 ) − 1.

Output of this process is the (nCbW)x(nCbH) array pbSamples of luma prediction sample values.

Variables bitDepth, shift1, shift2, shift3, shift4, offset4, and mvRefineThres are derived as follows:

* The variable bitDepth is set equal to BitDepthY.
* The variable shift1 is set to equal to Max( 2, 14 − bitDepth ).
* The variable shift2 is set to equal to Max( 8, bitDepth − 4 ).
* The variable shift3 is set to equal to Max( 5, bitDepth − 7 ).
* The variable shift4 is set equal to Max( 3, 15 − bitDepth ) and the variable offset4 is set equal to 1  <<  ( shift4 − 1 ).
* The variable mvRefineThres is set equal to Max( 2, 1  <<  ( 13 − bitDepth ) ).

For xIdx = 0..( nCbW >> 2 ) − 1 and yIdx = 0..( nCbH >> 2 ) − 1, the following applies:

* The variable xSb is set equal to ( xIdx << 2) + 1 and ySb is set equal to ( yIdx << 2 ) + 1.
* If bdofUtilizationFlag[ xSbIdx ][ yIdx ] is equal to FALSE, for x = xSb − 1..xSb + 2, y = ySb − 1.. ySb + 2, the prediction sample values of the current subblock are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 2bitDepth ) − 1, ( predSamplesL0[ x + 1 ][ y + 1 ] + offset2 + (8‑852)  
 predSamplesL1[ x + 1 ][ y + 1 ] )  >>  shift2 )

* Otherwise (bdofUtilizationFlag[ xSbIdx ][ yIdx ] is equal to TRUE), the prediction sample values of the current subblock are derived as follows:
  + For x =xSb − 1..xSb + 4, y = ySb − 1..ySb + 4, the following ordered steps apply:

1. The locations ( hx, vy ) for each of the corresponding sample locations ( x, y ) inside the prediction sample arrays are derived as follows:

hx = Clip3( 1, nCbW, x ) (8‑853)

vy = Clip3( 1, nCbH, y ) (8‑854)

1. The variables gradientHL0[ x ][ y ], gradientVL0[ x ][ y ], gradientHL1[ x ][ y ] and gradientVL1[ x ][ y ] are derived as follows:

gradientHL0[ x ][ y ]  =  (predSamplesL0[ hx + 1 ][vy] − predSampleL0[ hx − 1 ][ vy] ) >> shift1 (8‑855)

gradientVL0[ x ][ y ]  =   (predSampleL0[ hx ][ vy + 1 ] − predSampleL0[ hx][vy − 1 ] ) >> shift1 (8‑856)

gradientHL1[ x ][ y ]  =  (predSamplesL1[ hx + 1 ][vy] − predSampleL1[ hx − 1 ][ vy] ) >> shift1 (8‑857)

gradientVL1[ x ][ y ]  =   (predSampleL1[ hx ][ vy + 1 ] − predSampleL1[ hx][vy − 1 ] ) >> shift1 (8‑858)

1. The variables temp[ x ][ y ], tempH[ x ][ y ] and tempV[ x ][ y ] are derived as follows:

diff[ x ][ y ] = (predSamplesL0[ hx ][ vy ] >> shift2 ) − ( predSamplesL1[ hx ][ vy ] >> shift2 ) (8‑859)

tempH[ x ][ y ] = (gradientHL0[ x ][ y ] + gradientHL1[ x ][ y ] ) >> shift3 (8‑860)

tempV[ x ][ y ] = (gradientVL0[ x ][ y ] + gradientVL1[ x ][ y ] ) >> shift3 (8‑861)

* + The variables sGx2, sGy2, sGxGy, sGxdI and sGydI are derived as follows:

sGx2 = ΣiΣj( tempH[ xSb + i ][ ySb + j ] \* tempH[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑862)

sGy2 = ΣiΣj(tempV[ xSb + i ][ ySb + j ] \* tempV[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑863)

sGxGy = ΣiΣj(tempH[ xSb + i ][ ySb + j ] \* tempV[ xSb + i ][ ySb + j ] ) with i, j  −1..4 (8‑864)

sGxdI = ΣiΣj( − tempH[ xSb + i ][ ySb + j ] \* diff[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑865)

sGydI = ΣiΣj( − tempV[ xSb + i ][ ySb + j ] \* diff[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑866)

* + The horizontal and vertical motion offset of the current subblock are derived as:

vx = sGx2 > 0  ?  Clip3( −mvRefineThres, mvRefineThres, (8‑867)  
 −( sGxdI << 3 ) >> Floor( Log2( sGx2 ) ) )  :  0

vy = sGy2 > 0  ?  Clip3( −mvRefineThres, mvRefineThres, ( ( sGydI << 3 ) −  (8‑868)  
 ( ( vx \* sGxGym ) << 12 + vx \* sGxGys ) >> 1 ) >> Floor( Log2( sGx2 ) ) )  :  0

* + For x =xSb − 1..xSb + 2, y = ySb − 1..ySb + 2, the prediction sample values of the current sub-block are derived as follows:

bdofOffset = Round( ( vx \* ( gradientHL1[ x + 1 ][ y + 1 ] − gradientHL0[ x + 1 ][ y + 1 ] ) ) >> 1 ) (8‑869)  
 + Round( ( vy \* (gradientVL1[ x + 1 ][ y + 1 ] − gradientVL0[ x + 1 ][ y + 1 ] ) ) >> 1 )

[Ed. (JC): Round() operation is defined for float input. The Round() operation seems redundant here since the input is an integer value. To be confirmed by the proponent]

pbSamples[ x ][ y ] = Clip3( 0, ( 2bitDepth ) − 1, ( predSamplesL0[ x + 1 ][ y + 1 ]  + offset4 + (8‑870)  
 predSamplesL1[ x + 1 ][ y + 1 ] + bdofOffset )  >>  shift4 )

#### Weighted sample prediction process

##### General

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,
* the bi-prediction weight index gbiIdx,
* the variable cIdx specifying the 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.

The variable weightedPredFlag is derived as follows:

* If tile\_group\_type is equal to P, weightedPredFlag is set equal to weighted\_pred\_flag.
* Otherwise (tile\_group\_type is equal to B), weightedPredFlag is set equal to weighted\_bipred\_flag.

The following applies:

* If weightedPredFlag is equal to 0, the array pbSamples of the prediction samples is derived by invoking the default weighted sample prediction process as specified in clause 8.5.7.5.1 with the coding block width nCbW, the coding block height nCbH, two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1, the prediction list utilization flags predFlagL0 and predFlagL1, the bi-prediction weight index gbiIdx and the bit depth bitDepth as inputs.
* Otherwise (weightedPredFlag is equal to 1), the array pbSamples of the prediction samples is derived by invoking the weighted sample prediction process as specified in clause 8.5.7.5.3 with the coding block width nCbW, the coding block height nCbH, two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1, the prediction list utilization flags predFlagL0 and predFlagL1, the reference indices refIdxL0 and refIdxL1, the colour component index cIdx and the bit depth bitDepth as inputs.

##### Default 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,
* the bi-prediction weight index gbiIdx.
* the sample bit depth, bitDepth.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

Variables shift1, shift2, offset1, offset2, and offset3 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 ).
* The variable offset3 is set equal to 1  <<  ( shift2 + 2 ).

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 ) (8‑871)

* 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 ) (8‑872)

* Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the following applies:
* If gbiIdx is equal to 0, the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, (8‑873)  
 ( predSamplesL0[ x ][ y ] + predSamplesL1[ x ][ y ] + offset2 )  >>  shift2 )

* Otherwise (gbiIdx is not equal to 0), the following applies:
* The variable w1 is set equal to gbiWLut[ gbiIdx ] with gbiWLut[ k ] = { 4, 5, 3, 10, −2 }.
* The variable w0 is set equal to ( 8 − w1 ).
* The prediction sample values are derived as follows.

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, (8‑874)  
 ( w0\*predSamplesL0[ x ][ y ] + w1\*predSamplesL1[ x ][ y ] + offset3 )  >>  (shift2+3) )

##### Explicit 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,
* the variable cIdx specifying the colour component index,
* the sample bit depth, bitDepth.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable shift1 is set equal to Max( 2, 14 − bitDepth ).

The variables log2Wd, o0, o1, w0 and w1 are derived as follows:

* If cIdx is equal to 0 for luma samples, the following applies:

log2Wd = luma\_log2\_weight\_denom + shift1 (8‑875)

w0 = LumaWeightL0[ refIdxL0 ] (8‑876)

w1 = LumaWeightL1[ refIdxL1 ] (8‑877)

o0 = luma\_offset\_l0[ refIdxL0 ] << WpOffsetBdShiftY (8‑878)

o1 = luma\_offset\_l1[ refIdxL1 ] << WpOffsetBdShiftY (8‑879)

* Otherwise (cIdx is not equal to 0 for chroma samples), the following applies:

log2Wd = ChromaLog2WeightDenom + shift1 (8‑880)

w0 = ChromaWeightL0[ refIdxL0 ][ cIdx − 1 ] (8‑881)

w1 = ChromaWeightL1[ refIdxL1 ][ cIdx − 1 ] (8‑882)

o0 = ChromaOffsetL0[ refIdxL0 ][ cIdx − 1 ] << WpOffsetBdShiftC (8‑883)

o1 = ChromaOffsetL1[ refIdxL1 ][ cIdx − 1 ] << WpOffsetBdShiftC (8‑884)

The prediction sample 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:

if( log2Wd >= 1 )  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1,  
 ( ( predSamplesL0[ x ][ y ] \* w0 + 2log2Wd − 1 ) >> log2Wd ) + o0 ) (8‑885)  
else  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1, predSamplesL0[ x ][ y ] \* w0 + o0 )

* Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

if( log2Wd >= 1 )  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1,  
 ( ( predSamplesL1[ x ][ y ] \* w1 + 2log2Wd − 1 ) >> log2Wd ) + o1 ) (8‑886)  
else  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1, predSamplesL1[ x ][ y ] \* w1 + o1 )

* 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 ] \* w0 + predSamplesL1[ x ][ y ] \* w1 +  
 ( ( o0 + o1 + 1 )  <<  log2Wd ) )  >>  ( log2Wd + 1 ) ) (8‑887)

#### Weighted sample prediction process for combined merge and intra prediction

Inputs to this process are:

* the width of the current coding block cbWidth,
* the height of the current coding block cbHeight,
* two (cbWidth)x(cbHeight) arrays predSamplesInter and predSamplesIntra,
* the intra prediction mode predModeIntra,
* a variable cIdx specifying the colour component index.

Output of this process is the (cbWidth)x(cbHeight) array predSamplesComb 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.

The prediction samples predSamplesComb[ x ][ y ] with x = 0..cbWidth − 1 and y = 0..cbHeight − 1 are derived as follows:

* The weight w is derived as follows:
* If one or more of the following conditions are true, w is set equal to 4:
* cbWidth is less than 4.
* cbHeight is less than 4.
* predModeIntra is equal to INTRA\_PLANAR
* predModeIntra is equal to INTRA\_DC.
* Otherwise, if predModeIntra is INTRA\_ANGULAR50, w is specified in Table 8‑11 with nPos equal to y and nSize equal to cbHeight.
* Otherwise, if predModeIntra is INTRA\_ANGULAR18, w is specified in Table 8‑11 with nPos equal to x and nSize equal to cbWidth.
* Otherwise, w is set equal to 4.
* When cIdx is equal to 0 and tile\_group\_lmcs\_enabled\_flag is equal to 1, predSamplesInter is modified as follows:

idxY = predSamplesInter[ x ][ y ] >> Log2( OrgCW )   
predSamplesInter [ x ][ y ] = Clip1Y ( LmcsPivot[ idxY ] +  (8‑888)  
 ( ScaleCoeff[ idxY ] \* ( predSamplesInter[ x ][ y ] − InputPivot[ idxY ] ) +   
 ( 1 << 13 ) ) >> 14 )

* The prediction samples predSamplesComb[ x ][ y ] are derived as follows:

predSamplesComb[ x ][ y ] = ( w \* predSamplesIntra[ x ][ y ] +  (8‑889)  
 ( 8 − w ) \* predSamplesInter[ x ][ y ] ) >> 3 )

Table 8‑11 – Specification of w as a function of the position nP and the size nS

|  |  |  |  |
| --- | --- | --- | --- |
| **0 <= nP < ( nS / 4 )** | **( nS / 4 ) <= nP < ( nS / 2 )** | **( nS / 2 ) <= nP < ( 3 \*nS / 4 )** | **( 3 \*nS / 4 ) <= nP < nS** |
| 6 | 5 | 3 | 2 |

### Decoding process for triangle inter blocks

#### General

This process is invoked when decoding a coding unit with merge\_triangle\_flag[ xCb ][ yCb ] equal to 1.

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,
* the luma motion vectors in 1/16 fractional-sample accuracy mvLA and mvLB,
* the chroma motion vectors mvCLA and mvCLB,
* the reference indices refIdxLA and refIdxLB,
* the prediction list flags predFlagLA and predFlagLB.

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 predSamplesLAL and predSamplesLBL be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, predSamplesLACb, predSamplesLBCb, predSamplesLACr and predSamplesLBCr be (cbWidth / 2)x(cbHeight / 2) arrays of predicted chroma sample values.

The predSamplesL, predSamplesCb and predSamplesCr are derived by the following ordered steps:

1. For N being each of A and B, the following applies:

* The reference picture consisting of an ordered two-dimensional array refPicLNL of luma samples and two ordered two-dimensional arrays refPicLNCb and refPicLNCr of chroma samples is derived by invoking the process specified in clause 8.5.7.2 with X set equal to N and refIdxLX set equal to refIdxLN as input.
* The array predSamplesLNL is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the luma coding block width sbWidth set equal to cbWidth, the luma coding block height sbHeight set equal to cbHeight, the motion vector mvLX set equal to mvLN and the reference array refPicLXL set equal to refPicLNL, and the variable cIdx is set equal to 0 as inputs.
* The array predSamplesLNCb is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding block width sbWidth set equal to cbWidth / 2, the coding block height sbHeight set equal to cbHeight / 2, the motion vector mvLX set equal to mvCLN, and the reference array refPicLXCb set equal to refPicLNCb, and the variable cIdx is set equal to 1 as inputs.
* The array predSamplesLNCr is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding block width sbWidth set equal to cbWidth / 2, the coding block height sbHeight set equal to cbHeight / 2, the motion vector mvLX set equal to mvCLN, and the reference array refPicLXCr set equal to refPicLNCr, and the variable cIdx is set equal to 2 as inputs.

1. The partition direction of merge triangle mode variable triangleDir is set equal to merge\_triangle\_split\_dir[ xCb ][ yCb ].
2. The prediction samples inside the current luma coding block, predSamplesL[ xL ][ yL ] with xL = 0..cbWidth − 1 and yL = 0..cbHeight − 1, are derived by invoking the weighted sample prediction process for triangle merge mode specified in clause 8.5.8.2 with the coding block width nCbW set equal to cbWidth, the coding block height nCbH set equal to cbHeight, the sample arrays predSamplesLAL and predSamplesLBL, and the variables triangleDir, and cIdx equal to 0 as inputs.
3. The prediction samples inside the current chroma component Cb coding block, predSamplesCb[ xC ][ yC ] with xC = 0..cbWidth / 2 − 1 and yC = 0..cbHeight / 2 − 1, are derived by invoking the weighted sample prediction process for triangle merge mode specified in clause 8.5.8.2 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays predSamplesLACb and predSamplesLBCb, and the variables triangleDir, and cIdx equal to 1 as inputs.
4. The prediction samples inside the current chroma component Cr coding block, predSamplesCr[ xC ][ yC ] with xC = 0..cbWidth / 2 − 1 and yC = 0..cbHeight / 2 − 1, are derived by invoking the weighted sample prediction process for triangle merge mode specified in clause 8.5.8.2 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays predSamplesLACr and predSamplesLBCr, and the variables triangleDir, and cIdx equal to 2 as inputs.
5. The motion vector storing process for merge triangle mode specified in clause 8.5.8.3 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the partition direction triangleDir, the luma motion vectors mvLA and mvLB, the reference indices refIdxLA and refIdxLB, and the prediction list flags predFlagLA and predFlagLB as inputs.

#### Weighted sample prediction process for triangle merge mode

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 predSamplesLA and predSamplesLB,
* a variable triangleDir specifying the partition direction,
* a variable cIdx specifying colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable nCbR is derived as follows:

nCbR = ( nCbW > nCbH ) ? ( nCbW / nCbH ) : ( nCbH / nCbW ) (8‑890)

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 and offset1 are derived as follows:

– The variable shift1 is set equal to Max( 5, 17 − bitDepth).

– The variable offset1 is set equal to 1  <<  ( shift1 − 1 ).

Depending on the values of triangleDir, wS and cIdx, the prediction samples pbSamples[ x ][ y ] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

– The variable wIdx is derived as follows:

* If cIdx is equal to 0 and triangleDir is equal to 0, the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 8, ( x / nCbR − y ) + 4 ) ) (8‑891)  
 : ( Clip3( 0, 8, ( x / nCbR − y ) + 4 ) )

* Otherwise, if cIdx is equal to 0 and triangleDir is equal to 1, the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 8, ( nCbH − 1 − x / nCbR − y ) + 4 ) ) (8‑892)  
 ( Clip3( 0, 8, ( nCbW − 1 − x / nCbR − y ) + 4 ) )

* Otherwise, if cIdx is greater than 0 and triangleDir is equal to 0, the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 4, ( x / nCbR − y ) + 2 ) ) (8‑893)  
 : ( Clip3( 0, 4, ( x / nCbR − y ) + 2 ) )

* Otherwise (if cIdx is greater than 0 and triangleDir is equal to 1), the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 4, ( nCbH − 1 − x / nCbR − y ) + 2 ) ) (8‑894)  
 ( Clip3( 0, 4, ( nCbW − 1 − x / nCbR − y ) + 2 ) )

– The variable wValue specifying the weight of the prediction sample is derived using wIdx and cIdx as follows:

wValue = ( cIdx = = 0 ) ? Clip3( 0, 8, wIdx ) : Clip3( 0, 8, wIdx \* 2 ) (8‑895)

– The prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, ( predSamplesLA[ x ][ y ] \* wValue +  (8‑896)  
 predSamplesLB[ x ][ y ] \* ( 8 − wValue ) + offset1 ) >> shift1 )

#### Motion vector storing process for triangle merge mode

This process is invoked when decoding a coding unit with merge\_triangle\_flag[ xCb ][ yCb ] equal to 1.

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 triangleDir specifying the partition direction,
* the luma motion vectors in 1/16 fractional-sample accuracy mvLA and mvLB,
* the reference indices refIdxLA and refIdxLB,
* the prediction list flags predFlagLA and predFlagLB.

The variables numSbX and numSbY specifying the number of 4x4 blocks in the current coding block in horizontal and vertical direction are set equal to numSbX = cbWidth >> 2 and numSbY = cbHeight >> 2.

The variable minSb is set equal to min( numSbX, numSbY ).

The variable cbRatio is derived as follows:

cbRatio = ( cbWidth > cbHeight ) ? ( cbWidth / cbHeight ) : ( cbHeight / cbWidth ) (8‑897)

The variable refIdxTempLA is derived by invoking the reference picture mapping process for triangle merge mode specified in clause 8.5.8.4 with X set equal to A and refIdxLN set equal to refIdxLA as inputs.

The variable refIdxTempLB is derived by invoking the reference picture mapping process for triangle merge mode specified in clause 8.5.8.4 with the X set equal to B and refIdxLN set equal to refIdxLB as inputs.

For each 4x4 subblock at subblock index ( xSbIdx, ySbIdx ) with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the following applies:

* The variables xIdx and yIdx are derived as follows:

xIdx = ( cbWidth > cbHeight ) ? ( xSbIdx / cbRatio ) : xSbIdx (8‑898)

yIdx = ( cbWidth > cbHeight ) ? ySbIdx : ( ySbIdx / cbRatio ) (8‑899)

* The variable sType is derived as follows:
* If triangleDir is equal to 0, the following applies:

sType = ( xIdx = = yIdx ) ? 2 : ( ( xIdx > yIdx ) ? 0 : 1 ) (8‑900)

* Otherwise (triangleDir is equal to 1), the following applies:

sType = ( xIdx + yIdx = = minSb ) ? 2 : ( ( xIdx + yIdx < minSb ) ? 0 : 1 ) (8‑901)

* Depending on the value of sType, the following assignments are made:
* If sType is equal to 0, the following applies:

predFlagL0 = ( predFlagLA = = 0 ) ? 1 : 0 (8‑902)

predFlagL1 = ( predFlagLA = = 0 ) ? 0 : 1 (8‑903)

refIdxL0 = ( predFlagLA = = 0 ) ? refIdxLA : −1 (8‑904)

refIdxL1 = ( predFlagLA = = 0 ) ? −1 : refIdxLA (8‑905)

mvL0[ 0 ] = ( predFlagLA = = 0 ) ? mvLA[ 0 ] : 0 (8‑906)

mvL0[ 1 ] = ( predFlagLA = = 0 ) ? mvLA[ 1 ] : 0 (8‑907)

mvL1[ 0 ] = ( predFlagLA = = 0 ) ? 0 : mvLA[ 0 ] (8‑908)

mvL1[ 1 ] = ( predFlagLA = = 0 ) ? 0 : mvLA[ 1 ] (8‑909)

* Otherwise, if sType is equal to 1, the following applies:

predFlagL0 = ( predFlagLB = = 0 ) ? 1 : 0 (8‑910)

predFlagL1 = ( predFlagLB = = 0 ) ? 0 : 1 (8‑911)

refIdxL0 = ( predFlagLB = = 0 ) ? refIdxLB : −1 (8‑912)

refIdxL1 = ( predFlagLB = = 0 ) ? −1 : refIdxLB (8‑913)

mvL0[ 0 ] = ( predFlagLB = = 0 ) ? mvLB[ 0 ] : 0 (8‑914)

mvL0[ 1 ] = ( predFlagLB = = 0 ) ? mvLB[ 1 ] : 0 (8‑915)

mvL1[ 0 ] = ( predFlagLB = = 0 ) ? 0 : mvLB[ 0 ] (8‑916)

mvL1[ 1 ] = ( predFlagLB = = 0 ) ? 0 : mvLB[ 1 ] (8‑917)

* Otherwise (sType is equal to 2), the following applies:
  + If predFlagLA + predFlagLB is equal to 1,

predFlagL0 = 1 (8‑918)

predFlagL1 = 1 (8‑919)

refIdxL0 = ( predFlagLA = = 0 ) ? refIdxLA : refIdxLB (8‑920)

refIdxL1 = ( predFlagLA = = 0 ) ? refIdxLB : refIdxLA (8‑921)

mvL0[ 0 ] = ( predFlagLA = = 0 ) ? mvLA[ 0 ] : mvLB[ 0 ] (8‑922)

mvL0[ 1 ] = ( predFlagLA = = 0 ) ? mvLA[ 1 ] : mvLB[ 1 ] (8‑923)

mvL1[ 0 ] = ( predFlagLA = = 0 ) ? mvLB[ 0 ] : mvLA[ 0 ] (8‑924)

mvL1[ 1 ] = ( predFlagLA = = 0 ) ? mvLB[ 1 ] : mvLA[ 1 ] (8‑925)

* + If predFlagLA + predFlagLB is equal to 0, the following applies:

predFlagL0 = 1 (8‑926)

predFlagL1 = ( refIdxTempLA = = −1 && refIdxTempLB = = −1 ) ? 0 : 1 (8‑927)

refIdxL0 = ( refIdxTempLB != −1 ) ? refIdxLA : (8‑928)  
 ( ( refIdxTempLA != −1 ) ? refIdxLB : refIdxLA )

refIdxL1 = ( refIdxTempLB != −1 ) ? refIdxTempLB : (8‑929)  
 ( ( refIdxTempLA != −1 ) ? refIdxTempLA : −1 )

mvL0[ 0 ] = ( refIdxTempLB != −1 ) ? mvLA[ 0 ] : (8‑930)  
 ( ( refIdxTempLA != −1 ) ? mvLB[ 0 ] : mvLA[ 0 ] )

mvL0[ 1 ] = ( refIdxTempLB != −1 ) ? mvLA[ 1 ] : (8‑931)  
 ( ( refIdxTempLA != −1 ) ? mvLB[ 1 ] : mvLA[ 1 ] )

mvL1[ 0 ] = ( refIdxTempLB != −1 ) ? mvLB[ 0 ] : (8‑932)  
 ( ( refIdxTempLA != −1 ) ? mvLA[ 0 ] : 0 )

mvL1[ 1 ] = ( refIdxTempLB != −1 ) ? mvLB[ 1 ] : (8‑933)  
 ( ( refIdxTempLA != −1 ) ? mvLA[ 1 ] : 0 )

* + If predFlagLA + predFlagLB is equal to 2, the following applies:

predFlagL0 = ( refIdxTempLA = = −1 && refIdxTempLB = = −1 ) ? 0 : 1 (8‑934)

predFlagL1 = 1 (8‑935)

refIdxL0 = ( refIdxTempLB != −1 ) ? refIdxTempLB : (8‑936)  
 ( ( refIdxTempLA != −1 ) ? refIdxTempLA : −1 )

refIdxL1 = ( refIdxTempLB != −1 ) ? refIdxLA : (8‑937)  
 ( ( refIdxTempLA != −1 ) ? refIdxLB : refIdxLA )

mvL0[ 0 ] = ( refIdxTempLB != −1 ) ? mvLB[ 0 ] : (8‑938)  
 ( ( refIdxTempLA != −1 ) ? mvLA[ 0 ] : 0 )

mvL0[ 1 ] = ( refIdxTempLB != −1 ) ? mvLB[ 1 ] : (8‑939)  
 ( ( refIdxTempLA != −1 ) ? mvLA[ 1 ] : 0 )

mvL1[ 0 ] = ( refIdxTempLB != −1 ) ? mvLA[ 0 ] : (8‑940)  
 ( ( refIdxTempLA != −1 ) ? mvLB[ 0 ] : mvLA[ 0 ] )

mvL1[ 1 ] = ( refIdxTempLB != −1 ) ? mvLA[ 1 ] : (8‑941)  
 ( ( refIdxTempLA != −1 ) ? mvLB[ 1 ] : mvLA[ 1 ] )

* The following assignments are made for x = 0..3 and y = 0..3:

MvL0[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = mvL0 (8‑942)

MvL1[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = mvL1 (8‑943)

RefIdxL0[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = refIdxL0 (8‑944)

RedIdxL1[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = refIdxL1 (8‑945)

PredFlagL0[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = predFlagL0 (8‑946)

PredFlagL1[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = predFlagL1 (8‑947)

#### Reference picture mapping process for triangle merge mode

Input to this process are:

* a variable X representing a reference list being equal to 0 or 1,
* a reference index refIdxLN.

Output of this process is:

* a reference index refIdxTemp.

The variable refPicPoc is derived as follows:

refPicPoc = ( X = = 0 ) ? RefPicList[ 0 ][ refIdxLN ] : RefPicList[ 1 ][ refIdxLN ] (8‑948)

The reference picture list refPicListTemp is derived as follows:

refPicListTemp = ( X = = 0 ) ? RefPicList[ 1 ] : RefPicList[ 0 ] (8‑949)

The variable refIdxTemp is derived as follows:

* The variable mapStop is set equal to FALSE.
* For the variable refIdxm with m = 0..NumRefIdxActive[ 1 ] − 1, the following applies until mapStop is equal to FALSE:

refIdxTemp = ( refPicListTemp[ refIdxm ] = = refPicPoc ) ? refIdxm : −1 (8‑950)

mapStop = ( refIdxTemp != −1 ) ? TRUE : FALSE (8‑951)

### 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 (8‑952)

The luma sample location is derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTb0, yTb0 ) : ( xTb0 / 2, yTb0 / 2 ) (8‑953)

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 newTbW and newTbH are derived as follows:

newTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑954)

newTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑955)

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 set equal to newTbW and the height nTbH set equal to newTbH, 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 + newTbW, yTb0 ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, 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 + newTbH ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, 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 + newTbW, yTb0 + newTbH ), the transform block width nTbW set equal to newTbW and height nTbH set equal to newTbH, 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 8.7.2 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.

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

### General decoding process for coding units coded in IBC 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.

The derivation process for quantization parameters as specified in clause 8.7.1 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, and the variable treeType as inputs.

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

1. The motion vector components of the current coding unit are derived as follows:
   * + If treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the following applies:

* The derivation process for motion vector components as specified in clause 8.6.2.1 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 luma motion vector mvL[ 0 ][ 0 ] as output.
* When treeType is equal to SINGLE\_TREE, the derivation process for chroma motion vectors in clause 8.6.2.9 is invoked with luma motion vector mvL[ 0 ][ 0 ] as input, and chroma motion vector mvC[ 0 ][ 0 ] as output.
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are both set equal to 1.
  + - Otherwise, if treeType is equal to DUAL\_TREE\_CHROMA, the following applies:
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

numSbX = ( cbWidth >> 2 ) (8‑956)

numSbY = ( cbHeight >> 2 ) (8‑957)

* The chroma motion vectors mvC[ xSbIdx ][ ySbIdx ] are derived as follows for xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1:
* The luma motion vector mvL[ xSbIdx ][ ySbIdx ] is derived as follows:
* The location ( xCuY, yCuY ) of the collocated luma coding unit is dervied as follows:

xCuY = xCb + xSbIdx\*4 (8‑958)

yCuY = yCb + ySbIdx\*4 (8‑959)

* If CuPredMode[ xCuY ][ yCuY ] is equal to MODE\_INTRA, the following applies.

mvL[ xSbIdx ][ ySbIdx ][ 0 ] = 0 (8‑960)

mvL[ xSbIdx ][ ySbIdx ][ 1 ] = 0 (8‑961)

predFlagL0[ xSbIdx ][ ySbIdx ] = 0 (8‑962)

predFlagL1[ xSbIdx ][ ySbIdx ] = 0 (8‑963)

* Otherwise ( CuPredMode[ xCuY ][ yCuY ] is equal to MODE\_IBC ), the following applies:

mvL[ xSbIdx ][ ySbIdx ][ 0 ]=MvL0[ xCuY ][ yCuY ][ 0 ] (8‑964)

mvL[ xSbIdx ][ ySbIdx ][ 1 ]=MvL0[ xCuY ][ yCuY ][ 1 ] (8‑965)

predFlagL0[ xSbIdx ][ ySbIdx ] = 1 (8‑966)

predFlagL1[ xSbIdx ][ ySbIdx ] = 0 (8‑967)

* The derivation process for chroma motion vectors in clause 8.6.2.9 is invoked with mvL[ xSbIdx ][ ySbIdx ] as inputs, and mvC[ xSbIdx ][ ySbIdx ] as output.
* It is a requirement of bitstream conformance that the chroma motion vector mvC[ xSbIdx ][ ySbIdx ] shall obey the following constraints:
* When the derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ) and the neighbouring chroma location ( xCb / SubWidthC + ( mvC[ xSbIdx ][ ySbIdx ][ 0 ] >> 5 ), yCb / SubHeightC + ( mvC[ xSbIdx ][ ySbIdx ][ 1 ] >> 5 ) ) as inputs, and the output shall be equal to TRUE.
* When the derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ) and the neighbouring chroma location ( xCb / SubWidthC + ( mvC[ xSbIdx ][ ySbIdx ][ 0 ] >> 5 ) + cbWidth / SubWidthC − 1, yCb / SubHeightC + ( mvC[ xSbIdx ][ ySbIdx ][ 1 ] >> 5 ) + cbHeight / SubHeightC − 1 ) as inputs, and the output shall be equal to TRUE.
* One or both of the following conditions shall be true:
* ( mvC[ xSbIdx ][ ySbIdx ][ 0 ] >> 5 ) + xSbIdx \* 2 + 2 is less than or equal to 0.
* ( mvC[ xSbIdx ][ ySbIdx ][ 1 ] >> 5 ) + ySbIdx \* 2 + 2 is less than or equal to 0.

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

* If treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the prediction samples of the current coding unit are derived as follows:
  + - * The decoding process for ibc blocks as specified in clause 8.6.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the variable cIdx set equal to 0 as inputs, and the ibc prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array predSamplesL of prediction luma samples as outputs.
* Otherwise if treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the prediction samples of the current coding unit are derived as follows:
  + - * The decoding process ibc blocks as specified in clause 8.6.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvC[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 and the variable cIdx set equal to 1 as inputs, and the ibc prediction samples (predSamples) that are an (cbWidth / 2)x(cbHeight / 2) array predSamplesCb of prediction chroma samples for the chroma components Cb as outputs.
      * The decoding process for ibc blocks as specified in clause 8.6.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvC[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 and the variable cIdx set equal to 2 as inputs, and the ibc prediction samples (predSamples) that are an (cbWidth / 2)x(cbHeight / 2) array predSamplesCr of prediction chroma samples for the chroma components Cr as outputs.

1. The variables NumSbX[ xCb ][ yCb ] and NumSbY[ xCb ][ yCb ] are set equal to numSbX and numSbY, respectively.
2. The residual samples of the current coding unit are derived as follows:

* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_LUMA, the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.9 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 cIdxset equal to 0 as inputs, and the array resSamplesL as output.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.9 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 cIdxset equal to 1 as inputs, and the array resSamplesCb as output.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.9 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 cIdxset equal to 2 as inputs, and the array resSamplesCr as output.

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

* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_LUMA, the picture reconstruction process for a colour component as specified in clause 8.7.5 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 set equal to 0, 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.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the picture reconstruction process for a colour component as specified in clause 8.7.5 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 set equal to 1, 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.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the picture reconstruction process for a colour component as specified in clause 8.7.5 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 set equal to 2, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to predSamplesCr and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to resSamplesCr as inputs, and the output is a modified reconstructed picture before in-loop filtering.

### Derivation process for motion vector components for IBC blocks

#### 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 luma motion vector in 1/16 fractional-sample accuracy mvL.

The luma motion vector mvL is derived as follows:

* If merge\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for luma motion vectors for merge mode as specified in clause 8.6.2.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight inputs, and the output being the luma motion vector mvL.
* Otherwise, the following applies:

1. The variable mvd is derived as follows:

mvd[ 0 ] = MvdL0[ xCb ][ yCb ][ 0 ] (8‑968)

mvd[ 1 ] = MvdL0[ xCb ][ yCb ][ 1 ] (8‑210)

1. The derivation process for luma motion vector prediction in clause 8.6.2.6 is invoked with the luma coding block location ( xCb, yCb ), the coding block width cbWidth and the coding block height cbHeight as inputs, and the output being mvp.
2. The luma motion vector mvL is derived as follows:

u[ 0 ] = ( mvp[ 0 ] + mvd[ 0 ] + 218 ) % 218 (8‑969)

mvL[ 0 ] = ( u[ 0 ] >= 217 ) ? ( u[ 0 ] − 218 ) : u[ 0 ] (8‑970)

u[ 1 ] = ( mvp[ 1 ] + mvd[ 1 ] + 218 ) % 218  (8‑971)

mvL[ 1 ] = ( u[ 1 ] >= 217 ) ? ( u[ 1 ] − 218 ) : u[ 1 ] (8‑972)

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

The updating process for the history-based motion vector predictor list as specified in clause 8.6.2.10 is invoked with luma motion vector mvL.

It is a requirement of bitstream conformance that the luma motion vector mvL shall obey the following constraints:

* When the derivation process for block availability 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 ( xCb + ( mvL[ 0 ] >> 4 ), yCb + ( mvL[ 1 ] >> 4 ) ) as inputs, and the output shall be equal to TRUE.
* When the derivation process for block availability 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 ( xCb + ( mvL[ 0 ] >> 4 ) + cbWidth − 1, yCb + ( mvL[ 1 ] >> 4 ) + cbHeight − 1 ) as inputs, and the output shall be equal to TRUE.
* One or both the following conditions shall be true:
  + The value of ( mvL[ 0 ] >> 4 ) + cbWidth is less than or equal to 0.
  + The value of ( mvL[ 1 ] >> 4 ) + cbHeight is less than or equal to 0.
* The following conditions shall be true:

( yCb + ( mvL[ 1 ] >> 4 ) ) >> CtbLog2SizeY = yCb >> CtbLog2SizeY (8‑973)

( yCb + ( mvL[ 1 ] >> 4 ) + cbHeight − 1) >> CtbLog2SizeY = yCb >> CtbLog2SizeY (8‑974)

( xCb + ( mvL[ 0 ] >> 4 ) ) >> CtbLog2SizeY >= ( xCb >> CtbLog2SizeY ) − 1 (8‑975)

( xCb + ( mvL[ 0 ] >> 4 ) + cbWidth − 1) >> CtbLog2SizeY <= ( xCb >> CtbLog2SizeY ) (8‑976)

[Ed. (SL): conditions (8-218) and (8-216) might have been checked by 6.4.X.]

* When ( xCb + ( mvL[ 0 ] >> 4 ) ) >> CtbLog2SizeY is equal to ( xCb >> CtbLog2SizeY ) − 1, the derivation process for block availability 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 ( ( ( xCb + ( mvL[ 0 ] >> 4 ) + CtbSizeY ) >> ( CtbLog2SizeY − 1 ) ) << ( CtbLog2SizeY − 1 ), ( ( yCb + ( mvL[ 1 ] >> 4 ) ) >> ( CtbLog2SizeY − 1 ) ) << ( CtbLog2SizeY − 1 ) ) as inputs, and the output shall be equal to FALSE.

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

This process is only invoked when merge\_flag[ xCb ][ yPb ] is equal to 1 and CuPredMode[ xCb ][ yPb ] is equal to MODE\_IBC, 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 luma motion vectors in 1/16 fractional-sample accuracy mvL.

The luma motion vector mvL is derived by the following ordered steps:

1. The derivation process for merging candidates from neighbouring coding units as specified in clause 8.6.2.3 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 outputs being the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1 and availableFlagB2 and the motion vectors mvA0, mvA1, mvB0, mvB1 and mvB2.
2. The merging motion vector candidate list, mergeMvCandList, is constructed as follows:

i = 0  
if( availableFlagA1 )  
 mergeMvCandList [ i++ ] = mvA1  
if( availableFlagB1 )  
 mergeMvCandList [ i++ ] = mvB1if( availableFlagB0 ) (8‑977)  
 mergeMvCandList [ i++ ] = mvB0if( availableFlagA0 )  
 mergeMvCandList [ i++ ] = mvA0if( availableFlagB2 )  
 mergeMvCandList [ i++ ] = mvB2

1. The variable numCurrMergeCand is set equal to the number of merging candidates in the mergeMvCandList.
2. When numCurrMergeCand is less than (MaxNumMergeCand − 1) and NumHmvpIbcCand is greater than 0, the derivation process of history-based merging candidates as specified in 8.6.2.5 is invoked with mergeMvCandList, and numCurrMergeCand as inputs, and modified mergeMvCandList and numCurrMergeCand as outputs.
3. When numCurrMergeCand is less than MaxNumMergeCand and greater than 1, the derivation process for pairwise average merging candidate specified in clause 8.6.2.4 is invoked with mergeMvCandList and numCurrMergeCand as inputs, and the outputs are assigned to mergeMvCandList and numCurrMergeCand.

[Ed. (SL): Alert that the merging motion vector candidate list may not be full. In rare case the merge index may point to an empty entry of the merging motion vector candidate array.]

1. The following assignments are made:

mvL[ 0 ] = mergeMvCandList[ merge\_idx[ xCb ][ yCb ] ][ 0 ] (8‑978)

mvL[ 1 ] = mergeMvCandList[ merge\_idx[ xCb ][ yCb ] ][ 1 ] (8‑979)

#### 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:

* the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1 and availableFlagB2 of the neighbouring coding units,
* the motion vectors in 1/16 fractional-sample accuracy mvA0, mvA1, mvB0, mvB1 and mvB2 of the neighbouring coding units,

For the derivation of availableFlagA1 and mvA1 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 and mvLA1 are derived as follows:
* If availableA1 is equal to FALSE, availableFlagA1 is set equal to 0 and both components of mvA1 are set equal to 0.
* Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvA1 = MvL0[ xNbA1 ][ yNbA1 ] (8‑980)

For the derivation of availableFlagB1 and mvB1 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 and mvB1 are derived as follows:
* If one or more of the following conditions are true, availableFlagB1 is set equal to 0 and both components of mvB1 are set equal to 0:
  + availableB1 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB1, yNbB1 ) have the same motion vectors.
* Otherwise, availableFlagB1 is set equal to 1 and the following assignments are made:

mvB1 = MvL0[ xNbB1 ][ yNbB1 ] (8‑981)

For the derivation of availableFlagB0 and mvB0 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 and mvB0 are derived as follows:
  + If one or more of the following conditions are true, availableFlagB0 is set equal to 0 and both components of mvB0 are set equal to 0:
  + availableB0 is equal to FALSE.
  + availableB1 is equal to TRUE and the luma locations ( xNbB1, yNbB1 ) and ( xNbB0, yNbB0 ) have the same motion vectors.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB0, yNbB0 ) have the same motion vectors.
* Otherwise, availableFlagB0 is set equal to 1 and the following assignments are made:

mvB0 = MvL0[ xNbB0 ][ yNbB0 ] (8‑982)

For the derivation of availableFlagA0 and mvA0 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 and mvA0 are derived as follows:
* If one or more of the following conditions are true, availableFlagA0 is set equal to 0 and both components of mvA0 are set equal to 0:
  + availableA0 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbA0, yNbA0 ) have the same motion vectors.
  + availableB1 is equal to TRUE, the luma locations ( xNbB1, yNbB1 ) and ( xNbA0, yNbA0 ) have the same motion vectors.
  + availableB0 is equal to TRUE, the luma locations ( xNbB0, yNbB0 ) and ( xNbA0, yNbA0 ) have the same motion vectors.
* Otherwise, availableFlagA0 is set equal to 1 and the following assignments are made:

mvA0 = MvL0[ xNbA0 ][ yNbA0 ] (8‑983)

For the derivation of availableFlagB2 and mvB2 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 and mvB2 are derived as follows:
* If one or more of the following conditions are true, availableFlagB2 is set equal to 0 and both components of mvB2 are set equal to 0:
  + availableB2 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB2, yNbB2 ) have the same motion vectors.
  + availableB1 is equal to TRUE and the luma locations ( xNbB1, yNbB1 ) and ( xNbB2, yNbB2 ) have the same motion vectors.
  + availableB0 is equal to TRUE and the luma locations ( xNbB0, yNbB0 ) and ( xNbB2, yNbB2 ) have the same motion vectors.
  + availableA0 is equal to TRUE and the luma locations ( xNbA0, yNbA0 ) and ( xNbB2, yNbB2 ) have the same motion vectors.
* Otherwise, availableFlagB2 is set equal to 1 and the following assignments are made:

mvB2 = MvL0[ xNbB2 ][ yNbB2 ] (8‑984)

#### Derivation process for pairwise average merging candidate

Inputs to this process are:

* a merging motion vector candidate list mergeMvCandList,
* the number of elements numCurrMergeCand within mergeMvCandList.

Outputs of this process are:

* the merging motion vector candidate list mergeMvCandList,
* the number of elements numCurrMergeCand within mergeCandList.

The candidate mvAvgCand is derived as follows and numCurrMergeCand is incremented by 1:

mvAvgCand[ 0 ] = ( ( mergeMvCandList[ 0 ][ 0 ] + mergeMvCandList[ 1 ][ 0 ] ) / 32 ) << 4 (8‑985)

mvAvgCand[ 1 ] = ( ( mergeMvCandList[ 0 ][ 1 ] + mergeMvCandList[ 1 ][ 1 ] ) / 32 ) << 4 (8‑986)

The candidate mvAvgCand is added at the end of mergeMvCandList and numCurrMergeCand is incremented by 1 as follows:

mergeMvCandList[ numCurrMergeCand++ ] = mvAvgCand (8‑987)

#### Derivation process for history-based merging candidates

Inputs to this process are:

* a merge candidate list mergeCandList,
* the number of available merging candidates in the list numCurrMergeCand.

Outputs to this process are:

* the modified merging candidate list mergeCandList,
* the modified number of merging candidates in the list numCurrMergeCand.

The variables isPrunedA1 and isPrunedB1 are set both equal to FALSE.

For each candidate in HmvpIbcCandList[ hMvpIdx ] with index hMvpIdx = 1..NumHmvpIbcCand, the following ordered steps are repeated until numCurrMergeCand is equal to ( MaxNumMergeCand − 1):

1. The variable sameMotion is derived as follows:
   * If all of the following conditions are true for any merging candidate N with N being A1 or B1, sameMotion and isPrunedN are both set equal to TRUE:
   * hMvpIdx is less than or equal to 2.
   * The candidate HmvpIbcCandList[NumHmvpIbcCand − hMvpIdx] is equal to the merging candidate N.
   * isPrunedN is equal to FALSE.
   * Otherwise, sameMotion is set equal to FALSE.
2. When sameMotion is equal to FALSE, the candidate HmvpIbcCandList[NumHmvpIbcCand − hMvpIdx] is added to the merging candidate list as follows:

mergeCandList[ numCurrMergeCand++ ] = HmvpIbcCandList[ NumHmvpIbcCand − hMvpIdx ] (8‑988)

#### 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,

Output of this process is the prediction mvp in 1/16 fractional-sample accuracy of the motion vector mv.

The motion vector predictor mvp is derived in the following ordered steps:

1. The derivation process for motion vector predictor candidate list as specified in clause 8.6.2.7 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 motion vector predictor candidate list mvpList as output.
2. The motion vector predictor mvp is derived as follows:

mvp = mvpList[ mvp\_l0\_flag[ xCb ][ yCb ] ] (8‑989)

#### 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,

Output of this process is motion vector predictor candidate list mvpList in 1/16 fractional-sample accuracy.

The motion vector predictor candidate list mvpList 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 8.6.2.8 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 availability flags availableFlagN and the motion vectors mvN, with N being replaced by A or B, as outputs.
2. The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvN, 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 mvN, with N being replaced by A or B, as outputs.
3. The motion vector predictor candidate list, mvpList, is constructed as follows:

numCurrMvpCand = 0  
if( availableFlagA ) {  
 mvpList[ numCurrMvpCand++ ] = mvA  
 if( availableFlagB && ( mvA != mvB ) )  
 mvpList[ numCurrMvpCand++ ] = mvB (8‑990)  
} else if( availableFlagB )  
 mvpList[ numCurrMvpCand++ ] = mvB

1. When numCurrMvpCand is less than 2 and NumHmvpIbcCand is greater than 0, the following applies for i= 1..Min( 4, NumHmvpIbcCand ) until numCurrMvpCand is equal to 2:

* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to the motion vector of the candidate HmvpIbcCandList[ i − 1 ], rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded motion vector of the candidate HmvpIbcCandList[ i − 1 ] as output is assigned to mvpList[ numCurrMvpCand++ ].

1. When numCurrMvpCand is less than 2, the following applies for until numCurrMvpCand is equal to 2:

* mvpList[ numCurrMvpCand ][ 0 ] is set equal to 0.
* mvpList[ numCurrMvpCand ][ 1 ] is set equal to 0.
* numCurrMvpCand is increased by 1.

#### 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,

Outputs of this process are (with N being replaced by A or B):

* the motion vectors mvN in 1/16 fractional-sample accuracy of the neighbouring coding units,
* the availability flags availableFlagN of the neighbouring coding units.

Figure 8‑2 provides an overview of spatial motion vector neighbours.

The variable currCb specifies the current luma coding block at luma location ( xCb, yCb ) and the variable currPic specifies the current picture.

The motion vector mvA and the availability flag availableFlagA 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 availableFlagA is set equal to 0 and both components of mvA 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. The following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* When availableAk is equal to TRUE and availableFlagA is equal to 0, the following applies:
* If CuPredMode[ xNbAk ][ yNbAk ] is equal to MODE\_IBC, availableFlagA is set equal to 1 and the following applies:

mvA = MvL0[ xNbAk ][ yNbAk ] (8‑991)

The motion vector mvB and the availability flag availableFlagB 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 availableFlagB is set equal to 0 and the both components of mvB 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 availableFlagB is equal to 0, the following applies:
* If CuPredMode[ xNbBk ][ yNbBk ] is equal to MODE\_IBC, availableFlagB is set equal to 1 and the following assignments are made:

mvB = MvL0[ xNbBk ][ yNbBk ] (8‑992)

#### Derivation process for chroma motion vectors

Input to this process is:

* a luma motion vector in 1/16 fractional-sample accuracy mvL.

Output of this process is a chroma motion vector in 1/32 fractional-sample accuracy mvC.

A chroma motion vector is derived from the corresponding luma motion vector.

The chroma motion vector mvC is derived as follows:

mvC[ 0 ] = ( ( mvL[ 0 ] >> ( 3 + SubWidthC ) ) \* 32 (8‑993)

mvC[ 1 ] = ( ( mvL[ 1 ] >> ( 3 + SubHeightC ) ) \* 32 (8‑994)

#### Updating process for the history-based motion vector predictor candidate list

Inputs to this process are:

* luma motion vector mvL in 1/16 fractional-sample accuracy.

The candidate list HmvpIbcCandList is modified by the following ordered steps:

1. The variable identicalCandExist is set equal to FALSE and the variable removeIdx is set equal to 0.
2. When NumHmvpIbcCand is greater than 0, for each index hMvpIdx with hMvpIdx = 0..NumHmvpIbcCand − 1, the following steps apply until identicalCandExist is equal to TRUE:
   * When hMvpCand is equal to HmvpIbcCandList[ hMvpIdx ], identicalCandExist is set equal to TRUE and removeIdx is set equal to hMvpIdx.
3. The candidate list HmvpIbcCandList is updated as follows:
   * If identicalCandExist is equal to TRUE or NumHmvpIbcCand  is equal to MaxNumMergeCand − 1, the following applies:
   * For each index i with i = ( removeIdx + 1 )..( NumHmvpIbcCand  − 1 ), HmvpIbcCandList[ i − 1] is set equal to HmvpIbcCandList [ i ].
   * HmvpIbcCandList[ NumHmvpIbcCand − 1 ] is set equal to mvCand.
   * Otherwise (identicalCandExist is equal to FALSE and NumHmvpIbcCand  is less than MaxNumMergeCand − 1), the following applies:
   * HmvpIbcCandList[ NumHmvpIbcCand ++ ] is set equal to mvCand.

### Decoding process for ibc blocks

#### General

This process is invoked when decoding a coding unit coded in ibc 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,
* variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,
* the motion vectors mv[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* a variable cIdx specifying the colour component index of the current block.

Outputs of this process are:

* an array predSamples of prediction samples.

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 reference indices refIdxL0 and refIdxL1 are set equal to −1.

The prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] are set equal to 1 and predFlagL1[ xSbIdx ][ ySbIdx ] are set equal to 0 with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,

The bi-prediction weight index gbiIdx is set equal to 0,

The width and the height of the current coding sublock sbWidth and sbHeight in luma samples are derived as follows:

sbWidth  =  cbWidth / numSbX (8‑995)

sbHeight  =  cbHeight / numSbY (8‑996)

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 ) (8‑997)

* The variable bdofFlag is set equal to FALSE.

The current decoded picture consists of a pic\_width\_in\_luma\_samples by pic\_height\_in\_luma\_samples array of luma samples currPicL and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples currPicCb and currPicCr. The current decoded picture sample arrays currPicL, currPicCb and currPicCr correspond to decoded sample arrays SL, SCb and SCr derived in clause 8.8 for the current decoded picture.

* If cIdx is equal to 0, the following applies:
  + The array predSamplesL0L is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth, the coding subblock height sbHeight in luma samples, the luma motion vector mv[ xSb ][ ySb ], the reference array currPicL, bdofFlag, and cIdx as inputs.
* Otherwise if cIdx is equal to 1, the following applies:
  + The array predSamplesL0Cb is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth / 2, the coding subblock height sbHeight / 2, the chroma motion vector mv[ xSb ][ ySb ], the reference array currPicCb, bdofFlag, and cIdx as inputs.
* Otherwise (cIdx is equal to 2), the following applies:
  + The array predSamplesL0Cr is derived by invoking the fractional sample interpolation process specified in clause 8.5.7.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth / 2, the coding subblock height sbHeight / 2, the chroma motion vector mv[ xSb ][ ySb ], the reference array currPicCr, bdofFlag, and cIdx as inputs.
* The array predSamples of prediction samples is derived as follows:
* If cIdx is equal to 0, the prediction samples inside the current luma coding subblock, predSamples[ xL + xSb ][ yL + ySb ] with xL = 0..sbWidth − 1 and yL = 0..sbHeight − 1, are derived by invoking the weighted sample prediction process as specified in clause 8.5.7.5 is invoked with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays predSamplesL0 L and predSamplesL1L, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, gbiIdx, and cIdx as inputs, and predSamples[ xL + xSb ][ yL + ySb ] as outputs.
* Otherwise, if cIdx is equal to 1, the prediction samples inside the current chroma component Cb coding block, predSamples[ 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 8.5.7.5 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[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, gbiIdx, and cIdx as inputs.
* Otherwise (cIdx is equal to 2), the prediction samples inside the current chroma component Cr coding block, predSamples[ 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 8.5.7.5 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[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, gbiIdx, and cIdx as inputs.
* When cIdx is equal to 0, the following assignments are made for x = 0..sbWidth − 1 and y = 0..sbHeight − 1:

MvL0[ xSb + x ][ ySb + y ] = mv[ xSbIdx ][ ySbIdx ] (8‑998)

MvL1[ xSb + x ][ ySb + y ] = 0 (8‑999)

RefIdxL0[ xSb + x ][ ySb + y ] = refIdxL0 (8‑1000)

RefIdxL1[ xSb + x ][ ySb + y ] = refIdxL1 (8‑1001)

PredFlagL0[ xSb + x ][ ySb + y ] = predFlagL0[ xSbIdx ][ ySbIdx ] (8‑1002)

PredFlagL1[ xSb + x ][ ySb + y ] = predFlagL1[ xSbIdx ][ ySbIdx ] (8‑1003)

GbiIdx[ xSb + x ][ ySb + y ] = gbiIdx (8‑1004)

## Scaling, transformation and array construction process

### Derivation process for quantization parameters

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left luma 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 tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed.

In this process, the luma quantization parameter Qp′Y and the chroma quantization parameters Qp′Cb and Qp′Cr are derived.

The luma location ( xQg, yQg ), specifies the top-left luma sample of the current quantization group relative to the top left luma sample of the current picture. The horizontal and vertical positions xQg and yQg are set equal to CuQgTopLeftX and CuQgTopLeftY, respectively.

NOTE – : The current quantization group is a rectangluar region inside a coding tree block that shares the same qPY\_PRED. Its width and height are equal to the width and height of the coding tree node of which the top-left luma sample position is assigned to the variables CuQgTopLeftX and CuQgTopLeftY.

When treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the predicted luma quantization parameter qPY\_PRED is derived by the following ordered steps:

1. The variable qPY\_PREV is derived as follows:

– If one or more of the following conditions are true, qPY\_PREV is set equal to TileGroupQpY:

– The current quantization group is the first quantization group in a tile group.

– The current quantization group is the first quantization group in a tile.

– Otherwise, qPY\_PREV is set equal to the luma quantization parameter QpY of the last luma coding unit in the previous quantization group in decoding order.

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 location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xQg − 1, yQg ) as inputs, and the output is assigned to availableA. The variable qPY\_A is derived as follows:

– If one or more of the following conditions are true, qPY\_A is set equal to qPY\_PREV:

– availableA is equal to FALSE.

– the CTB address ctbAddrA of the CTB containing the luma coding block covering the luma location ( xQg − 1, yQg ) is not equal to CtbAddrInTs, where ctbAddrA is derived as follows:

xTmp = ( xQg − 1 )  >>  MinTbLog2SizeY  
yTmp = yQg  >>  MinTbLog2SizeY  
minTbAddrA = MinTbAddrZs[ xTmp ][ yTmp ]  
ctbAddrA = minTbAddrA  >>  ( 2  \* ( CtbLog2SizeY − MinTbLog2SizeY ) ) (8‑1005)

– Otherwise, qPY\_A is set equal to the luma quantization parameter QpY of the coding unit containing the luma coding block covering ( xQg − 1, yQg ).

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 location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xQg, yQg − 1 ) as inputs, and the output is assigned to availableB. The variable qPY\_B is derived as follows:

– If one or more of the following conditions are true, qPY\_B is set equal to qPY\_PREV:

– availableB is equal to FALSE.

– the CTB address ctbAddrB of the CTB containing the luma coding block covering the luma location ( xQg,  yQg − 1 ) is not equal to CtbAddrInTs, where ctbAddrB is derived as follows:

xTmp = xQg  >>  MinTbLog2SizeY  
yTmp = ( yQg − 1 )  >>  MinTbLog2SizeY  
minTbAddrB = MinTbAddrZs[ xTmp ][ yTmp ]  
ctbAddrB = minTbAddrB  >>  ( 2 \* ( CtbLog2SizeY − MinTbLog2SizeY ) ) (8‑1006)

– Otherwise, qPY\_B is set equal to the luma quantization parameter QpY of the coding unit containing the luma coding block covering ( xQg, yQg − 1 ).

1. The predicted luma quantization parameter qPY\_PRED is derived as follows:

* If all the following conditions are true, then qPY\_PRED is set equal to the luma quantization parameter QpY of the coding unit containing the luma coding block covering ( xQg, yQg − 1 ):
* availableB is equal to TRUE.
* the current quantization group is the first quantization group in a CTB row within a tile
* Otherwise, qPY\_PRED is derived as follows:

qPY\_PRED = ( qPY\_A + qPY\_B + 1 ) >> 1 (8‑1007)

The variable QpY is derived as follows:

QpY = ( ( qPY\_PRED + CuQpDeltaVal + 64 + 2 \* QpBdOffsetY )%( 64 + QpBdOffsetY ) ) − QpBdOffsetY (8‑1008)

The luma quantization parameter Qp′Y is derived as follows:

Qp′Y = QpY + QpBdOffsetY (8‑1009)

[Ed. (BB): Modify highlighted sections when tiles without tile groups are incorporated]

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the following applies:

– When treeType is equal to DUAL\_TREE\_CHROMA, the variable QpY is set equal to the luma quantization parameter QpY of the luma coding unit that covers the luma location ( xCb + cbWidth / 2, yCb + cbHeight / 2 ).

– The variables qPCb and qPCr are derived as follows:

qPiCb = Clip3( −QpBdOffsetC, 69, QpY + pps\_cb\_qp\_offset + tile\_group\_cb\_qp\_offset ) (8‑1010)

qPiCr = Clip3( −QpBdOffsetC, 69, QpY + pps\_cr\_qp\_offset + tile\_group\_cr\_qp\_offset ) (8‑1011)

– If ChromaArrayType is equal to 1, the variables qPCb and qPCr are set equal to the value of QpC as specified in Table 8‑12 based on the index qPiequal to qPiCb and qPiCr, respectively.

– Otherwise, the variables qPCb and qPCr are set equal to Min( qPi, 63 ), based on the index qPiequal to qPiCb and qPiCr, respectively.

– The chroma quantization parameters for the Cb and Cr components, Qp′Cb and Qp′Cr, are derived as follows:

Qp′Cb = qPCb + QpBdOffsetC (8‑1012)

Qp′Cr = qPCr + QpBdOffsetC (8‑1013)

Table 8‑12 – Specification of QpC as a function of qPi for ChromaArrayType equal to 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| qPi | < 30 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | > 43 |
| QpC | = qPi | 29 | 30 | 31 | 32 | 33 | 33 | 34 | 34 | 35 | 35 | 36 | 36 | 37 | 37 | = qPi − 6 |

### 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 (8‑1014)

bdShift = Max( 20 − bitDepth, 0 ) (8‑1015)

tsShift = 5 + ( ( Log2( nTbW ) + Log2( nTbH ) ) / 2 ) (8‑1016)

The (nTbW)x(nTbH) array of residual samples resSamples is derived as follows:

* 1. The scaling process for transform coefficients as specified in clause 8.7.3 is invoked with the transform block location ( xTbY, yTbY ), the transform block width nTbW and the transform block 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 ] is equal to 1 and cIdx is equal to 0, the residual sample array values r[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

r[ x ][ y ] = d[ x ][ y ]  <<  tsShift (8‑1017)

* Otherwise (transform\_skip\_flag[ xTbY ][ yTbY ] is equal to 0 or and cIdx is not equal to 0), the transformation process for scaled transform coefficients as specified in clause 8.7.4.1 is invoked with the transform block location ( xTbY, yTbY ), the transform block width nTbW and the transform block 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 (8‑1018)

### 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  (8‑1019)

* Otherwise, if cIdx is equal to 1, the following applies:

qP = Qp′Cb (8‑1020)

* Otherwise (cIdx is equal to 2), the following applies:

qP = Qp′Cr (8‑1021)

The variable rectNonTsFlag is derived as follows:

rectNonTsFlag = ( ( ( Log2( nTbW ) + Log2( nTbH ) ) & 1 )  = =  1 && (8‑1022)  
 transform\_skip\_flag[ xTbY ][ yTbY ] = = 0 )

The variables bdShift, rectNorm and bdOffset are derived as follows:

bdShift = bitDepth + ( ( rectNonTsFlag  ?  8  :  0 ) + (8‑1023)  
 ( Log2( nTbW ) + Log2( nTbH ) ) / 2 ) − 5 + dep\_quant\_enabled\_flag

rectNorm = rectNonTsFlag ? 181 : 1 (8‑1024)

bdOffset = ( 1 << bdShift ) >> 1 (8‑1025)

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 ) (8‑1026)

– Otherwise (dep\_quant\_enabled\_flag is equal to 0), the following applies:

ls[ x ][ y ] = ( m[ x ][ y ] \* levelScale[ qP % 6 ] ) << ( qP / 6 ) (8‑1027)

* The value dnc[ x ][ y ] is derived as follows:

dnc[ x ][ y ] = (8‑1028)  
 ( 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 ] ) (8‑1029)

### 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 implicitMtsEnabled is derived as follows:

* If sps\_mts\_enabled\_flag is equal to 1 and one of the following conditions is true, implicitMtsEnabled is set equal to 1:
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT
* cu\_sbt\_flag is equal to 1 and Max( nTbW, nTbH ) is less than or equal to 32
* sps\_explicit\_mts\_intra\_enabled\_flag and sps\_explicit\_mts\_inter\_enabled\_flag are both equal to 0 and CuPredMode[ xTbY ][ yTbY ] is equal to MODE\_INTRA
* Otherwise, implicitMtsEnabled is set equal to 0.

The variable trTypeHor specifying the horizontal transform kernel and the variable trTypeVer specifying the vertical transform kernel are derived as follows:

* If cIdx is greater than 0, trTypeHor and trTypeVer are set equal to 0.
* Otherwise, if implicitMtsEnabled is equal to 1, the following applies:
* If IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT, trTypeHor and trTypeVer are specified in Table 8‑15 depending on intraPredMode.
* Otherwise, if cu\_sbt\_flag is equal to 1, trTypeHor and trTypeVer are specified in Table 8‑14 depending on cu\_sbt\_horizontal\_flag and cu\_sbt\_pos\_flag.
* Otherwise (sps\_explicit\_mts\_intra\_enabled\_flag and sps\_explicit\_mts\_inter\_enabled\_flag are equal to 0), trTypeHor and trTypeVer are derived as follows:

trTypeHor = ( nTbW >= 4 && nTbW <= 16 && nTbW <= nTbH ) ? 1 : 0 (8‑1030)

trTypeVer = ( nTbH >= 4 && nTbH <= 16 && nTbH <= nTbW ) ? 1 : 0 (8‑1031)

* Otherwise, trTypeHor and trTypeVer are specified in Table 8‑13 depending on tu\_mts\_idx[ xTbY ][ yTbY ].

The variables nonZeroW and nonZeroH are derived as follows:

nonZeroW = Min( nTbW, ( trTypeHor > 0 )  ?  16  :  32 ) (8‑1032)

nonZeroH = Min( nTbH, ( trTypeVer > 0 )  ?  16  :  32 ) (8‑1033)

The (nTbW)x(nTbH) array r of residual samples is derived as follows:

1. When nTbH is greater than 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 8.7.4.2 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. When nTbH and nTbW are both greater than 1, 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 ] + 64 ) >> 7 ) (8‑1034)

1. When nTbW is greater than 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 8.7.4.2 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 8‑13 – Specification of trTypeHor and trTypeVer depending on tu\_mts\_idx[ x ][ y ]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **tu\_mts\_idx[ x0 ][ y0 ]** | **0** | **1** | **2** | **3** | **4** |
| trTypeHor | 0 | 1 | 2 | 1 | 2 |
| trTypeVer | 0 | 1 | 1 | 2 | 2 |

Table 8‑14 – Specification of trTypeHor and trTypeVer depending on cu\_sbt\_horizontal\_flag and cu\_sbt\_pos\_flag

|  |  |  |  |
| --- | --- | --- | --- |
| **cu\_sbt\_horizontal\_flag** | **cu\_sbt\_pos\_flag** | trTypeHor | trTypeVer |
| **0** | **0** | 2 | 1 |
| **0** | **1** | 1 | 1 |
| **1** | **0** | 1 | 2 |
| **1** | **1** | 1 | 1 |

Table 8‑15 – Specification of trTypeHor and trTypeVer depending on predModeIntra

|  |  |  |
| --- | --- | --- |
| predModeIntra | trTypeHor | trTypeVer |
| INTRA\_PLANAR,  INTRA\_ANGULAR31,  INTRA\_ANGULAR32,  INTRA\_ANGULAR34,  INTRA\_ANGULAR36,  INTRA\_ANGULAR37 | ( nTbW >= 4 &&  nTbW <= 16 ) ? 1 : 0 | ( nTbH >= 4 &&  nTbH <= 16 ) ? 1 : 0 |
| INTRA\_ANGULAR33,  INTRA\_ANGULAR35 | 0 | 0 |
| INTRA\_ANGULAR2,  INTRA\_ANGULAR4,…,INTRA\_ANGULAR28,  INTRA\_ANGULAR30,  INTRA\_ANGULAR39,  INTRA\_ANGULAR41,…,INTRA\_ANGULAR63,  INTRA\_ANGULAR65 | ( nTbW >= 4 &&  nTbW <= 16 ) ? 1 : 0 | 0 |
| INTRA\_ANGULAR3,  INTRA\_ANGULAR5,…, INTRA\_ANGULAR27,  INTRA\_ANGULAR29,  INTRA\_ANGULAR38,  INTRA\_ANGULAR40,…,INTRA\_ANGULAR64,  INTRA\_ANGULAR66 | 0 | ( nTbH >= 4 &&  nTbH <= 16 ) ? 1 : 0 |

#### 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 8.7.4.3 in invoked with the transform size nTbS and the transform kernel Type trType as inputs, 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 (8‑1035)

* 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 (8‑1036)

#### 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 (8‑1037)

transMatrixCol0to15 = (8‑1038)

{

{ 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 }

{ 91 90 90 90 88 87 86 84 83 81 79 77 73 71 69 65 }

{ 90 90 88 85 82 78 73 67 61 54 46 38 31 22 13 4 }

{ 90 88 84 79 71 62 52 41 28 15 2 −11 −24 −37 −48 −59 }

{ 90 87 80 70 57 43 25 9 −9 −25 −43 −57 −70 −80 −87 −90 }

{ 90 84 73 59 41 20 −2 −24 −44 −62 −77 −86 −90 −90 −83 −71 }

{ 90 82 67 46 22 −4 −31 −54 −73 −85 −90 −88 −78 −61 −38 −13 }

{ 90 79 59 33 2 −28 −56 −77 −88 −90 −81 −62 −37 −7 24 52 }

{ 89 75 50 18 −18 −50 −75 −89 −89 −75 −50 −18 18 50 75 89 }

{ 88 71 41 2 −37 −69 −87 −90 −73 −44 −7 33 65 86 90 77 }

{ 88 67 31 −13 −54 −82 −90 −78 −46 −4 38 73 90 85 61 22 }

{ 87 62 20 −28 −69 −90 −84 −56 −11 37 73 90 81 48 2 −44 }

{ 87 57 9 −43 −80 −90 −70 −25 25 70 90 80 43 −9 −57 −87 }

{ 86 52 −2 −56 −87 −84 −48 7 59 88 83 44 −11 −62 −90 −81 }

{ 85 46 −13 −67 −90 −73 −22 38 82 88 54 −4 −61 −90 −78 −31 }

{ 84 41 −24 −77 −90 −56 7 65 91 69 11 −52 −88 −79 −28 37 }

{ 83 36 −36 −83 −83 −36 36 83 83 36 −36 −83 −83 −36 36 83 }

{ 83 28 −44 −88 −73 −11 59 91 62 −7 −71 −90 −48 24 81 84 }

{ 82 22 −54 −90 −61 13 78 85 31 −46 −90 −67 4 73 88 38 }

{ 81 15 −62 −90 −44 37 88 69 −7 −77 −84 −24 56 91 52 −28 }

{ 80 9 −70 −87 −25 57 90 43 −43 −90 −57 25 87 70 −9 −80 }

{ 79 2 −77 −81 −7 73 83 11 −71 −84 −15 69 86 20 −65 −87 }

{ 78 −4 −82 −73 13 85 67 −22 −88 −61 31 90 54 −38 −90 −46 }

{ 77 −11 −86 −62 33 90 44 −52 −90 −24 69 83 2 −81 −71 20 }

{ 75 −18 −89 −50 50 89 18 −75 −75 18 89 50 −50 −89 −18 75 }

{ 73 −24 −90 −37 65 81 −11 −88 −48 56 86 2 −84 −59 44 90 }

{ 73 −31 −90 −22 78 67 −38 −90 −13 82 61 −46 −88 −4 85 54 }

{ 71 −37 −90 −7 86 48 −62 −79 24 91 20 −81 −59 52 84 −11 }

{ 70 −43 −87 9 90 25 −80 −57 57 80 −25 −90 −9 87 43 −70 }

{ 69 −48 −83 24 90 2 −90 −28 81 52 −65 −71 44 84 −20 −90 }

{ 67 −54 −78 38 85 −22 −90 4 90 13 −88 −31 82 46 −73 −61 }

{ 65 −59 −71 52 77 −44 −81 37 84 −28 −87 20 90 −11 −90 2 }

{ 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 }

{ 62 −69 −56 73 48 −79 −41 83 33 −86 −24 88 15 −90 −7 91 }

{ 61 −73 −46 82 31 −88 −13 90 −4 −90 22 85 −38 −78 54 67 }

{ 59 −77 −37 87 11 −91 15 86 −41 −73 62 56 −79 −33 88 7 }

{ 57 −80 −25 90 −9 −87 43 70 −70 −43 87 9 −90 25 80 −57 }

{ 56 −83 −15 90 −28 −77 65 44 −87 −2 88 −41 −69 73 33 −90 }

{ 54 −85 −4 88 −46 −61 82 13 −90 38 67 −78 −22 90 −31 −73 }

{ 52 −87 7 83 −62 −41 90 −20 −77 71 28 −91 33 69 −79 −15 }

{ 50 −89 18 75 −75 −18 89 −50 −50 89 −18 −75 75 18 −89 50 }

{ 48 −90 28 65 −84 7 79 −73 −15 87 −59 −37 91 −41 −56 88 }

{ 46 −90 38 54 −90 31 61 −88 22 67 −85 13 73 −82 4 78 }

{ 44 −91 48 41 −90 52 37 −90 56 33 −90 59 28 −88 62 24 }

{ 43 −90 57 25 −87 70 9 −80 80 −9 −70 87 −25 −57 90 −43 }

{ 41 −90 65 11 −79 83 −20 −59 90 −48 −33 87 −71 −2 73 −86 }

{ 38 −88 73 −4 −67 90 −46 −31 85 −78 13 61 −90 54 22 −82 }

{ 37 −86 79 −20 −52 90 −69 2 65 −90 56 15 −77 87 −41 −33 }

{ 36 −83 83 −36 −36 83 −83 36 36 −83 83 −36 −36 83 −83 36 }

{ 33 −81 87 −48 −15 71 −90 62 −2 −59 90 −73 20 44 −86 83 }

{ 31 −78 90 −61 4 54 −88 82 −38 −22 73 −90 67 −13 −46 85 }

{ 28 −73 91 −71 24 33 −77 90 −69 20 37 −79 90 −65 15 41 }

{ 25 −70 90 −80 43 9 −57 87 −87 57 −9 −43 80 −90 70 −25 }

{ 24 −65 88 −86 59 −15 −33 71 −90 83 −52 7 41 −77 91 −79 }

{ 22 −61 85 −90 73 −38 −4 46 −78 90 −82 54 −13 −31 67 −88 }

{ 20 −56 81 −91 83 −59 24 15 −52 79 −90 84 −62 28 11 −48 }

{ 18 −50 75 −89 89 −75 50 −18 −18 50 −75 89 −89 75 −50 18 }

{ 15 −44 69 −84 91 −86 71 −48 20 11 −41 65 −83 90 −87 73 }

{ 13 −38 61 −78 88 −90 85 −73 54 −31 4 22 −46 67 −82 90 }

{ 11 −33 52 −69 81 −88 91 −87 79 −65 48 −28 7 15 −37 56 }

{ 9 −25 43 −57 70 −80 87 −90 90 −87 80 −70 57 −43 25 −9 }

{ 7 −20 33 −44 56 −65 73 −81 86 −90 91 −90 87 −83 77 −69 }

{ 4 −13 22 −31 38 −46 54 −61 67 −73 78 −82 85 −88 90 −90 }

{ 2 −7 11 −15 20 −24 28 −33 37 −41 44 −48 52 −56 59 −62 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..63 (8‑1039)

transMatrixCol16to31 = (8‑1040)

{

{ 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 }

{ 62 59 56 52 48 44 41 37 33 28 24 20 15 11 7 2 }

{ −4 −13 −22 −31 −38 −46 −54 −61 −67 −73 −78 −82 −85 −88 −90 −90 }

{ −69 −77 −83 −87 −90 −91 −90 −86 −81 −73 −65 −56 −44 −33 −20 −7 }

{ −90 −87 −80 −70 −57 −43 −25 −9 9 25 43 57 70 80 87 90 }

{ −56 −37 −15 7 28 48 65 79 87 91 88 81 69 52 33 11 }

{ 13 38 61 78 88 90 85 73 54 31 4 −22 −46 −67 −82 −90 }

{ 73 87 90 83 65 41 11 −20 −48 −71 −86 −91 −84 −69 −44 −15 }

{ 89 75 50 18 −18 −50 −75 −89 −89 −75 −50 −18 18 50 75 89 }

{ 48 11 −28 −62 −84 −90 −79 −52 −15 24 59 83 91 81 56 20 }

{ −22 −61 −85 −90 −73 −38 4 46 78 90 82 54 13 −31 −67 −88 }

{ −79 −91 −77 −41 7 52 83 90 71 33 −15 −59 −86 −88 −65 −24 }

{ −87 −57 −9 43 80 90 70 25 −25 −70 −90 −80 −43 9 57 87 }

{ −41 15 65 90 79 37 −20 −69 −90 −77 −33 24 71 91 73 28 }

{ 31 78 90 61 4 −54 −88 −82 −38 22 73 90 67 13 −46 −85 }

{ 83 86 44 −20 −73 −90 −59 2 62 90 71 15 −48 −87 −81 −33 }

{ 83 36 −36 −83 −83 −36 36 83 83 36 −36 −83 −83 −36 36 83 }

{ 33 −41 −87 −77 −15 56 90 65 −2 −69 −90 −52 20 79 86 37 }

{ −38 −88 −73 −4 67 90 46 −31 −85 −78 −13 61 90 54 −22 −82 }

{ −86 −73 −2 71 87 33 −48 −90 −59 20 83 79 11 −65 −90 −41 }

{ −80 −9 70 87 25 −57 −90 −43 43 90 57 −25 −87 −70 9 80 }

{ −24 62 88 28 −59 −90 −33 56 90 37 −52 −90 −41 48 91 44 }

{ 46 90 38 −54 −90 −31 61 88 22 −67 −85 −13 73 82 4 −78 }

{ 88 56 −41 −91 −37 59 87 15 −73 −79 7 84 65 −28 −90 −48 }

{ 75 −18 −89 −50 50 89 18 −75 −75 18 89 50 −50 −89 −18 75 }

{ 15 −79 −69 33 91 28 −71 −77 20 90 41 −62 −83 7 87 52 }

{ −54 −85 4 88 46 −61 −82 13 90 38 −67 −78 22 90 31 −73 }

{ −90 −33 73 69 −41 −88 −2 87 44 −65 −77 28 90 15 −83 −56 }

{ −70 43 87 −9 −90 −25 80 57 −57 −80 25 90 9 −87 −43 70 }

{ −7 88 33 −79 −56 62 73 −41 −86 15 91 11 −87 −37 77 59 }

{ 61 73 −46 −82 31 88 −13 −90 −4 90 22 −85 −38 78 54 −67 }

{ 91 7 −90 −15 88 24 −86 −33 83 41 −79 −48 73 56 −69 −62 }

{ 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 }

{ −2 −90 11 90 −20 −87 28 84 −37 −81 44 77 −52 −71 59 65 }

{ −67 −54 78 38 −85 −22 90 4 −90 13 88 −31 −82 46 73 −61 }

{ −90 20 84 −44 −71 65 52 −81 −28 90 2 −90 24 83 −48 −69 }

{ −57 80 25 −90 9 87 −43 −70 70 43 −87 −9 90 −25 −80 57 }

{ 11 84 −52 −59 81 20 −91 24 79 −62 −48 86 7 −90 37 71 }

{ 73 31 −90 22 78 −67 −38 90 −13 −82 61 46 −88 4 85 −54 }

{ 90 −44 −59 84 2 −86 56 48 −88 11 81 −65 −37 90 −24 −73 }

{ 50 −89 18 75 −75 −18 89 −50 −50 89 −18 −75 75 18 −89 50 }

{ −20 −71 81 2 −83 69 24 −90 52 44 −90 33 62 −86 11 77 }

{ −78 −4 82 −73 −13 85 −67 −22 88 −61 −31 90 −54 −38 90 −46 }

{ −87 65 20 −86 69 15 −84 71 11 −83 73 7 −81 77 2 −79 }

{ −43 90 −57 −25 87 −70 −9 80 −80 9 70 −87 25 57 −90 43 }

{ 28 52 −91 56 24 −84 77 −7 −69 88 −37 −44 90 −62 −15 81 }

{ 82 −22 −54 90 −61 −13 78 −85 31 46 −90 67 4 −73 88 −38 }

{ 84 −81 24 48 −90 71 −7 −62 91 −59 −11 73 −88 44 28 −83 }

{ 36 −83 83 −36 −36 83 −83 36 36 −83 83 −36 −36 83 −83 36 }

{ −37 −28 79 −88 52 11 −69 91 −65 7 56 −90 77 −24 −41 84 }

{ −85 46 13 −67 90 −73 22 38 −82 88 −54 −4 61 −90 78 −31 }

{ −81 90 −62 11 44 −83 88 −59 7 48 −84 87 −56 2 52 −86 }

{ −25 70 −90 80 −43 −9 57 −87 87 −57 9 43 −80 90 −70 25 }

{ 44 2 −48 81 −90 73 −37 −11 56 −84 90 −69 28 20 −62 87 }

{ 88 −67 31 13 −54 82 −90 78 −46 4 38 −73 90 −85 61 −22 }

{ 77 −90 86 −65 33 7 −44 73 −90 87 −69 37 2 −41 71 −88 }

{ 18 −50 75 −89 89 −75 50 −18 −18 50 −75 89 −89 75 −50 18 }

{ −52 24 7 −37 62 −81 90 −88 77 −56 28 2 −33 59 −79 90 }

{ −90 82 −67 46 −22 −4 31 −54 73 −85 90 −88 78 −61 38 −13 }

{ −71 83 −90 90 −86 77 −62 44 −24 2 20 −41 59 −73 84 −90 }

{ −9 25 −43 57 −70 80 −87 90 −90 87 −80 70 −57 43 −25 9 }

{ 59 −48 37 −24 11 2 −15 28 −41 52 −62 71 −79 84 −88 90 }

{ 90 −90 88 −85 82 −78 73 −67 61 −54 46 −38 31 −22 13 −4 }

{ 65 −69 71 −73 77 −79 81 −83 84 −86 87 −88 90 −90 90 −91 }

},

transMatrix[ m ][ n ] = ( n & 1 ? −1 : 1 ) \* transMatrixCol16to31[47 −m ][ n ] (8‑1041)  
 with m =32..47, n = 0..63

transMatrix[ m ][ n ] = ( n & 1 ? −1 : 1 ) \* transMatrixCol0to15[63 −m ][ n ] (8‑1042)  
 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 ] = (8‑1043)

{

{ 29 55 74 84 }

{ 74 74 0 −74 }

{ 84 −29 −74 55 }

{ 55 −84 74 −29 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 8, the following applies:

transMatrix[ m ][ n ] = (8‑1044)

{

{ 17 32 46 60 71 78 85 86 }

{ 46 78 86 71 32 −17 −60 −85 }

{ 71 85 32 −46 −86 −60 17 78 }

{ 85 46 −60 −78 17 86 32 −71 }

{ 86 −17 −85 32 78 −46 −71 60 }

{ 78 −71 −17 85 −60 −32 86 −46 }

{ 60 −86 71 −17 −46 85 −78 32 }

{ 32 −60 78 −86 85 −71 46 −17 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 16, the following applies:

transMatrix[ m ][ n ] = (8‑1045)

{

{ 8 17 25 33 40 48 55 62 68 73 77 81 85 87 88 88 }

{ 25 48 68 81 88 88 81 68 48 25 0 −25 −48 −68 −81 −88 }

{ 40 73 88 85 62 25 −17 −55 −81 −88 −77 −48 −8 33 68 87 }

{ 55 87 81 40 −17 −68 −88 −73 −25 33 77 88 62 8 −48 −85 }

{ 68 88 48 −25 −81 −81 −25 48 88 68 0 −68 −88 −48 25 81 }

{ 77 77 0 −77 −77 0 77 77 0 −77 −77 0 77 77 0 −77 }

{ 85 55 −48 −87 −8 81 62 −40 −88 −17 77 68 −33 −88 −25 73 }

{ 88 25 −81 −48 68 68 −48 −81 25 88 0 −88 −25 81 48 −68 }

{ 88 −8 −88 17 87 −25 −85 33 81 −40 −77 48 73 −55 −68 62 }

{ 87 −40 −68 73 33 −88 8 85 −48 −62 77 25 −88 17 81 −55 }

{ 81 −68 −25 88 −48 −48 88 −25 −68 81 0 −81 68 25 −88 48 }

{ 73 −85 25 55 −88 48 33 −87 68 8 −77 81 −17 −62 88 −40 }

{ 62 −88 68 −8 −55 88 −73 17 48 −87 77 −25 −40 85 −81 33 }

{ 48 −81 88 −68 25 25 −68 88 −81 48 0 −48 81 −88 68 −25 }

{ 33 −62 81 −88 85 −68 40 −8 −25 55 −77 88 −87 73 −48 17 }

{ 17 −33 48 −62 73 −81 87 −88 88 −85 77 −68 55 −40 25 −8 }

},

* 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..15 (8‑1046)

transMatrixCol0to15 = (8‑1047)

{

{ 4 9 13 17 21 26 30 34 38 42 46 50 53 56 60 63 }

{ 13 26 38 50 60 68 77 82 86 89 90 88 85 80 74 66 }

{ 21 42 60 74 84 89 89 84 74 60 42 21 0 −21 −42 −60 }

{ 30 56 77 87 89 80 63 38 9 −21 −50 −72 −85 −90 −84 −68 }

{ 38 68 86 88 74 46 9 −30 −63 −84 −90 −78 −53 −17 21 56 }

{ 46 78 90 77 42 −4 −50 −80 −90 −74 −38 9 53 82 89 72 }

{ 53 85 85 53 0 −53 −85 −85 −53 0 53 85 85 53 0 −53 }

{ 60 89 74 21 −42 −84 −84 −42 21 74 89 60 0 −60 −89 −74 }

{ 66 90 56 −13 −74 −87 −46 26 80 84 34 −38 −85 −78 −21 50 }

{ 72 86 34 −46 −89 −63 13 78 82 21 −56 −90 −53 26 84 77 }

{ 77 80 9 −72 −84 −17 66 86 26 −60 −88 −34 53 90 42 −46 }

{ 80 72 −17 −86 −60 34 90 46 −50 −89 −30 63 85 13 −74 −78 }

{ 84 60 −42 −89 −21 74 74 −21 −89 −42 60 84 0 −84 −60 42 }

{ 86 46 −63 −78 21 90 26 −77 −66 42 87 4 −85 −50 60 80 }

{ 88 30 −78 −56 60 77 −34 −87 4 89 26 −80 −53 63 74 −38 }

{ 90 13 −87 −26 84 38 −78 −50 72 60 −63 −68 53 77 −42 −82 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15 (8‑1048)

transMatrixCol16to31 = (8‑1049)

{

{ 66 68 72 74 77 78 80 82 84 85 86 87 88 89 90 90 }

{ 56 46 34 21 9 −4 −17 −30 −42 −53 −63 −72 −78 −84 −87 −90 }

{ −74 −84 −89 −89 −84 −74 −60 −42 −21 0 21 42 60 74 84 89 }

{ −46 −17 13 42 66 82 90 86 74 53 26 −4 −34 −60 −78 −88 }

{ 80 90 82 60 26 −13 −50 −77 −89 −85 −66 −34 4 42 72 87 }

{ 34 −13 −56 −84 −88 −68 −30 17 60 85 87 66 26 −21 −63 −86 }

{ −85 −85 −53 0 53 85 85 53 0 −53 −85 −85 −53 0 53 85 }

{ −21 42 84 84 42 −21 −74 −89 −60 0 60 89 74 21 −42 −84 }

{ 88 72 9 −60 −90 −63 4 68 89 53 −17 −77 −86 −42 30 82 }

{ 9 −66 −88 −42 38 87 68 −4 −74 −85 −30 50 90 60 −17 −80 }

{ −90 −50 38 89 56 −30 −87 −63 21 85 68 −13 −82 −74 4 78 }

{ 4 82 68 −21 −87 −56 38 90 42 −53 −88 −26 66 84 9 −77 }

{ 89 21 −74 −74 21 89 42 −60 −84 0 84 60 −42 −89 −21 74 }

{ −17 −90 −30 74 68 −38 −88 −9 84 53 −56 −82 13 89 34 −72 }

{ −86 9 90 21 −82 −50 66 72 −42 −85 13 90 17 −84 −46 68 }

{ 30 86 −17 −89 4 90 9 −88 −21 85 34 −80 −46 74 56 −66 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 4, the following applies:

transMatrix[ m ][ n ] = (8‑1050)

{

{ 84 74 55 29 }

{ 74 0 −74 −74 }

{ 55 −74 −29 84 }

{ 29 −74 84 −55 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 8, the following applies:

transMatrix[ m ][ n ] = (8‑1051)

{

{ 86 85 78 71 60 46 32 17 }

{ 85 60 17 −32 −71 −86 −78 −46 }

{ 78 17 −60 −86 −46 32 85 71 }

{ 71 −32 −86 −17 78 60 −46 −85 }

{ 60 −71 −46 78 32 −85 −17 86 }

{ 46 −86 32 60 −85 17 71 −78 }

{ 32 −78 85 −46 −17 71 −86 60 }

{ 17 −46 71 −85 86 −78 60 −32 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 16, the following applies:

transMatrix[ m ][ n ] = (8‑1052)

{

{ 88 88 87 85 81 77 73 68 62 55 48 40 33 25 17 8 }

{ 88 81 68 48 25 0 −25 −48 −68 −81 −88 −88 −81 −68 −48 −25 }

{ 87 68 33 −8 −48 −77 −88 −81 −55 −17 25 62 85 88 73 40 }

{ 85 48 −8 −62 −88 −77 −33 25 73 88 68 17 −40 −81 −87 −55 }

{ 81 25 −48 −88 −68 0 68 88 48 −25 −81 −81 −25 48 88 68 }

{ 77 0 −77 −77 0 77 77 0 −77 −77 0 77 77 0 −77 −77 }

{ 73 −25 −88 −33 68 77 −17 −88 −40 62 81 −8 −87 −48 55 85 }

{ 68 −48 −81 25 88 0 −88 −25 81 48 −68 −68 48 81 −25 −88 }

{ 62 −68 −55 73 48 −77 −40 81 33 −85 −25 87 17 −88 −8 88 }

{ 55 −81 −17 88 −25 −77 62 48 −85 −8 88 −33 −73 68 40 −87 }

{ 48 −88 25 68 −81 0 81 −68 −25 88 −48 −48 88 −25 −68 81 }

{ 40 −88 62 17 −81 77 −8 −68 87 −33 −48 88 −55 −25 85 −73 }

{ 33 −81 85 −40 −25 77 −87 48 17 −73 88 −55 −8 68 −88 62 }

{ 25 −68 88 −81 48 0 −48 81 −88 68 −25 −25 68 −88 81 −48 }

{ 17 −48 73 −87 88 −77 55 −25 −8 40 −68 85 −88 81 −62 33 }

{ 8 −25 40 −55 68 −77 85 −88 88 −87 81 −73 62 −48 33 −17 }

},

* 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..15 (8‑1053)

transMatrixCol0to15 = (8‑1054)

{

{ 90 90 89 88 87 86 85 84 82 80 78 77 74 72 68 66 }

{ 90 87 84 78 72 63 53 42 30 17 4 −9 −21 −34 −46 −56 }

{ 89 84 74 60 42 21 0 −21 −42 −60 −74 −84 −89 −89 −84 −74 }

{ 88 78 60 34 4 −26 −53 −74 −86 −90 −82 −66 −42 −13 17 46 }

{ 87 72 42 4 −34 −66 −85 −89 −77 −50 −13 26 60 82 90 80 }

{ 86 63 21 −26 −66 −87 −85 −60 −17 30 68 88 84 56 13 −34 }

{ 85 53 0 −53 −85 −85 −53 0 53 85 85 53 0 −53 −85 −85 }

{ 84 42 −21 −74 −89 −60 0 60 89 74 21 −42 −84 −84 −42 21 }

{ 82 30 −42 −86 −77 −17 53 89 68 4 −63 −90 −60 9 72 88 }

{ 80 17 −60 −90 −50 30 85 74 4 −68 −87 −38 42 88 66 −9 }

{ 78 4 −74 −82 −13 68 85 21 −63 −87 −30 56 89 38 −50 −90 }

{ 77 −9 −84 −66 26 88 53 −42 −90 −38 56 87 21 −68 −82 −4 }

{ 74 −21 −89 −42 60 84 0 −84 −60 42 89 21 −74 −74 21 89 }

{ 72 −34 −89 −13 82 56 −53 −84 9 88 38 −68 −74 30 90 17 }

{ 68 −46 −84 17 90 13 −85 −42 72 66 −50 −82 21 90 9 −86 }

{ 66 −56 −74 46 80 −34 −85 21 88 −9 −90 −4 89 17 −86 −30 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15 (8‑1055)

transMatrixCol16to31 = (8‑1056)

{

{ 63 60 56 53 50 46 42 38 34 30 26 21 17 13 9 4 }

{ −66 −74 −80 −85 −88 −90 −89 −86 −82 −77 −68 −60 −50 −38 −26 −13 }

{ −60 −42 −21 0 21 42 60 74 84 89 89 84 74 60 42 21 }

{ 68 84 90 85 72 50 21 −9 −38 −63 −80 −89 −87 −77 −56 −30 }

{ 56 21 −17 −53 −78 −90 −84 −63 −30 9 46 74 88 86 68 38 }

{ −72 −89 −82 −53 −9 38 74 90 80 50 4 −42 −77 −90 −78 −46 }

{ −53 0 53 85 85 53 0 −53 −85 −85 −53 0 53 85 85 53 }

{ 74 89 60 0 −60 −89 −74 −21 42 84 84 42 −21 −74 −89 −60 }

{ 50 −21 −78 −85 −38 34 84 80 26 −46 −87 −74 −13 56 90 66 }

{ −77 −84 −26 53 90 56 −21 −82 −78 −13 63 89 46 −34 −86 −72 }

{ −46 42 90 53 −34 −88 −60 26 86 66 −17 −84 −72 9 80 77 }

{ 78 74 −13 −85 −63 30 89 50 −46 −90 −34 60 86 17 −72 −80 }

{ 42 −60 −84 0 84 60 −42 −89 −21 74 74 −21 −89 −42 60 84 }

{ −80 −60 50 85 −4 −87 −42 66 77 −26 −90 −21 78 63 −46 −86 }

{ −38 74 63 −53 −80 26 89 4 −87 −34 77 60 −56 −78 30 88 }

{ 82 42 −77 −53 68 63 −60 −72 50 78 −38 −84 26 87 −13 −90 }

},

### Picture reconstruction process

#### General

Inputs to this process are:

* a location ( xCurr, yCurr ) specifying the top-left sample of the current block relative to the top‑left sample of the current picture component,
* the variables nCurrSw and nCurrSh specifying the width and height, respectively, of the current block,
* a variable cIdx specifying the colour component of the current block,
* an (nCurrSw) x (nCurrSh) array predSamples specifying the predicted samples of the current block,
* an (nCurrSw) x (nCurrSh) array resSamples specifying the residual samples of the current block.

Output of this process is a reconstructed picture sample array recSamples.

Depending on the value of the colour component cIdx, the following assignments are made:

* If cIdx is equal to 0, recSamples corresponds to the reconstructed picture sample array SL and the function clipCidx1 corresponds to Clip1Y.
* Otherwise, if cIdx is equal to 1, recSamples corresponds to the reconstructed chroma sample array SCb and the function clipCidx1 corresponds to Clip1C.
* Otherwise (cIdx is equal to 2), recSamples corresponds to the reconstructed chroma sample array SCr and the function clipCidx1 corresponds to Clip1C.

Depending on the value of tile\_group\_lmcs\_enabled\_flag, the following applies:

* If tile\_group\_lmcs\_enabled\_flag is equal to 0, the (nCurrSw)x(nCurrSh) block of the reconstructed samples recSamples at location ( xCurr, yCurr ) is derived as follows for i = 0..nCurrSw − 1, j = 0..nCurrSh − 1:

recSamples[ xCurr + i ][ yCurr + j ] = clipCidx1( predSamples[ i ][ j ] + resSamples[ i ][ j ] ) (8‑1057)

* Otherwise (tile\_group\_lmcs\_enabled\_flag is equal to 1), the following applies:
* If cIdx is equal to 0, the following applies:
* The picture reconstruction with mapping process for luma samples as specified in clause 8.7.5.2 is invoked with the luma location ( xCurr, yCurr ), the block width nCurrSw and height nCurrSh, the predicted luma sample array predSamples, and the residual luma sample array resSamples as inputs, and the output is the reconstructed luma sample array recSamples.
* The picture reconstruction with inverse mapping process for luma samples as specified in clause 8.7.5.3 is invoked with the reconstructed luma sample array recSamples as inputs, and the output is the modified reconstructed luma sample array recSamples.
* Otherwise (cIdx is greater than 0), the picture reconstruction with luma dependent chroma residual scaling process for chroma samples as specified in clause 8.7.5.4 is invoked with the chroma location ( xCurr, yCurr ), the transform block width nCurrSw and height nCurrSh, the predicted chroma sample array predSamples, and the residual chroma sample array resSamples as inputs, and the output is the reconstructed chroma sample array recSamples.

#### Picture reconstruction with mapping process for luma samples

Inputs to this process are:

* a location ( xCurr, yCurr ) of the top-left sample of the current block relative to the top-left sample of the current picture,
* a variable nCurrSw specifying the block width,
* a variable nCurrSh specifying the block height,
* an (nCurrSw)x(nCurrSh) array predSamples specifying the luma predicted samples of the current block,
* an (nCurrSw)x(nCurrSh) array resSamples specifying the luma residual samples of the current block.

Outputs of this process is:

* a reconstructed luma picture sample array recSamples.

The (nCurrSw)x(nCurrSh) array of mapped predicted luma samples PredMapSamples, which is used in the picture reconstruction with luma dependent chroma residual scaling process for chroma samples as specified in clause 8.7.5.4, is derived as follows:

* If one of the following conditions is true, PredMapSamples[ i ][ j ] is set equal to predSamples[ i ][ j ] for i = 0..nCurrSw − 1, j = 0..nCurrSh − 1:
* CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_INTRA.
* CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_IBC.
* CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_INTER and ciip\_flag[ xCurr ][ yCurr ] is equal to 1.
* Otherwise (CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_INTER and ciip\_flag[ xCurr ][ yCurr ] is equal to 0), the following applies:

idxY = predSamples[ i ][ j ] >> Log2( OrgCW )  
PredMapSamples[ i ][ j ] =  LmcsPivot[ idxY ]   
  + ( ScaleCoeff[ idxY ] \* ( predSamples[ i ][ j ] − InputPivot[ idxY ] ) + ( 1 << 13 ) ) >> 14  (8‑1058)  
 with i = 0..nCurrSw − 1, j = 0..nCurrSh − 1

The reconstructed luma picture sample recSamples is derived as follows:

recSamples[ xCurr + i ][ yCurr + j ] = Clip1Y ( PredMapSamples[ i ][ j ]+ resSamples[ i ][ j ] ] ) (8‑1059)  
 with i = 0..nCurrSw − 1, j = 0..nCurrSh − 1

#### Picture reconstruction with inverse mapping process for luma sample

Input to this process is a reconstructed picture luma sample array SL.

The output to this process is a modified reconstructed picture luma sample array SL.

The inverse mapping process for a luma sample SL[ x ][ y ] with x = 0..pic\_width\_in\_luma\_samples − 1, y = 0..pic\_height\_in\_luma\_samples − 1 is invoked as specified in clause 8.7.5.3.1 with SL[ x ][ y ] as the input and the output is the modified luma sample SL[ x ][ y ].

##### Picture inverse mapping process of luma samples

Input to this process is a luma sample lumaSample.

Output of this process is a modified luma sample invLumaSample .

The value of invLumaSample is derived by applying the following ordered steps:

1. The variable idxYInv is derived by invoking the identification of piece-wise function index as specified in clause 8.7.5.3.2 with lumaSample as the input] and idxYInv as the output.
2. The variable invSample is derived as follows:

invSample = InputPivot[ idxYInv ] +  ( InvScaleCoeff[ idxYInv ] \*  (8‑1060)  
 ( lumaSample[ xP ][ yP ] − LmcsPivot[ idxYInv ] ) + ( 1 << 13 ) ) >> 14

1. The inverse mapped luma sample invLumaSample is derived as follows:

invLumaSample = ClipRange? Clip3(LmcsMinVal, LmcsMaxVal, invSample) :  
 ClipCidx1(invSample) (8‑1061)

##### Identification of piecewise function index for luma components

Input to this process is a luma sample S.

Output of this process is an index idxS identifing the piece to which the luma sample S belongs.

The variable idxS is derived as follows:

if ( S < LmcsPivot[ lmcs\_min\_bin\_idx + 1 ] )  
 idxS = lmcs\_min\_bin\_idx  
else if ( S >= LmcsPivot[ LmcsMaxBinIdx ] )   
 idxS = LmcsMaxBinIdx  
else { (8‑1062)  
 for( idxS = lmcs\_min\_bin\_idx; idxS < LmcsMaxBinIdx; idxS++ ) {  
 if( S < LmcsPivot [ idxS + 1 ] )  
 break  
 }  
}

#### Picture reconstruction with luma dependent chroma residual scaling process for chroma samples

Inputs to this process are:

* a location ( xCurr, yCurr ) of the top-left sample of the current transform block relative to the top-left sample of the current picture,
* a variable nCurrSw specifying the transform block width,
* a variable nCurrSh specifying the transform block height,
* an (nCurrSw)x(nCurrSh) array predSamples specifying the chroma prediction samples of the current block,
* an (nCurrSw)x(nCurrSh) array resSamples specifying the chroma residual samples of the current block.

Output of this process is a reconstructed chroma picture sample array recSamples.

The reconstructed chroma picture sample recSamples is derived as follows for i = 0..nCurrSw − 1, j = 0..nCurrSh − 1:

* If tile\_group\_chroma\_residual\_scale\_flag is equal to 0 or  nCurrSw \* nCurrSh is less than or euqal to 4, the following applies:

recSamples[ xCurr + i ][ yCurr + j ] = Clip1C ( predSamples[ i ][ j ] + resSamples[ i ][ j ] )  (8‑1063)

* Otherwise (tile\_group\_chroma\_residual\_scale\_flag is equal to 1 and nCurrSw \* nCurrSh is greater than 4), the following applies:
* For the derivation of the variable varScale the following ordered steps apply:

1. The variable invAvgLuma is derived as follows:

invAvgLuma = Clip1Y( (  (8‑1064)  + nCurrSw \* nCurrSh \*2 ) / ( nCurrSw \* nCurrSh \*4 ) )

1. The variable idxYInv is derived by invoking the identification of piece-wise function index as specified in clause 8.7.5.3.2 with invAvgLuma as the input and idxYInv as the output.
2. The variable varScale is derived as follows:

varScale = ChromaScaleCoeff[ idxYInv ] (8‑1065)

* The recSamples is derived as follows:
* If tu\_cbf\_cIdx [ xCurr ][ yCurr ] equal to 1, the following applies:

recSamples[ xCurr + i ][ yCurr + j ] = ClipCidx1( predSamples[ i ][ j ] +  (8‑1066)  
 Sign( resSamples[ i ][ j ] ) \* ( ( Abs( resSamples[ i ][ j ] ) \* varScale + ( 1 << 10 ) ) >> 11 ) )

* Otherwise (tu\_cbf\_cIdx[ xCurr ][ yCurr ] equal to 0), the following applies:

recSamples[ xCurr + i ][ yCurr + j ] = ClipCidx1(predSamples[ i ][ j ] ) (8‑1067)

## In-loop Filter Process

### General

The three in-loop filters, namely deblocking filter, sample adaptive offset 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 8.8.2.1 is invoked with the reconstructed picture sample array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr as inputs, and the modified reconstructed picture sample array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr after deblocking as outputs.

– The array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr are assigned to the array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr (which represent the decoded picture), respectively.

1. When sps\_sao\_enabled\_flag is equal to 1, the following applies:

– The sample adaptive offset process as specified in clause 8.8.3.1 is invoked with the reconstructed picture sample array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr as inputs, and the modified reconstructed picture sample array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr after sample adaptive offset as outputs.

– The array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr are assigned to the array SL and, when ChromaArrayType is not equal to 0, the arrays 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 8.8.4.1 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

Inputs to this process are the reconstructed picture prior to deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr.

The vertical edges in a picture are filtered first. Then the horizontal edges in a picture are filtered with samples modified by the vertical edge filtering process as input. The vertical and horizontal edges in the CTBs of each CTU are processed separately on a coding unit basis. The vertical edges of the coding blocks in a coding unit are filtered starting with the edge on the left-hand side of the coding blocks proceeding through the edges towards the right-hand side of the coding blocks in their geometrical order. The horizontal edges of the coding blocks in a coding unit are filtered starting with the edge on the top of the coding blocks proceeding through the edges towards the bottom of the coding blocks in their geometrical order.

NOTE – Although the filtering process is specified on a picture basis in this Specification, the filtering process can be implemented on a coding unit basis with an equivalent result, provided the decoder properly accounts for the processing dependency order so as to produce the same output values.

The deblocking filter process is applied to all coding subblock edges and transform block edges of a picture, except the following types of edges:

– Edges that are at the boundary of the picture,

– Edges that coincide with tile boundaries when loop\_filter\_across\_tiles\_enabled\_flag is equal to 0,

– Edges that coincide with upper or left boundaries of tile groups with tile\_group\_loop\_filter\_across\_tile\_groups\_enabled\_flag equal to 0 or tile\_group\_deblocking\_filter\_disabled\_flag equal to 1,

– Edges within tile groups with tile\_group\_deblocking\_filter\_disabled\_flag equal to 1,

– Edges that do not correspond to 8x8 sample grid boundaries of the considered component,

– Edges within chroma components for which both sides of the edge use inter prediction,

– Edges of chroma transform blocks that are not edges of the associated transform unit.

– Edges across the luma transform blocks of a coding unit that has an IntraSubPartitionsSplit value not equal to ISP\_NO\_SPLIT.

[Ed. (BB): Adapt syntax once tiles are integrated.]

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 8‑16.

Table 8‑16 – Name of association to edgeType

|  |  |
| --- | --- |
| edgeType | Name of edgeType |
| 0 (vertical edge) | EDGE\_VER |
| 1 (horizontal edge) | EDGE\_HOR |

When tile\_group\_deblocking\_filter\_disabled\_flag of the current tile group is equal to 0, the following applies:

* The variable treeType is derived as follows:
* If tile\_group\_type is equal to I and qtbtt\_dual\_tree\_intra\_flag is equal to 1, treeType is set equal to DUAL\_TREE\_LUMA.
* Otherwise, treeType is set equal to SINGLE\_TREE.
* The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.2.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr as outputs.
* The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.2.2 with the variable treeType, the modifed reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr as outputs.
* When tile\_group\_type is equal to I and qtbtt\_dual\_tree\_intra\_flag is equal to 1, the following applies:
* The variable treeType is set equal to DUAL\_TREE\_CHROMA
* The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.2.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays recPictureCb and recPictureCr as outputs.
* The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.2.2 with the variable treeType, the modifed reconstructed picture after deblocking, i.e., the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays recPictureCb and recPictureCr as outputs.

#### Deblocking filter process for one direction

Inputs to this process are:

* the variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed,
* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the reconstructed picture prior to deblocking, i.e., the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr.

For each coding unit with coding block width log2CbW, coding block height log2CbH and location of top-left sample of the coding block ( xCb, yCb ), when edgeType is equal to EDGE\_VER and xCb % 8 is equal 0 or when edgeType is equal to EDGE\_HOR and yCb % 8 is equal to 0, the edges are filtered by the following ordered steps:

1. The coding block width nCbW is set equal to 1  <<  log2CbW and the coding block height nCbH is set equal to 1  <<  log2CbH
2. The variable filterEdgeFlag is derived as follows:

* If edgeType is equal to EDGE\_VER and one or more of the following conditions are true, filterEdgeFlag is set equal to 0:
* The left boundary of the current coding block is the left boundary of the picture.
* The left boundary of the current coding block is the left boundary of the tile and loop\_filter\_across\_tiles\_enabled\_flag is equal to 0.
* The left boundary of the current coding block is the left boundary of the tile group and tile\_group\_loop\_filter\_across\_tile\_groups\_enabled\_flag is equal to 0.
* Otherwise if edgeType is equal to EDGE\_HOR and one or more of the following conditions are true, the variable filterEdgeFlag is set equal to 0:
* The top boundary of the current luma coding block is the top boundary of the picture.
* The top boundary of the current coding block is the top boundary of the tile and loop\_filter\_across\_tiles\_enabled\_flag is equal to 0.
* The top boundary of the current coding block is the top boundary of the tile group and tile\_group\_loop\_filter\_across\_tile\_groups\_enabled\_flag is equal to 0.
* Otherwise, filterEdgeFlag is set equal to 1.

[Ed. (BB): Adapt syntax once tiles are integrated.]

1. All elements of the two-dimensional (nCbW)x(nCbH) array edgeFlags are initialized to be equal to zero.
2. The derivation process of transform block boundary specified in clause 8.8.2.3 is invoked with the location ( xB0, yB0 ) set equal to ( 0, 0 ), the block width nTbW set equal to nCbW, the block height nTbH set equal to nCbH, the variable treeType, the variable filterEdgeFlag, the array edgeFlags, and the variable edgeType as inputs, and the modified array edgeFlags as output.
3. The derivation process of coding subblock boundary specified in clause 8.8.2.4 is invoked with the location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, the array edgeFlags, and the variable edgeType as inputs, and the modified array edgeFlags as output.
4. The picture sample array recPicture is derived as follows:

* If treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, recPicture is set equal to the reconstructed luma picture sample array prior to deblocking recPictureL.
* Otherwise (treeType is equal to DUAL\_TREE\_CHROMA), recPicture is set equal to the reconstructed chroma picture sample array prior to deblocking recPictureCb.

1. The derivation process of the boundary filtering strength specified in clause 8.8.2.5 is invoked with the picture sample array recPicture, the luma location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, the variable edgeType, and the array edgeFlags as inputs, and an (nCbW)x(nCbH) array verBs as output.
2. The edge filtering process is invoked as follows:

* If edgeType is equal to EDGE\_VER, the vertical edge filtering process for a coding unit as specified in clause 8.8.2.6.1 is invoked with the variable treeType, the reconstructed picture prior to deblocking, i.e., when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr, the location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, and the array verBs as inputs, and the modified reconstructed picture, i.e., when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr, as output.
* Otherwise if edgeType is equal to EDGE\_HOR, the horizontal edge filtering process for a coding unit as specified in clause 8.8.2.6.2 is invoked with the variable treeType, the modified reconstructed picture prior to deblocking, i.e., when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr, the location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, and the array horBs as inputs and the modified reconstructed picture, i.e., when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr, as output.

#### Derivation process of transform block boundary

Inputs to this process are:

* a location ( xB0, yB0 ) specifying the top-left sample of the current block relative to the top‑left sample of the current coding block,
* a variable nTbW specifying the width of the current block,
* a variable nTbH specifying the height of the current block,
* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed,
* a variable filterEdgeFlag,
* a two-dimensional (nCbW)x(nCbH) array edgeFlags,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbW)x(nCbH) array edgeFlags.

The maximum transform block size maxTbSize is derived as follows:

maxTbSize = ( treeType  = =  DUAL\_TREE\_CHROMA ) ? MaxTbSizeY / 2 : MaxTbSizeY (8‑1068)

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 newTbW and newTbH are derived as follows:

newTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑1069)

newTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑1070)

1. The derivation process of transform block boundary as specified in this clause is invoked with the location ( xB0, yB0 ), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags, and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.
2. If nTbW is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location ( xTb0, yTb0 ) set equal to ( xTb0 + newTbW, yTb0 ), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.
3. If nTbH is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location ( xTb0, yTb0 ) set equal to ( xTb0, yTb0 + newTbH ), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.
4. If nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location ( xTb0, yTb0 ) set equal to ( xTb0 + newTbW, yTb0 + newTbH ), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

– Otherwise, the following applies:

* If edgeType is equal to EDGE\_VER, the value of edgeFlags[ xB0 ][ yB0 + k ] for k = 0..nTbH − 1 is derived as follows:
* If xB0 is equal to 0, edgeFlags[ xB0 ][ yB0 + k ] is set equal to filterEdgeFlag.
* Otherwise, edgeFlags[ xB0 ][ yB0 + k ] is set equal to 1.
* Otherwise (edgeType is equal to EDGE\_HOR), the value of edgeFlags[ xB0 + k ][ yB0 ] for k = 0..nTbW − 1 is derived as follows:
* If yB0 is equal to 0, edgeFlags[ xB0 + k ][ yB0 ] is set equal to filterEdgeFlag.
* Otherwise, edgeFlags[ xB0 + k ][ yB0 ] is set equal to 1.

#### Derivation process of coding subblock boundary

Inputs to this process are:

* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block,
* a two-dimensional (nCbW)x(nCbH) array edgeFlags,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbW)x(nCbH) array edgeFlags.

The number of coding subblock in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

* If CuPredMode[ xCb ][ yCb ]  !=  MODE\_INTER, numSbX and numSbY are both set equal to 1.
* Otherwise, numSbX and numSbY are set equal to NumSbX[ xCb ][ yCb ] and NumSbY[ xCb ][ yCb ], respectively.

Depending on the value of edgeType the following applies:

* If edgeType is equal to EDGE\_VER and numSbX is greater than 1, the following applies for i = 1..min( ( nCbW / 8 ) − 1, numSbX − 1), k = 0..nCbH − 1:

edgeFlags[ i \* Max( 8, nCbW / numSbX ) ][ k ] = 1 (8‑1071)

* Otherwise if edgeType is equal to EDGE\_HOR and numSbY is greater than 1, the following applies for j = 1..min( ( nCbH / 8 ) − 1, numSbY − 1 ), k = 0..nCbW − 1:

edgeFlags[ k ][ j \* Max( 8, nCbH / numSbY ) ] = 1 (8‑1072)

#### Derivation process of boundary filtering strength

Inputs to this process are:

* a picture sample array recPicture,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,
* a two-dimensional (nCbW)x(nCbH) array edgeFlags.

Output of this process is a two-dimensional (nCbW)x(nCbH) array bS specifying the boundary filtering strength.

The variables xDi, yDj, xN and yN are derived as follows:

* If edgeType is equal to EDGE\_VER, xDi is set equal to ( i  <<  3 ), yDj is set equal to ( j  <<  2 ), xN is set equal to Max( 0, ( nCbW / 8 ) − 1 ) and yN is set equal to ( nCbH / 4 ) − 1.
* Otherwise (edgeType is equal to EDGE\_HOR), xDi is set equal to ( i  <<  2 ), yDj is set equal to ( j  <<  3 ), xN is set equal to ( nCbW / 4 ) − 1 and yN is set equal to Max( 0, ( nCbH / 8 ) − 1 ).

For xDi with i = 0..xN and yDj with j = 0..yN, the following applies:

* If edgeFlags[ xDi ][ yDj ] is equal to 0, the variable bS[ xDi ][ yDj ] is set equal to 0.
* Otherwise, the following applies:
* The sample values p0 and q0 are derived as follows:
  + - If edgeType is equal to EDGE\_VER, p0 is set equal to recPicture [ xCb + xDi − 1 ][ yCb + yDj ] and q0 is set equal to recPicture [ xCb + xDi ][ yCb + yDj ].
    - Otherwise (edgeType is equal to EDGE\_HOR), p0 is set equal to recPicture [ xCb + xDi ][ yCb + yDj − 1 ] and q0 is set equal to recPicture [ xCb + xDi ][ yCb + yDj ].
* The variable bS[ xDi ][ yDj ] is derived as follows:
  + - If the sample p0 or q0 is in the coding block of a coding unit coded with intra prediction mode, bS[ xDi ][ yDj ] is set equal to 2.
    - Otherwise, if the block edge is also a transform block edge and the sample p0 or q0 is in a transform block which contains one or more non-zero transform coefficient levels, bS[ xDi ][ yDj ] is set equal to 1.
    - Otherwise, if the prediction mode of the coding subblock containing the sample p0 is different from the prediction mode of the coding subblock containing the sample q0, bS[ xDi ][ yDj ] is set equal to 1.
    - Otherwise, if one or more of the following conditions are true, bS[ xDi ][ yDj ] is set equal to 1:
      * The coding subblock containing the sample p0 and the coding subblock containing the sample q0 are both coded in IBC prediction mode, and the absolute difference between the horizontal or vertical component of the motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in units of quarter luma samples.
      * For the prediction of the coding subblock containing the sample p0 different reference pictures or a different number of motion vectors are used than for the prediction of the coding subblock containing the sample q0.

NOTE 1 – The determination of whether the reference pictures used for the two coding sublocks are the same or different is based only on which pictures are referenced, without regard to whether a prediction is formed using an index into reference picture list 0 or an index into reference picture list 1, and also without regard to whether the index position within a reference picture list is different.

NOTE 2 – The number of motion vectors that are used for the prediction of a coding subblock with top-left sample covering ( xSb, ySb ), is equal to PredFlagL0[ xSb ][ ySb ] + PredFlagL1[ xSb ][ ySb ].

* + - * One motion vector is used to predict the coding subblock containing the sample p0 and one motion vector is used to predict the coding subblock containing the sample q0, and the absolute difference between the horizontal or vertical component of the motion vectors used is greater than or equal to 4 in units of quarter luma samples.
      * Two motion vectors and two different reference pictures are used to predict the coding subblock containing the sample p0, two motion vectors for the same two reference pictures are used to predict the coding subblock containing the sample q0 and the absolute difference between the horizontal or vertical component of the two motion vectors used in the prediction of the two coding subblocks for the same reference picture is greater than or equal to 4 in units of quarter luma samples.
      * Two motion vectors for the same reference picture are used to predict the coding subblock containing the sample p0, two motion vectors for the same reference picture are used to predict the coding subblock containing the sample q0 and both of the following conditions are true:
        + The absolute difference between the horizontal or vertical component of list 0 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in quarter luma samples, or the absolute difference between the horizontal or vertical component of the list 1 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in units of quarter luma samples.
        + The absolute difference between the horizontal or vertical component of list 0 motion vector used in the prediction of the coding subblock containing the sample p0 and the list 1 motion vector used in the prediction of the coding subblock containing the sample q0 is greater than or equal to 4 in units of quarter luma samples, or the absolute difference between the horizontal or vertical component of the list 1 motion vector used in the prediction of the coding subblock containing the sample p0 and list 0 motion vector used in the prediction of the coding subblock containing the sample q0 is greater than or equal to 4 in units of quarter luma samples.
    - Otherwise, the variable bS[ xDi ][ yDj ] is set equal to 0.

#### Edge filtering process

##### Vertical edge filtering process

Inputs to this process are:

* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed,
* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the reconstructed picture prior to deblocking, i.e., the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr.

When treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max( 0, ( nCbW / 8 ) − 1 ) and yN is set equal to ( nCbH / 4 ) − 1.
2. For xDk equal to k  <<  3 with k = 0..nN and yDm equal to m  <<  2 with m = 0..yN, the following applies:

* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply:

1. The decision process for block edges as specified in clause 8.8.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the luma picture sample array recPictureL, the location of the luma coding block ( xCb, yCb ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER,the boundary filtering strength bS[ xDk ][ yDm ], and the bit depth bD set equal to BitDepthY as inputs, and the decisions dE, dEp and dEq, and the variable tC as outputs.
2. The filtering process for block edges as specified in clause 8.8.2.6.4 is invoked with the picture sample array recPicture set equal to the luma picture sample array recPictureL, the location of the luma coding block ( xCb, yCb ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, the decisions dE, dEp and dEq, and the variable tC as inputs, and the modified luma picture sample array recPictureL as output.

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE the filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max( 0, ( nCbW / 8 ) − 1 ) and yN is set equal to Max( 0, ( nCbH / 8 ) − 1 ).
2. The variable edgeSpacing is set equal to 8 / SubWidthC.
3. The variable edgeSections is set equal to yN \* ( 2 / SubHeightC ).
4. For xDk equal to k \* edgeSpacing with k = 0..xN and yDm equal to m  <<  2 with m = 0..edgeSections, the following applies:

* When bS[ xDk \* SubWidthC ][ yDm \* SubHeightC ] is equal to 2 and ( ( ( xCb / SubWidthC + xDk )  >>  3 )  <<  3 ) is equal to xCb / SubWidthC + xDk, the following ordered steps apply:

1. The filtering process for chroma block edges as specified in clause 8.8.2.6.5 is invoked with the chroma picture sample array recPictureCb, the location of the chroma coding block ( xCb / SubWidthC, yCb / SubHeightC ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER and a variable cQpPicOffset set equal to pps\_cb\_qp\_offset as inputs, and the modified chroma picture sample array recPictureCb as output.
2. The filtering process for chroma block edges as specified in clause 8.8.2.6.5 is invoked with the chroma picture sample array recPictureCr, the location of the chroma coding block ( xCb / SubWidthC, yCb / SubHeightC ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER and a variable cQpPicOffset set equal to pps\_cr\_qp\_offset as inputs, and the modified chroma picture sample array recPictureCr as output.

When treeType is equal to DUAL\_TREE\_CHROMA, the filtering process for edges in the two chroma coding blocks of the current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max( 0, ( nCbW / 8 ) − 1 ) and yN is set equal to ( nCbH / 4 ) − 1.
2. For xDk equal to k  <<  3 with k = 0..xN and yDm equal to m  <<  2 with m = 0..yN, the following applies:

* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply:

1. The decision process for block edges as specified in clause 8.8.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the chroma picture sample array recPictureCb, the location of the chroma coding block ( xCb, yCb ), the location of the chroma block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, the boundary filtering strength bS[ xDk ][ yDm ], and the bit depth bD set equal to BitDepthC as inputs, and the decisions dE, dEp and dEq, and the variable tC as outputs.
2. The filtering process for block edges as specified in clause 8.8.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array recPictureCb, the location of the chroma coding block ( xCb, yCb ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, the decisions dE, dEp and dEq, and the variable tC as inputs, and the modified chroma picture sample array recPictureCb as output.
3. The filtering process for block edges as specified in clause 8.8.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array recPictureCr, the location of the chroma coding block ( xCb, yCb ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, the decisions dE, dEp and dEq, and the variable tC as inputs, and the modified chroma picture sample array recPictureCr as output.

##### Horizontal edge filtering process

Inputs to this process are:

* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed,
* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the reconstructed picture prior to deblocking, i.e., the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr.

When treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable yN is set equal to Max( 0, ( nCbH / 8 ) − 1 ) and xN is set equal to ( nCbW / 4 ) − 1.
2. For yDm equal to m  <<  3 with m = 0..yN and xDk equal to k  <<  2 with k = 0..xN, the following applies:

* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply:

1. The decision process for block edges as specified in clause 8.8.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the luma picture sample array recPictureL, the location of the luma coding block ( xCb, yCb ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, the boundary filtering strength bS[ xDk ][ yDm ], and the bit depth bD set equal to BitDepthY as inputs, and the decisions dE, dEp and dEq, and the variable tC as outputs.
2. The filtering process for block edges as specified in clause 8.8.2.6.4 is invoked with the picture sample array recPicture set equal to the luma picture sample array recPictureL, the location of the luma coding block ( xCb, yCb ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, the decisions dEp, dEp and dEq, and the variable tC as inputs, and the modified luma picture sample array recPictureL as output.

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE the filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max( 0, ( nCbW / 8 ) − 1 ) and yN is set equal to Max( 0, ( nCbH / 8 ) − 1 ).
2. The variable edgeSpacing is set equal to 8 / SubHeightC.
3. The variable edgeSections is set equal to xN \* ( 2 / SubWidthC ).
4. For yDm equal to m \* edgeSpacing with m = 0..yN and xDk equal to k  <<  2 with k = 0..edgeSections, the following applies:

* When bS[ xDk \* SubWidthC ][ yDm \* SubHeightC ] is equal to 2 and ( ( ( yCb / SubHeightC + yDm )  >>  3 )  <<  3 ) is equal to yCb / SubHeightC + yDm, the following ordered steps apply:

1. The filtering process for chroma block edges as specified in clause 8.8.2.6.5 is invoked with the chroma picture sample array recPictureCb, the location of the chroma coding block ( xCb / SubWidthC, yCb / SubHeightC ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR and a variable cQpPicOffset set equal to pps\_cb\_qp\_offset as inputs, and the modified chroma picture sample array recPictureCb as output.
2. The filtering process for chroma block edges as specified in clause 8.8.2.6.5 is invoked with the chroma picture sample array recPictureCr, the location of the chroma coding block ( xCb / SubWidthC, yCb / SubHeightC ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR and a variable cQpPicOffset set equal to pps\_cr\_qp\_offset as inputs, and the modified chroma picture sample array recPictureCr as output.

When treeType is equal to DUAL\_TREE\_CHROMA, the filtering process for edges in the two chroma coding blocks of the current coding unit consists of the following ordered steps:

1. The variable yN is set equal to Max( 0, ( nCbH / 8 ) − 1 ) and xN is set equal to ( nCbW / 4 ) − 1.
2. For yDm equal to m  <<  3 with m = 0..yN and xDk equal to k  <<  2 with k = 0..xN, the following applies:

* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply:

1. The decision process for block edges as specified in clause 8.8.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the chroma picture sample array recPictureCb, the location of the chroma coding block ( xCb, yCb ), the location of the chroma block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, the boundary filtering strength bS[ xDk ][ yDm ], and the bit depth bD set equal to BitDepthC as inputs, and the decisions dE, dEp and dEq, and the variable tC as outputs.
2. The filtering process for block edges as specified in clause 8.8.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array recPictureCb, the location of the chroma coding block ( xCb, yCb ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, the decisions dE, dEp and dEq, and the variable tC as inputs, and the modified chroma picture sample array recPictureCb as output.
3. The filtering process for block edges as specified in clause 8.8.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array recPictureCr, the location of the chroma coding block ( xCb, yCb ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, the decisions dE, dEp and dEq, and the variable tC as inputs, and the modified chroma picture sample array recPictureCr as output.

##### Decision process for block edges

Inputs to this process are:

* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed,
* a picture sample array recPicture,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a location ( xBl, yBl ) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,
* a variable bS specifying the boundary filtering strength,
* a variable bD specifying the bit depth of the current component.

Outputs of this process are:

* the variables dE, dEp and dEq containing decisions,
* the variable tC.

If edgeType is equal to EDGE\_VER, the sample values pi,k and qi,k with i = 0..3 and k = 0 and 3 are derived as follows:

qi,k = recPictureL[ xCb + xBl + i ][ yCb + yBl + k ] (8‑1073)

pi,k = recPictureL[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1074)

Otherwise (edgeType is equal to EDGE\_HOR), the sample values pi,k and qi,k with i = 0..3 and k = 0 and 3 are derived as follows:

qi,k = recPicture[ xCb + xBl + k ][ yCb + yBl + i ] (8‑1075)

pi,k = recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1076)

The variable qpOffset is derived as follows:

* If sps\_ladf\_enabled\_flag is equal to 1 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the following applies:
* The variable lumaLevel of the reconstructed luma level is derived as follow:

lumaLevel = ( ( p0,0 + p0,3 + q0,0 + q0,3 ) >> 2 ), (8‑1077)

* The variable qpOffset is set equal to sps\_ladf\_lowest\_interval\_qp\_offset and modified as follows:

for( i = 0; i < sps\_num\_ladf\_intervals\_minus2 + 1; i++ ) {  
 if( lumaLevel > SpsLadfIntervalLowerBound[ i + 1 ] )  
 qpOffset = sps\_ladf\_qp\_offset[ i ] (8‑1078)  
 else  
 break  
}

* Otherwise (treeType is equal to DUAL\_TREE\_CHROMA), qpOffset is set equal to 0.

The variables QpQ and QpP are derived as follows:

* If treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, QpQ and QpP are set equal to the QpY values of the coding units which include the coding blocks containing the sample q0,0 and p0,0, respectively.
* Otherwise (treeType is equal to DUAL\_TREE\_CHROMA), QpQ and QpP are set equal to the QpC values of the coding units which include the coding blocks containing the sample q0,0 and p0,0, respectively.

The variable qP is derived as follows:

qP = ( ( QpQ + QpP + 1 )  >>  1 ) + qpOffset (8‑1079)

The value of the variable β′ is determined as specified in Table 8‑17 based on the quantization parameter Q derived as follows:

Q = Clip3( 0, 63, qP + ( tile\_group\_beta\_offset\_div2  <<  1 ) ) (8‑1080)

where tile\_group\_beta\_offset\_div2 is the value of the syntax element tile\_group\_beta\_offset\_div2 for the tile group that contains sample q0,0.

The variable β is derived as follows:

β = β′ \* ( 1  <<  ( bD− 8 ) ) (8‑1081)

The value of the variable tC′ is determined as specified in Table 8‑17 based on the quantization parameter Q derived as follows:

Q = Clip3( 0, 65, qP + 2 \* ( bS − 1 ) + ( tile\_group\_tc\_offset\_div2  <<  1 ) ) (8‑1082)

where tile\_group\_tc\_offset\_div2 is the value of the syntax element tile\_group\_tc\_offset\_div2 for the tile group that contains sample q0,0.

The variable tC is derived as follows:

tC = tC′ \* ( 1  <<  ( bD − 8 ) ) (8‑1083)

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE\_VER, the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

dp0 = Abs( p2,0 − 2 \* p1,0 + p0,0 ) (8‑1084)

dp3 = Abs( p2,3 − 2 \* p1,3 + p0,3 ) (8‑1085)

dq0 = Abs( q2,0 − 2 \* q1,0 + q0,0 ) (8‑1086)

dq3 = Abs( q2,3 − 2 \* q1,3 + q0,3 ) (8‑1087)

dpq0 = dp0 + dq0 (8‑1088)

dpq3 = dp3 + dq3 (8‑1089)

dp = dp0 + dp3 (8‑1090)

dq = dq0 + dq3 (8‑1091)

d = dpq0 + dpq3 (8‑1092)

1. The variables dE, dEp and dEq are set equal to 0.
2. When d is less than β, the following ordered steps apply:
3. The variable dpq is set equal to 2 \* dpq0.
4. For the sample location ( xCb + xBl, yCb + yBl ), the decision process for a sample as specified in clause 8.8.2.6.6 is invoked with sample values p0,0, p3,0, q0,0, and q3,0, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam0.
5. The variable dpq is set equal to 2 \* dpq3.
6. For the sample location ( xCb + xBl, yCb + yBl + 3 ), the decision process for a sample as specified in clause 8.8.2.6.6 is invoked with sample values p0,3, p3,3, q0,3, and q3,3, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam3.
7. The variable dE is set equal to 1.
8. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
9. When dp is less than ( β + ( β  >>  1 ) )  >>  3, the variable dEp is set equal to 1.
10. When dq is less than ( β + ( β  >>  1 ) )  >>  3, the variable dEq is set equal to 1.

– Otherwise (edgeType is equal to EDGE\_HOR), the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

dp0 = Abs( p2,0 − 2 \* p1,0 + p0,0 ) (8‑1093)

dp3 = Abs( p2,3 − 2 \* p1,3 + p0,3 ) (8‑1094)

dq0 = Abs( q2,0 − 2 \* q1,0 + q0,0 ) (8‑1095)

dq3 = Abs( q2,3 − 2 \* q1,3 + q0,3 ) (8‑1096)

dpq0 = dp0 + dq0 (8‑1097)

dpq3 = dp3 + dq3 (8‑1098)

dp = dp0 + dp3 (8‑1099)

dq = dq0 + dq3 (8‑1100)

d = dpq0 + dpq3 (8‑1101)

1. The variables dE, dEp and dEq are set equal to 0.
2. When d is less than β, the following ordered steps apply:
3. The variable dpq is set equal to 2 \* dpq0.
4. For the sample location ( xCb + xBl, yCb + yBl ), the decision process for a sample as specified in clause 8.8.2.6.6 is invoked with sample values p0,0, p3,0, q0,0 and q3,0, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam0.
5. The variable dpq is set equal to 2 \* dpq3.
6. For the sample location ( xCb + xBl + 3, yCb + yBl ), the decision process for a sample as specified in clause 8.8.2.6.6 is invoked with sample values p0,3, p3,3, q0,3 and q3,3, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam3.
7. The variable dE is set equal to 1.
8. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
9. When dp is less than ( β + ( β  >>  1 ) )  >>  3, the variable dEp is set equal to 1.
10. When dq is less than ( β + ( β  >>  1 ) )  >>  3, the variable dEq is set equal to 1.

Table 8‑17 – Derivation of threshold variables β′ and tC′ from input Q

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Q** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **β**′ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| **tC**′ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Q** | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 |
| **β**′ | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 20 | 22 | 24 | 26 | 28 |
| **tC**′ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
| **Q** | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| **β**′ | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| **tC**′ | 3 | 4 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 8 | 9 | 10 | 11 | 13 | 14 | 16 | 18 |
| **Q** | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 |  |  |
| **β**′ | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 | 80 | 82 | 84 | 86 | 88 | - | - |  |  |
| **tC**′ | 20 | 22 | 25 | 28 | 31 | 35 | 39 | 44 | 50 | 56 | 63 | 70 | 79 | 88 | 99 |  |  |

##### Filtering process for block edges

Inputs to this process are:

– a picture sample array recPicture,

– a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,

– a location ( xBl, yBl ) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,

– the variables dE, dEp and dEq containing decisions,

– the variable tC.

Output of this process is the modified picture sample array recPicture.

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE\_VER, the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0..3 are derived as follows:

qi,k = recPictureL[ xCb + xBl + i ][ yCb + yBl + k ] (8‑1102)

pi,k = recPictureL[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1103)

1. When dE is not equal to 0, for each sample location ( xCb + xBl, yCb + yBl + k ), k = 0..3, the following ordered steps apply:
2. The filtering process for a sample as specified in clause 8.8.2.6.7 is invoked with the sample values pi,k, qi,k with i = 0..3, the locations ( xPi, yPi ) set equal to ( xCb + xBl − i − 1, yCb + yBl + k ) and ( xQi, yQi ) set equal to ( xCb + xBl + i, yCb + yBl + k ) with i = 0..2, the decision dE, the variables dEp and dEq and the variable tC as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values pi' and qj' as outputs.
3. When nDp is greater than 0, the filtered sample values pi' with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl − i − 1 ][ yCb + yBl + k ] = pi' (8‑1104)

1. When nDq is greater than 0, the filtered sample values qj' with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + j ][ yCb + yBl + k ] = qj' (8‑1105)

– Otherwise (edgeType is equal to EDGE\_HOR), the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0..3 are derived as follows:

qi,k = recPictureL[ xCb + xBl + k ][ yCb + yBl + i ] (8‑1106)

pi,k = recPictureL[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1107)

1. When dE is not equal to 0, for each sample location ( xCb + xBl + k, yCb + yBl ), k = 0..3, the following ordered steps apply:
2. The filtering process for a sample as specified in clause 8.8.2.6.7 is invoked with the sample values pi,k, qi,k with i = 0..3, the locations ( xPi, yPi ) set equal to ( xCb + xBl + k, yCb + yBl − i − 1 ) and ( xQi, yQi ) set equal to ( xCb + xBl + k, yCb + yBl + i ) with i = 0..2, the decision dE, the variables dEp and dEq, and the variable tC as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values pi' and qj' as outputs.
3. When nDp is greater than 0, the filtered sample values pi' with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] = pi' (8‑1108)

1. When nDq is greater than 0, the filtered sample values qj' with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl + j ] = qj' (8‑1109)

##### Filtering process for chroma block edges

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

– a chroma picture sample array s′,

– a chroma location ( xCb, yCb ) specifying the top-left sample of the current chroma coding block relative to the top-left chroma sample of the current picture,

– a chroma location ( xBl, yBl ) specifying the top-left sample of the current chroma block relative to the top-left sample of the current chroma coding block,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,

– a variable cQpPicOffset specifying the picture-level chroma quantization parameter offset.

Output of this process is the modified chroma picture sample array s′.

If edgeType is equal to EDGE\_VER, the values pi and qi with i = 0..1 and k = 0..3 are derived as follows:

qi,k = s′[ xCb + xBl + i ][ yCb + yBl + k ] (8‑1110)

pi,k = s′[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1111)

Otherwise (edgeType is equal to EDGE\_HOR), the sample values pi and qi with i = 0..1 and k = 0..3 are derived as follows:

qi,k = s′[ xCb + xBl + k ][ yCb + yBl + i ] (8‑1112)

pi,k = s′[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1113)

The variables QpQ and QpP are set equal to the QpY values of the coding units which include the coding blocks containing the sample q0,0 and p0,0, respectively.

If ChromaArrayType is equal to 1, the variable QpC is determined as specified in Table 8‑12 based on the index qPi derived as follows:

qPi = ( ( QpQ + QpP + 1 )  >>  1 ) + cQpPicOffset (8‑1114)

Otherwise (ChromaArrayType is greater than 1), the variable QpC is set equal to Min( qPi, 63 ).

NOTE – The variable cQpPicOffset provides an adjustment for the value of pps\_cb\_qp\_offset or pps\_cr\_qp\_offset, according to whether the filtered chroma component is the Cb or Cr component. However, to avoid the need to vary the amount of the adjustment within the picture, the filtering process does not include an adjustment for the value of tile\_group\_cb\_qp\_offset or tile\_group\_cr\_qp\_offset.

The value of the variable tC′ is determined as specified in Table 8‑17 based on the chroma quantization parameter Q derived as follows:

Q = Clip3( 0, 65, QpC + 2 + ( tile\_group\_tc\_offset\_div2  <<  1 ) ) (8‑1115)

where tile\_group\_tc\_offset\_div2 is the value of the syntax element tile\_group\_tc\_offset\_div2 for the tile group that contains sample q0,0.

The variable tC is derived as follows:

tC = tC′ \* ( 1  <<  ( BitDepthC − 8 ) ) (8‑1116)

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE\_VER, for each sample location ( xCb + xBl, yCb + yBl + k ), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.8.2.6.8 is invoked with the sample values pi,k, qi,k, with i = 0..1, the locations ( xCb + xBl − 1, yCb + yBl + k ) and ( xCb + xBl, yCb + yBl + k ) and the variable tC as inputs, and the filtered sample values p0′ and q0′ as outputs.
2. The filtered sample values p0′ and q0′ replace the corresponding samples inside the sample array s′ as follows:

s′[ xCb + xBl ][ yCb + yBl + k ] = q0′ (8‑1117)

s′[ xCb + xBl − 1 ][ yCb + yBl + k ] = p0′ (8‑1118)

– Otherwise (edgeType is equal to EDGE\_HOR), for each sample location ( xCb + xBl + k, yCb + yBl ), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.8.2.6.8 is invoked with the sample values pi,k, qi,k, with i = 0..1, the locations ( xCb + xBl + k, yCb + yBl − 1 ) and ( xCb + xBl + k, yCb + yBl ), and the variable tC as inputs, and the filtered sample values p0′ and q0′ as outputs.
2. The filtered sample values p0′ and q0′ replace the corresponding samples inside the sample array s′ as follows:

s′[ xCb + xBl + k ][ yCb + yBl ] = q0′ (8‑1119)

s′[ xCb + xBl + k ][ yCb + yBl − 1 ] = p0′ (8‑1120)

##### Decision process for a sample

Inputs to this process are:

– the sample values p0, p3, q0 and q3,

– the variables dpq, β and tC.

Output of this process is the variable dSam containing a decision.

The variable dSam is specified as follows:

– If dpq is less than ( β  >>  2 ), Abs( p3 − p0 ) + Abs( q0 − q3 ) is less than ( β  >>  3 ) and Abs( p0 − q0 ) is less than ( 5 \* tC + 1 )  >>  1, dSam is set equal to 1.

– Otherwise, dSam is set equal to 0.

##### Filtering process for a sample

Inputs to this process are:

– the sample values pi and qi with i = 0..3,

– the locations of pi and qi, ( xPi, yPi ) and ( xQi, yQi ) with i = 0..2,

– a variable dE,

– the variables dEp and dEq containing decisions to filter samples p1 and q1, respectively,

– a variable tC.

Outputs of this process are:

– the number of filtered samples nDp and nDq,

– the filtered sample values pi′ and qj′ with i = 0..nDp − 1, j = 0..nDq − 1.

Depending on the value of dE, the following applies:

– If the variable dE is equal to 2, nDp and nDq are both set equal to 3 and the following strong filtering applies:

p0′ = Clip3( p0 − 2 \* tC, p0 + 2 \* tC, ( p2 + 2 \* p1 + 2 \* p0 + 2 \* q0 + q1 + 4 )  >>  3 ) (8‑1121)

p1′ = Clip3( p1 − 2 \* tC, p1 + 2 \* tC, ( p2 + p1 + p0 + q0 + 2 )  >>  2 ) (8‑1122)

p2′ = Clip3( p2 − 2 \* tC, p2 + 2\*tC, ( 2 \* p3 + 3 \* p2 + p1 + p0 + q0 + 4 )  >>  3 ) (8‑1123)

q0′ = Clip3( q0 − 2 \* tC, q0 + 2 \* tC, ( p1 + 2 \* p0 + 2 \* q0 + 2 \* q1 + q2 + 4 )  >>  3 ) (8‑1124)

q1′ = Clip3( q1 − 2 \* tC, q1 + 2 \* tC, ( p0 + q0 + q1 + q2 + 2 )  >>  2 ) (8‑1125)

q2′= Clip3( q2 − 2 \* tC, q2 + 2 \* tC, ( p0 + q0 + q1 + 3 \* q2 + 2 \* q3 + 4 )  >>  3 ) (8‑1126)

– Otherwise, nDp and nDq are set both equal to 0 and the following weak filtering applies:

* + The following applies:

Δ = ( 9 \* ( q0 −  p0 ) − 3 \* ( q1 − p1 ) + 8 )  >>  4 (8‑1127)

* + When Abs(Δ) is less than tC \* 10, the following ordered steps apply:
    - The filtered sample values p0′ and q0′ are specified as follows:

Δ = Clip3( −tC, tC, Δ ) (8‑1128)

p0′ = Clip1Y( p0 + Δ ) (8‑1129)

q0′ = Clip1Y( q0 − Δ ) (8‑1130)

* + - When dEp is equal to 1, the filtered sample value p1′ is specified as follows:

Δp = Clip3( −( tC  >>  1 ), tC  >>  1, ( ( ( p2 + p0 + 1 )  >>  1 ) − p1 + Δ )  >>  1 ) (8‑1131)

p1′ = Clip1Y( p1 + Δp ) (8‑1132)

* + - When dEq is equal to 1, the filtered sample value q1′ is specified as follows:

Δq = Clip3( −( tC  >>  1 ), tC  >>  1, ( ( ( q2 + q0 + 1 )  >>  1 ) − q1 − Δ )  >>  1 ) (8‑1133)

q1′ = Clip1Y( q1 + Δq ) (8‑1134)

* + - nDp is set equal to dEp + 1 and nDq is set equal to dEq + 1.

When nDp is greater than 0 and one or more of the following conditions are true, nDp is set equal to 0:

– pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xP0 ][ yP0 ] is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample p0 is equal to 1.

When nDq is greater than 0 and one or more of the following conditions are true, nDq is set equal to 0:

– pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xQ0 ][ yQ0 ] is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample q0 is equal to 1.

##### Filtering process for a chroma sample

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

– the chroma sample values pi and qi with i = 0..1,

– the chroma locations of p0 and q0, ( xP0, yP0 ) and ( xQ0, yQ0 ),

– a variable tC.

Outputs of this process are the filtered sample values p0′ and q0′.

The filtered sample values p0′ and q0′ are derived as follows:

Δ = Clip3( −tC, tC, ( ( ( ( q0 − p0 )  <<  2 ) + p1 − q1 + 4 )  >>  3 ) ) (8‑1135)

p0′ = Clip1C( p0 + Δ ) (8‑1136)

q0′ = Clip1C( q0 − Δ ) (8‑1137)

When one or more of the following conditions are true, the filtered sample value, p0′ is substituted by the corresponding input sample value p0:

– pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xP0 \* SubWidthC ][ yP0 \* SubHeightC ] is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample p0 is equal to 1.

When one or more of the following conditions are true, the filtered sample value, q0′ is substituted by the corresponding input sample value q0:

– pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xQ0 \* SubWidthC ][ yQ0 \* SubHeightC ] is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample q0 is equal to 1.

### Sample adaptive offset process

#### General

Inputs to this process are the reconstructed picture sample array prior to sample adaptive offset recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture sample array after sample adaptive offset saoPictureL and, when ChromaArrayType is not equal to 0, the arrays saoPictureCb and saoPictureCr.

This process is performed on a CTB basis after the completion of the deblocking filter process for the decoded picture.

The sample values in the modified reconstructed picture sample array saoPictureL and, when ChromaArrayType is not equal to 0, the arrays saoPictureCb and saoPictureCr are initially set equal to the sample values in the reconstructed picture sample array recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr, respectively.

For every CTU with CTB location ( rx, ry ), where rx = 0..PicWidthInCtbsY − 1 and ry = 0..PicHeightInCtbsY − 1, the following applies:

– When tile\_group\_sao\_luma\_flag of the current tile group is equal to 1, the CTB modification process as specified in clause 8.8.3.2 is invoked with recPicture set equal to recPictureL, cIdx set equal to 0, ( rx, ry ), and both nCtbSw and nCtbSh set equal to CtbSizeY as inputs, and the modified luma picture sample array saoPictureL as output.

– When ChromaArrayType is not equal to 0 and tile\_group\_sao\_chroma\_flag of the current tile group is equal to 1, the CTB modification process as specified in clause 8.8.3.2 is invoked with recPicture set equal to recPictureCb, cIdx set equal to 1, ( rx, ry ), nCtbSw set equal to ( 1  <<  CtbLog2SizeY ) / SubWidthC and nCtbSh set equal to ( 1  <<  CtbLog2SizeY ) / SubHeightC as inputs, and the modified chroma picture sample array saoPictureCb as output.

– When ChromaArrayType is not equal to 0 and tile\_group\_sao\_chroma\_flag of the current tile group is equal to 1, the CTB modification process as specified in clause 8.8.3.2 is invoked with recPicture set equal to recPictureCr, cIdx set equal to 2, ( rx, ry ), nCtbSw set equal to ( 1  <<  CtbLog2SizeY ) / SubWidthC and nCtbSh set equal to ( 1  <<  CtbLog2SizeY ) / SubHeightC as inputs, and the modified chroma picture sample array saoPictureCr as output.

#### CTB modification process

Inputs to this process are:

– the picture sample array recPicture for the colour component cIdx,

– a variable cIdx specifying the colour component index,

– a pair of variables ( rx, ry ) specifying the CTB location,

– the CTB width nCtbSw and height nCtbSh.

Output of this process is a modified picture sample array saoPicture for the colour component cIdx.

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 location ( xCtb, yCtb ), specifying the top-left sample of the current CTB for the colour component cIdx relative to the top-left sample of the current picture component cIdx, is derived as follows:

( xCtb, yCtb ) = ( rx \* nCtbSw, ry \* nCtbSh ) (8‑1138)

The sample locations inside the current CTB are derived as follows:

( xSi, ySj ) = ( xCtb + i, yCtb + j ) (8‑1139)

( xYi, yYj ) = ( cIdx  = =  0 ) ? ( xSi, ySj ) : ( xSi \* SubWidthC, ySj \* SubHeightC ) (8‑1140)

For all sample locations ( xSi, ySj ) and ( xYi, yYj ) with i = 0..nCtbSw − 1 and j = 0..nCtbSh − 1, depending on the values of pcm\_loop\_filter\_disabled\_flag, pcm\_flag[ xYi ][ yYj ] and cu\_transquant\_bypass\_flag of the coding unit which includes the coding block covering recPicture[ xSi ][ ySj ], the following applies:

– If one or more of the following conditions are true, saoPicture[ xSi ][ ySj ] is not modified:

* pcm\_loop\_filter\_disabled\_flag and pcm\_flag[ xYi ][ yYj ] are both equal to 1.
* cu\_transquant\_bypass\_flag is equal to 1.
* SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 0.

[Ed. (BB): Modify highlighted sections prending on future decision transform/quantizaion bypass.]

– Otherwise, if SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2, the following ordered steps apply:

1. The values of hPos[ k ] and vPos[ k ] for k = 0..1 are specified in Table 8‑18 based on SaoEoClass[ cIdx ][ rx ][ ry ].
2. The variable edgeIdx is derived as follows:

* The modified sample locations ( xSik′, ySjk′ ) and ( xYik′, yYjk′ ) are derived as follows:

( xSik′, ySjk′ ) = ( xSi + hPos[ k ], ySj + vPos[ k ] ) (8‑1141)

( xYik′, yYjk′ ) = ( cIdx  = =  0 ) ? ( xSik′, ySjk′ ) : ( xSik′ \* SubWidthC, ySjk′ \* SubHeightC ) (8‑1142)

* If one or more of the following conditions for all sample locations ( xSik′, ySjk′ ) and ( xYik′, yYjk′ ) with k = 0..1 are true, edgeIdx is set equal to 0:
* The sample at location ( xSik′, ySjk′ ) is outside the picture boundaries.
* The sample at location ( xSik′, ySjk′ ) belongs to a different tile group and one of the following two conditions is true:
* MinTbAddrZs[ xYik′  >>  MinTbLog2SizeY ][ yYjk′  >>  MinTbLog2SizeY ] is less than MinTbAddrZs[ xYi  >>  MinTbLog2SizeY ][ yYj  >>  MinTbLog2SizeY ] and tile\_group\_loop\_filter\_across\_tile\_groups\_enabled\_flag in the tile group which the sample recPicture[ xSi ][ ySj ] belongs to is equal to 0.
* MinTbAddrZs[ xYi  >>  MinTbLog2SizeY ][ yYj  >>  MinTbLog2SizeY ] is less than MinTbAddrZs[ xYik′  >>  MinTbLog2SizeY ][ yYjk′  >>  MinTbLog2SizeY ] and tile\_group\_loop\_filter\_across\_tile\_groups\_enabled\_flag in the tile group which the sample recPicture[ xSik′ ][ ySjk′ ] belongs to is equal to 0.
* loop\_filter\_across\_tiles\_enabled\_flag is equal to 0 and the sample at location ( xSik′, ySjk′ ) belongs to a different tile.

[Ed. (BB): Modify highlighted sections when tiles without tile groups are incorporated]

* Otherwise, edgeIdx is derived as follows:
* The following applies:

edgeIdx = 2 + Sign( recPicture[ xSi ][ ySj ] − recPicture[ xSi + hPos[ 0 ] ][ ySj + vPos[ 0 ] ] ) +  
 Sign( recPicture[ xSi ][ ySj ] − recPicture[ xSi + hPos[ 1 ] ][ ySj + vPos[ 1 ] ] ) (8‑1143)

* When edgeIdx is equal to 0, 1, or 2, edgeIdx is modified as follows:

edgeIdx = ( edgeIdx = = 2 ) ? 0 : ( edgeIdx + 1 ) (8‑1144)

1. The modified picture sample array saoPicture[ xSi ][ ySj ] is derived as follows:

saoPicture[ xSi ][ ySj ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, recPicture[ xSi ][ ySj ] +  
 SaoOffsetVal[ cIdx ][ rx ][ ry ][ edgeIdx ] ) (8‑1145)

* Otherwise (SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 1), the following ordered steps apply:

1. The variable bandShift is set equal to bitDepth − 5.
2. The variable saoLeftClass is set equal to sao\_band\_position[ cIdx ][ rx ][ ry ].
3. The list bandTable is defined with 32 elements and all elements are initially set equal to 0. Then, four of its elements (indicating the starting position of bands for explicit offsets) are modified as follows:

for( k = 0; k < 4; k++ )  
 bandTable[ ( k + saoLeftClass ) & 31 ] = k + 1 (8‑1146)

1. The variable bandIdx is set equal to bandTable[ recPicture[ xSi ][ ySj ]  >>  bandShift ].
2. The modified picture sample array saoPicture[ xSi ][ ySj ] is derived as follows:

saoPicture[ xSi ][ ySj ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, recPicture[ xSi ][ ySj ] +  
 SaoOffsetVal[ cIdx ][ rx ][ ry ][ bandIdx ] ) (8‑1147)

Table 8‑18 – Specification of hPos and vPos according to the sample adaptive offset class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SaoEoClass[ cIdx ][ rx ][ ry ] | 0 | 1 | 2 | 3 |
| hPos[ 0 ] | −1 | 0 | −1 | 1 |
| hPos[ 1 ] | 1 | 0 | 1 | −1 |
| vPos[ 0 ] | 0 | −1 | −1 | −1 |
| vPos[ 1 ] | 0 | 1 | 1 | 1 |

### 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, respectively.

When tile\_group\_alf\_enabled\_flag is equal to 1, 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 8.8.4.2 is invoked with recPictureL, alfPictureL, and the luma coding tree block location ( xCtb, yCtb ) set equal to ( rx  <<  CtbLog2SizeY, ry  <<  CtbLog2SizeY ) 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 8.8.4.4 is invoked with recPicture set equal to recPictureCb, alfPicture set equal to alfPictureCb, and the chroma coding tree block location ( xCtbC, yCtbC ) set equal to ( rx  <<  ( CtbLog2SizeY − 1 ), ry  <<  ( CtbLog2SizeY − 1 ) ) 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 8.8.4.4 is invoked with recPicture set equal to recPictureCr, alfPicture set equal to alfPictureCr, and the chroma coding tree block location ( xCtbC, yCtbC ) set equal to ( rx  <<  ( CtbLog2SizeY − 1 ), ry  <<  ( CtbLog2SizeY − 1 ) ) as inputs, and the output is the modified filtered picture alfPictureCr.

#### 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.

Output of this process is the modified filtered reconstructed luma picture sample array alfPictureL.

The derivation process for filter index clause 8.8.4.3 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 ] = AlfCoeffL[ filtIdx[ x ][ y ] ][ j ] (8‑1148)

* + 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] } (8‑1149)

* + 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] } (8‑1150)

* + 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] } (8‑1151)

* + 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] } (8‑1152)

* + 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 ) (8‑1153)

vy = Clip3( 0, pic\_height\_in\_luma\_samples − 1, yCtb + y ) (8‑1154)

* + The variable sum is derived as follows:

sum = filterCoeff[ 0 ]   \* ( recPictureL[ hx, vy + 3 ] + recPictureL[ hx, vy − 3 ] ) +   
 filterCoeff[ 1 ]   \* ( recPictureL[ hx + 1, vy + 2 ] + recPictureL[ hx − 1, vy − 2 ] ) +   
 filterCoeff[ 2 ]   \* ( recPictureL[ hx, vy + 2 ] + recPictureL[ hx, vy − 2 ] ) +   
 filterCoeff[ 3 ]   \* ( recPictureL[ hx − 1, vy + 2 ] + recPictureL[ hx + 1, vy − 2 ] ) +   
 filterCoeff[ 4 ]   \* ( recPictureL[ hx + 2, vy + 1 ] + recPictureL[ hx − 2, vy − 1 ] ) +   
 filterCoeff[ 5 ]   \* ( recPictureL[ hx + 1, vy + 1 ] + recPictureL[ hx − 1, vy − 1 ] ) +   
 filterCoeff[ 6 ]   \* ( recPictureL[ hx, vy + 1 ] + recPictureL[ hx, vy − 1 ] ) + (8‑1155)  
 filterCoeff[ 7 ]   \* ( recPictureL[ hx − 1, vy + 1 ] + recPictureL[ hx + 1, vy − 1 ] ) +   
 filterCoeff[ 8 ]   \* ( recPictureL[ hx − 2, vy + 1 ] + recPictureL[ hx + 2, vy − 1 ] ) +   
 filterCoeff[ 9 ]   \* ( recPictureL[ hx + 3, vy ] + recPictureL[ hx − 3, vy ] ) +   
 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 + 64 ) >> 7 (8‑1156)

* + 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 ) (8‑1157)

#### 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 ) (8‑1158)

vy = Clip3( 0, pic\_height\_in\_luma\_samples − 1, y ) (8‑1159)

The classification filter index array filtIdx and the transpose index array transposeIdx are derived by the following ordered steps:

1. The variables filtH[ x ][ y ], filtV[ x ][ y ], filtD0[ x ][ y ] and filtD1[ x ][ y ] with x, y = − 2..CtbSizeY + 1 are derived as follows:

* If both x and y are even numbers or both x and y are uneven numbers, the following applies:

filtH[ x ][ y ] = Abs(  ( recPicture[ hxCtb+x, vyCtb+y ] << 1 ) − recPicture[ hxCtb+x−1, vyCtb+y ] −  (8‑1160)  
  recPicture[ hxCtb+x+1, vyCtb+y ]  )

filtV[ x ][ y ] = Abs(  ( recPicture[ hxCtb+x, vyCtb+y ] << 1 ) − recPicture[ hxCtb+x, vyCtb+y−1 ] −  (8‑1161)  
  recPicture[ hxCtb+x, vyCtb+y+1 ]  )

filtD0[ x ][ y ] = Abs(  ( recPicture[ hxCtb+x, vyCtb+y ] << 1 ) − recPicture[ hxCtb+x−1, vyCtb+y−1 ] −  (8‑1162)  
 recPicture[ hxCtb+x+1, vyCtb+y+1 ]  )

filtD1[ x ][ y ] = Abs(  ( recPicture[ hxCtb+x, vyCtb+y ] << 1 ) − recPicture[ hxCtb+x+1, vyCtb+y−1 ] −  (8‑1163)  
 recPicture[ hxCtb+x−1, vyCtb+y+1 ]  )

* Otherwise, filtH[ x ][ y ], filtV[ x ][ y ], filtD0[ x ][ y ] and filtD1[ x ][ y ] are set equal to 0.

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:

sumH[ x ][ y ] = ΣiΣj filtH[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑1164)

sumV[ x ][ y ] = ΣiΣj filtV[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑1165)

sumD0[ x ][ y ] = ΣiΣj filtD0[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑1166)

sumD1[ x ][ y ] = ΣiΣj filtD1[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = −2..5 (8‑1167)

sumOfHV[ x ][ y ] = sumH[ x ][ y ] + sumV[ x ][ y ] (8‑1168)

1. The variables dir1[ x ][ y ], dir2[ x ][ y ] and dirS[ x ][ y ] with x, y = 0..CtbSizeY − 1 are derived as follows:

* The variables hv1, hv0 and dirHV are derived as follows:
* If sumV[ x >> 2 ][ y >> 2 ] is greater than sumH[ x >> 2 ][ y >> 2 ], the following applies:

hv1 = sumV[ x >> 2 ][ y >> 2 ] (8‑1169)

hv0 = sumH[ x >> 2 ][ y >> 2 ]  8‑1170)

dirHV = 1 (8‑1171)

* Otherwise, the following applies:

hv1 = sumH[ x >> 2 ][ y >> 2 ] (8‑1172)

hv0 = sumV[ x >> 2 ][ y >> 2 ] (8‑1173)

dirHV = 3 (8‑1174)

* The variables d1, d0 and dirD are derived as follows:
* If sumD0[ x >> 2 ][ y >> 2 ] is greater than sumD1[ x >> 2 ][ y >> 2 ], the following applies:

d1 = sumD0[ x >> 2 ][ y >> 2 ] (8‑1175)

d0 = sumD1[ x >> 2 ][ y >> 2 ]  (8‑1176)

dirD = 0 (8‑1177)

* Otherwise, the following applies:

d1 = sumD1[ x >> 2 ][ y >> 2 ] (8‑1178)

d0 = sumD0[ x >> 2 ][ y >> 2 ] (8‑1179)

dirD = 2 (8‑1180)

* The variables hvd1, hvd0, are derived as follows:

hvd1 = ( d1 \* hv0 > hv1 \* d0 )  ?  d1  :  hv1 (8‑1181)

hvd0 = ( d1 \* hv0 > hv1 \* d0 )  ?  d0  :  hv0 (8‑1182)

* The variables dirS[ x ][ y ], dir1[ x ][ y ] and dir2[ x ][ y ] derived as follows:

dir1[ x ][ y ] = ( d1 \* hv0 > hv1 \* d0 )  ?  dirD  :  dirHV (8‑1183)

dir2[ x ][ y ] = ( d1 \* hv0 > hv1 \* d0 )  ?  dirHV  :  dirD (8‑1184)

dirS[ x ][ y ] = ( hvd1 > 2 \* hvd0 )  ?  1  :  ( ( hvd1 \* 2 > 9 \* hvd0 )  ?  2  :  0 ) (8‑1185)

1. The variable avgVar[ x ][ y ] with x, y = 0..CtbSizeY − 1 is derived as follows:

varTab[ ] = { 0, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4 } (8‑1186)

avgVar[ x ][ y ] = varTab[ Clip3( 0, 15, ( sumOfHV[ x >> 2 ][ y >> 2 ] \* 64 ) >> ( 3 + BitDepthY ) ) ] (8‑1187)

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[ dir1[ x ][ y ] \* 2 + ( dir2[ x ][ y ] >> 1 ) ]

filtIdx[ x ][ y ] = avgVar[ x ][ y ]

When dirS[ x ][ y ] is not equal 0, filtIdx[ x ][ y ] is modified as follows:

filtIdx[ x ][ y ] += ( ( ( dir1[ x ][ y ] & 0x1 ) << 1 ) + dirS[ x ][ y ] ) \* 5 (8‑1188)

#### 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.

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 (8‑1189)

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 ) (8‑1190)

vy = Clip3( 0, pic\_height\_in\_luma\_samples / SubHeightC − 1, yCtbC + y ) (8‑1191)

* + The variable sum is derived as follows:

sum = AlfCoeffC[ 0 ] \* ( recPicture[ hx, vy + 2 ] + recPicture[ hx, vy− 2 ] ) +   
 AlfCoeffC[ 1 ] \* ( recPicture[ hx + 1, vy + 1 ] + recPicture[ hx − 1, vy − 1 ] ) +   
 AlfCoeffC[ 2 ] \* ( recPicture[ hx, vy + 1 ] + recPicture[ hx, vy − 1 ] ) + (8‑1192)  
 AlfCoeffC[ 3 ] \* ( recPicture[ hx − 1, vy + 1 ] + recPicture[ hx + 1, vy − 1 ] ) +   
 AlfCoeffC[ 4 ] \* ( recPicture[ hx + 2, vy ] + recPicture[ hx − 2, vy ] ) +   
 AlfCoeffC[ 5 ] \* ( recPicture[ hx + 1, vy ] + recPicture[ hx − 1, vy ] ) +   
 AlfCoeffC[ 6 ] \* recPicture[ hx, vy ]

sum = ( sum + 64 ) >> 7 (8‑1193)

* + 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 ) (8‑1194)

# 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 is 0 to maxVal inclusive, 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 is 0 to maxValinclusive, 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( th )  
if( synVal >= val − b ) {  
 synVal <<= 1  
 synVal += read\_bits( 1 )  
 synVal −= val − b  
}

where the value returned from read\_bits( th ) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

## CABAC parsing process for tile group data

### General

Inputs to this process are a request for a value of a syntax element and values of prior parsed syntax elements.

Output of this process is the value of the syntax element.

The initialization process as specified in clause 9.5.2 is invoked when starting the parsing of the CTU syntax specified in clause 7.3.6.2 and the CTU is the first CTU in a tile. [Ed. (YK): The start of the tile group data is also covered by this sentence as each start of the tile group data is the start of a tile.]

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as specified in subclause 9.5.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 9.5.4.

In case the request for a value of a syntax element is processed for the syntax element pcm\_flag and the decoded value of pcm\_flag is equal to 1, the decoding engine is initialized after the decoding of any pcm\_alignment\_zero\_bit and all pcm\_sample\_luma and pcm\_sample\_chroma data as specified in subclause 9.5.2.3.

The whole CABAC parsing process for a syntax element synEl is illustrated in Figure 9‑1.



Figure 9‑1 – Illustration of CABAC parsing process for a syntax element synEl (informative)

### Initialization process

#### General

Outputs of this process are initialized CABAC internal variables.

The initialization process for a syntax element synEl is illustrated in the flowchart of Figure 9‑2 and described as follows:

* The context variables of the arithmetic decoding engine are initialized by invoking the initialization process for context variables as specified in subclause 9.5.2.2.
* The decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision are initialized by invoking the initialization process for the arithmetic decoding engine as specified in subclause 9.5.2.3.



Figure 9‑2 – Illustration of CABAC initialization process (informative)

#### Initialization process for context variables

Outputs of this process are the initialized CABAC context variables indexed by ctxTable and ctxIdx.

[Ed. (BB): Add tables with current init values]

For each context variable, the two variables pStateIdx0 and pStateIdx1 are initialized as follows:

* Table 9‑6 to Table X-X contain the values of the 8 bit variable initValue used in the initialization of context variables that are assigned to all syntax elements in subclauses 7.3.6.1 through 7.3.6.11, except end\_of\_tile\_one\_bit and pcm\_flag.
* From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

slopeIdx = initValue >> 4  
offsetIdx = initValue & 15 (9‑5)

* The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

m = slopeIdx \* 5 − 45  
n = ( offsetIdx << 3 ) − 16 (9‑6)

* The two values assigned to pStateIdx0 and pStateIdx1 for the initialization are derived from TileGroupQpY, which is derived in Equation 7‑54. Given the variables m and n, the initialization is specified as follows:

preCtxState = Clip3( 0, 127, ( ( m \* Clip3( 0, 51, TileGroupQpY ) ) >> 4 ) + n ) (9‑7)

* The two values assigned to pStateIdx0 and pStateIdx1 for the initialization are derived as follows from initial state mapping table initStateIdxToState[ preCtxState ] specified in Table 9‑4:

pStateIdx0 = initStateIdxToState[preCtxState] >> 4  
pStateIdx1 = initStateIdxToState[preCtxState] (9‑8)

NOTE 1 – The variables pStateIdx0 and pStateIdx1 correspond to probability state indices as further described in subclause 9.5.4.3.

In Table 9‑5, the ctxIdxOffset associated tofor which initialization is needed for each of the three initialization types, specified by the variable initType, are listed. Also listed is the table number that includes the values of initValue needed for the initialization. For P and B tile group types, the derivation of initType depends on the value of the cabac\_init\_flag syntax element. The variable initType is derived as follows:

if( tile\_group\_type = = I )  
 initType = 0  
else if( tile\_group\_type = = P )  
 initType = cabac\_init\_flag ? 2 : 1 (9‑9)  
else  
 initType = cabac\_init\_flag ? 1 : 2

Table 9‑4 – Initial state mapping table initStateIdxToState[ preCtxState ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **preCtxState** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **initStateIdxToState** | 307 | 323 | 340 | 359 | 378 | 398 | 419 | 442 | 466 | 491 | 517 | 544 | 574 | 604 | 637 | 671 |
| **preCtxState** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **initStateIdxToState** | 707 | 745 | 784 | 826 | 871 | 917 | 966 | 1018 | 1073 | 1130 | 1191 | 1254 | 1321 | 1392 | 1467 | 1545 |
| **preCtxState** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **initStateIdxToState** | 1628 | 1715 | 1807 | 1903 | 2005 | 2112 | 2226 | 2345 | 2470 | 2602 | 2741 | 2888 | 3043 | 3206 | 3377 | 3558 |
| **preCtxState** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **initStateIdxToState** | 3748 | 3949 | 4160 | 4383 | 4617 | 4864 | 5124 | 5399 | 5687 | 5992 | 6312 | 6650 | 7006 | 7381 | 7775 | 8192 |
| **preCtxState** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** | **75** | **76** | **77** | **78** | **79** |
| **initStateIdxToState** | 8192 | 8608 | 9002 | 9377 | 9733 | 10071 | 10391 | 10696 | 10984 | 11259 | 11519 | 11766 | 12000 | 12223 | 12434 | 12635 |
| **preCtxState** | **80** | **81** | **82** | **83** | **84** | **85** | **86** | **87** | **88** | **89** | **90** | **91** | **92** | **93** | **94** | **95** |
| **initStateIdxToState** | 12825 | 13006 | 13177 | 13340 | 13495 | 13642 | 13781 | 13913 | 14038 | 14157 | 14271 | 14378 | 14480 | 14576 | 14668 | 14755 |
| **preCtxState** | **96** | **97** | **98** | **99** | **100** | **101** | **102** | **103** | **104** | **105** | **106** | **107** | **108** | **109** | **110** | **111** |
| **initStateIdxToState** | 14838 | 14916 | 14991 | 15062 | 15129 | 15192 | 15253 | 15310 | 15365 | 15417 | 15466 | 15512 | 15557 | 15599 | 15638 | 15676 |
| **preCtxState** | **112** | **113** | **114** | **115** | **116** | **117** | **118** | **119** | **120** | **121** | **122** | **123** | **124** | **125** | **126** | **127** |
| **initStateIdxToState** | 15712 | 15746 | 15779 | 15809 | 15839 | 15866 | 15892 | 15917 | 15941 | 15964 | 15985 | 16005 | 16024 | 16043 | 16060 | 16076 |

| Table 9‑5 – Association of ctxIdxOffset and syntax elements for each initializationType in the initialization process | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Syntax structure** | **Syntax element** | **ctxTable** | **initType** | | |
| **0** | **1** | **2** |
| coding\_tree\_unit( ) | alf\_ctb\_flag[ ][ ][ ] | Table 9‑6 | 0 | 9 | 18 |
| sao( ) | sao\_merge\_left\_flag sao\_merge\_up\_flag | Table 9‑7 | 0 | 1 | 2 |
| sao\_type\_idx\_luma sao\_type\_idx\_chroma | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ctxInc** | **initValue of sao\_merge\_left\_flag and sao\_merge\_up\_flag** | | | **shiftIdx** | | initType = = 0 | initType = = 1 | initType = = 2 | | **0** | 47 | 233 | 199 | 0 |   Table 9‑8 | 0 | 1 | 2 |
| coding\_tree( ) | split\_cu\_flag |  |  |  |  |
| split\_qt\_flag |  |  |  |  |
| mtt\_split\_cu\_vertical\_flag |  |  |  |  |
| mtt\_split\_cu\_binary\_flag |  |  |  |  |
| coding\_unit( ) | cu\_skip\_flag[ ][ ] |  |  | 0 | 3 |
| pred\_mode\_ibc\_flag |  |  |  |  |
| pred\_mode\_flag |  |  |  |  |
| intra\_luma\_ref\_idx[ ][ ] |  |  |  |  |
| intra\_subpartitions\_mode\_flag |  |  |  |  |
| intra\_subpartition\_split\_flag |  |  |  |  |
| intra\_luma\_mpm\_flag[ ][ ] |  |  |  |  |
| intra\_chroma\_pred\_mode[ ][ ] |  |  |  |  |
| merge\_flag[ ][ ] |  |  |  |  |
| inter\_pred\_idc[ x0 ][ y0 ] |  |  |  |  |
| inter\_affine\_flag[ ][ ] |  |  |  |  |
| cu\_affine\_type\_flag[ ][ ] |  |  |  |  |
| ref\_idx\_l0[ ][ ] |  |  |  |  |
| mvp\_l0\_flag[ ][ ] |  |  |  |  |
| ref\_idx\_l1[ ][ ] |  |  |  |  |
| mvp\_l1\_flag[ ][ ] |  |  |  |  |
| avmr\_flag[ ][ ] |  |  |  |  |
| amvr\_precision\_flag[ ][ ] |  |  |  |  |
| gbi\_idx[ ][ ] |  |  |  |  |
| cu\_cbf |  |  |  |  |
| cu\_sbt\_flag |  |  |  |  |
| cu\_sbt\_quad\_flag |  |  |  |  |
| cu\_sbt\_horizontal\_flag |  |  |  |  |
| cu\_sbt\_pos\_flag |  |  |  |  |
| merge\_data( ) | mmvd\_flag[ ][ ] |  |  |  |  |
| mmvd\_merge\_flag[ ][ ] |  |  |  |  |
| mmvd\_distance\_idx[ ][ ] |  |  |  |  |
| ciip\_flag[ ][ ] |  |  |  |  |
| ciip\_luma\_mpm\_flag[ ][ ] |  |  |  |  |
| merge\_subblock\_flag[ ][ ] |  |  |  |  |
| merge\_subblock\_idx[ ][ ] |  |  |  |  |
| merge\_triangle\_flag[ ][ ] |  |  |  |  |
| merge\_triangle\_idx0[ ][ ] |  |  |  |  |
| merge\_triangle\_idx1[ ][ ] |  |  |  |  |
| merge\_idx[ ][ ] |  |  |  |  |
| mvd\_coding( ) | abs\_mvd\_greater0\_flag[ ] |  |  |  |  |
| abs\_mvd\_greater1\_flag[ ] |  |  |  |  |
| transform\_unit( ) | tu\_cbf\_luma[ ][ ][ ] |  |  |  |  |
| tu\_cbf\_cb[ ][ ][ ] |  |  |  |  |
| tu\_cbf\_cr[ ][ ][ ] |  |  |  |  |
| cu\_qp\_delta\_abs |  |  |  |  |
| transform\_skip\_flag[ ][ ] |  |  |  |  |
| tu\_mts\_idx[ ][ ] |  |  |  |  |
| residual\_coding( ) | last\_sig\_coeff\_x\_prefix |  |  |  |  |
| last\_sig\_coeff\_y\_prefix |  |  |  |  |
| coded\_sub\_block\_flag[ ][ ] |  |  |  |  |
| sig\_coeff\_flag[ ][ ] |  |  |  |  |
| par\_level\_flag[ ] |  |  |  |  |
| abs\_level\_gt1\_flag[ ] |  |  |  |  |
| abs\_level\_gt3\_flag[ ] |  |  |  |  |

Table 9‑6 – Specification of initValue and shiftIdx for ctxIdxInc of alf\_ctb\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of alf\_ctb\_flag** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 154 | 139 | 219 | 0 |
| **1** | 186 | 186 | 236 | 0 |
| **2** | 174 | 203 | 238 | 4 |
| **3** | 183 | 183 | 232 | 0 |
| **4** | 233 | 247 | 249 | 0 |
| **5** | 250 | 249 | 235 | 1 |
| **6** | 168 | 183 | 246 | 0 |
| **7** | 248 | 232 | 234 | 0 |
| **8** | 250 | 249 | 251 | 1 |

Table 9‑7 – Specification of initValue and shiftIdx for ctxInc of sao\_merge\_left\_flag and sao\_merge\_up\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of sao\_merge\_left\_flag and sao\_merge\_up\_flag** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 47 | 233 | 199 | 0 |

Table 9‑8 – Specification of initValue and shiftIdx for ctxInc of sao\_type\_idx\_luma and sao\_type\_idx\_chroma

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of sao\_type\_idx\_luma and sao\_type\_idx\_chroma** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 47 | 95 | 95 | 0 |

#### Initialization process for the arithmetic decoding engine

Outputs of this process are the initialized decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision.

The status of the arithmetic decoding engine is represented by the variables ivlCurrRange and ivlOffset. In the initialization procedure of the arithmetic decoding process, ivlCurrRange is set equal to 510 and ivlOffset is set equal to the value returned from read\_bits( 9 ) interpreted as a 9 bit binary representation of an unsigned integer with the most significant bit written first.

The bitstream shall not contain data that result in a value of ivlOffset being equal to 510 or 511.

NOTE – The description of the arithmetic decoding engine in this Specification utilizes 16 bit register precision. However, a minimum register precision of 9 bits is required for storing the values of the variables ivlCurrRange and ivlOffset after invocation of the arithmetic decoding process (DecodeBin) as specified in subclause 9.5.4.3. The arithmetic decoding process for a binary decision (DecodeDecision) as specified in subclause 9.5.4.3.2 and the decoding process for a binary decision before termination (DecodeTerminate) as specified in subclause 9.5.4.3.5 require a minimum register precision of 9 bits for the variables ivlCurrRange and ivlOffset. The bypass decoding process for binary decisions (DecodeBypass) as specified in subclause 9.5.4.3.4 requires a minimum register precision of 10 bits for the variable ivlOffset and a minimum register precision of 9 bits for the variable ivlCurrRange.

### 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 9‑9 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the truncated binary (TB) 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 9.5.3.7, respectively.

| Table 9‑9 – Syntax elements and associated binarizations | | | |
| --- | --- | --- | --- |
| **Syntax structure** | **Syntax element** | **Binarization** | |
| **Process** | **Input parameters** |
| tile\_group\_data( ) | end\_of\_tile\_one\_bit | FL | cMax = 1 |
| coding\_tree\_unit( ) | alf\_ctb\_flag[ ][ ][ ] | FL | cMax = 1 |
| sao( ) | sao\_merge\_left\_flag | FL | cMax = 1 |
| sao\_merge\_up\_flag | FL | cMax = 1 |
| sao\_type\_idx\_luma | TR | cMax = 2, cRiceParam = 0 |
| sao\_type\_idx\_chroma | TR | cMax = 2, cRiceParam = 0 |
| sao\_offset\_abs[ ][ ][ ][ ] | TR | cMax = ( 1  <<  ( Min( bitDepth, 10 ) − 5 ) ) − 1, cRiceParam = 0 |
| sao\_offset\_sign[ ][ ][ ][ ] | FL | cMax = 1 |
| sao\_band\_position[ ][ ][ ] | FL | cMax = 31 |
| sao\_eo\_class\_luma | FL | cMax = 3 |
| sao\_eo\_class\_chroma | FL | cMax = 3 |
| coding\_tree( ) | split\_cu\_flag | FL | cMax = 1 |
| split\_qt\_flag | FL | cMax = 1 |
| mtt\_split\_cu\_vertical\_flag | FL | cMax = 1 |
| mtt\_split\_cu\_binary\_flag | FL | cMax = 1 |
| coding\_unit( ) | cu\_skip\_flag[ ][ ] | FL | cMax = 1 |
| pred\_mode\_ibc\_flag | FL | cMax = 1 |
| pred\_mode\_flag | FL | cMax = 1 |
| pcm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_luma\_ref\_idx[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| intra\_subpartitions\_mode\_flag | FL | cMax = 1 |
| intra\_subpartition\_split\_flag | FL | cMax = 1 |
| intra\_luma\_mpm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_luma\_mpm\_idx[ ][ ] | TR | cMax = 5, cRiceParam = 0 |
| intra\_luma\_mpm\_remainder[ ][ ] | TB | cMax = 60 |
| intra\_chroma\_pred\_mode[ ][ ] | 9.5.3.8 | - |
| merge\_flag[ ][ ] | FL | cMax = 1 |
| inter\_pred\_idc[ x0 ][ y0 ] | 9.5.3.9 | cbWidth, cbHeight |
| inter\_affine\_flag[ ][ ] | FL | cMax = 1 |
| cu\_affine\_type\_flag[ ][ ] | FL | cMax = 1 |
| ref\_idx\_l0[ ][ ] | TR | cMax = NumRefIdxActive[ 0 ] − 1, cRiceParam = 0 |
| mvp\_l0\_flag[ ][ ] | FL | cMax = 1 |
| ref\_idx\_l1[ ][ ] | TR | cMax = NumRefIdxActive[ 1 ] − 1, cRiceParam = 0 |
| mvp\_l1\_flag[ ][ ] | FL | cMax = 1 |
| avmr\_flag[ ][ ] | FL | cMax = 1 |
| amvr\_precision\_flag[ ][ ] | FL | cMax = 1 |
| gbi\_idx[ ][ ] | TR | cMax = NoBackwardPredFlag ? 4: 2 |
| cu\_cbf | FL | cMax = 1 |
| cu\_sbt\_flag | FL | cMax = 1 |
| cu\_sbt\_quad\_flag | FL | cMax = 1 |
| cu\_sbt\_horizontal\_flag | FL | cMax = 1 |
| cu\_sbt\_pos\_flag | FL | cMax = 1 |
| merge\_data( ) | mmvd\_flag[ ][ ] | FL | cMax = 1 |
| mmvd\_merge\_flag[ ][ ] | FL | cMax = 1 |
| mmvd\_distance\_idx[ ][ ] | TR | cMax = 7, cRiceParam = 0 |
| mmvd\_direction\_idx[ ][ ] | FL | cMax = 3 |
| ciip\_flag[ ][ ] | FL | cMax = 1 |
| ciip\_luma\_mpm\_flag[ ][ ] | FL | cMax = 1 |
| ciip\_luma\_mpm\_idx[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| merge\_subblock\_flag[ ][ ] | FL | cMax = 1 |
| merge\_subblock\_idx[ ][ ] | TR | cMax = MaxNumSubblockMergeCand − 1, cRiceParam = 0 |
| merge\_triangle\_flag[ ][ ] | FL | cMax = 1 |
| merge\_triangle\_idx[ ][ ] | FL | cMax = 1 |
| merge\_triangle\_idx0[ ][ ] | TR | cMax = 4, cRiceParam = 0 |
| merge\_triangle\_idx1[ ][ ] | TR | cMax = 3, cRiceParam = 0 |
| merge\_idx[ ][ ] | TR | cMax = MaxNumMergeCand − 1, cRiceParam = 0 |
| 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\_qp\_delta\_abs | 9.5.3.10 | - |
| cu\_qp\_delta\_sign\_flag | FL | cMax = 1 |
| transform\_skip\_flag[ ][ ] | FL | cMax = 1 |
| tu\_mts\_idx[ ][ ] | TR | cMax = 4, cRiceParam = 0 |
| residual\_coding( ) | last\_sig\_coeff\_x\_prefix | TR | cMax = ( log2TbWidth << 1 ) − 1, cRiceParam = 0 |
| last\_sig\_coeff\_y\_prefix | TR | cMax = ( log2TbHeight << 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 |
| abs\_level\_gt1\_flag[ ] | FL | cMax = 1 |
| abs\_level\_gt3\_flag[ ] | FL | cMax = 1 |
| abs\_remainder[ ] | 9.5.3.11 | cIdx, current sub-block index i, x0, y0 |
| dec\_abs\_level[ ] | 9.5.3.12 | cIdx, x0, y0, xC, yC, log2TbWidth, log2TbHeight |
| coeff\_sign\_flag[ ] | FL | cMax = 1 |

#### Rice parameter derivation process for dec\_abs\_level[ ]

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 array AbsLevel[ x ][ y ] 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 ]   
 if( xC < (1 << log2TbWidth) − 2 )  
 locSumAbs += AbsLevel[ xC + 2 ][ yC ]   
 if( yC < (1 << log2TbHeight) − 1 )  
 locSumAbs += AbsLevel[ xC + 1 ][ yC + 1 ] (9‑10)  
}  
if( yC < (1 << log2TbHeight) − 1 ) {  
 locSumAbs += AbsLevel[ xC ][ yC + 1 ]   
 if( yC < (1 << log2TbHeight) − 2 )  
 locSumAbs += AbsLevelPass1 [ xC ][ yC + 2 ]   
}   
if( locSumAbs > 31 )  
 locSumAbs = 31

The variable s is set equal to Max( 0, QState – 1 ).

Given the variables locSumAbs and s, the Rice parameter cRiceParam and the variable ZeroPos[ n ]0 are derived as specified in Table 9‑10.

Table 9‑10 – Specification of cRiceParam and ZeroPos[ n ] based on locSumAbs and s

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **s** | **locSumAbs** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
|  | cRiceParam | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| **0** | ZeroPos[ n ] | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 |
| **1** | ZeroPos[ n ] | 1 | 1 | 1 | 1 | 2 | 3 | 4 | 4 | 4 | 6 | 6 | 6 | 8 | 8 | 8 | 8 |
| **2** | ZeroPos[ n ] | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 6 | 6 | 6 | 8 | 8 | 8 | 8 |
|  | **locSumAbs** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
|  | cRiceParam | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| **0** | ZeroPos[ n ] | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 16 | 16 | 16 | 16 |
| **1** | ZeroPos[ n ] | 4 | 4 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 16 | 16 | 16 | 16 | 16 | 16 |
| **2** | ZeroPos[ n ] | 8 | 8 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |

#### 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 (9‑11)

* 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 9‑11 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 9‑11 – 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 ) (9‑12)

* The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified in clause 9.5.3.7 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.

#### Truncated Binary (TB) binarization process

Input to this process is a request for a TB binarization for a syntax element with value synVal and cMax. Output of this process is the TB binarization of the syntax element.The bin string of the TB binarization process of a syntax element synVal is specified as follows:

n = cMax + 1  
k = Floor( Log2( n ) ) (9‑13)  
u = ( 1  <<  ( k + 1) ) − n

* If synVal is less than u, the TB bin string is derived by invoking the FL binarization process specified in clause 9.5.3.7 for synVal with a cMax value equal to ( 1  <<  k ) − 1.
* Otherwise (synVal is greater than or equal to u), the TB bin string is derived by invoking the FL binarization process specified in clause 9.5.3.7 for ( synVal + u ) with a cMax value equal to ( 1  <<  ( k + 1) ) − 1.

#### 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 ) (9‑14)  
 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.

#### Limited k-th order Exp-Golomb binarization process

Inputs to this process is a request for a limited k-th order Exp-Golomb (EGk) binarization and the Rice parameter riceParam.

Output of this process is the limited EGk binarization associating each value symbolVal with a corresponding bin string.

The variables log2TransformRange and maxPreExtLen are derived as follows:

log2TransformRange = 15 (9‑15)

maxPreExtLen = 26 − log2TransformRange (9‑16)

The bin string of the limited 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:

codeValue = symbolVal >> riceParam  
preExtLen = 0  
while( ( preExtLen < maxPreExtLen ) && ( codeValue > ( ( 2 << preExtLen ) − 2 ) ) ) {  
 preExtLen++  
 put( 1 )  
}  
if( preExtLen = = maxPreExtLen ) (9‑17)  
 escapeLength = log2TransformRange  
else {  
 escapeLength = preExtLen + riceParam  
 put( 0 )   
}  
symbolVal = symbolVal − ( ( ( 1 << preExtLen ) − 1 ) << riceParam )  
while( ( escapeLength− − ) > 0 )  
 put( ( symbolVal >> escapeLength ) & 1 )

#### 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 9‑12 and Table 9‑13.

Table 9‑12 – 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 9‑13 – Binarization for intra\_chroma\_pred\_mode when sps\_cclm\_enabled\_flag is equal to 1

|  |  |
| --- | --- |
| **Value of intra\_chroma\_pred\_mode** | **Bin string** |
| 7 | 0 |
| 4 | 10 |
| 5 | 1110 |
| 6 | 1111 |
| 0 | 11000 |
| 1 | 11001 |
| 2 | 11010 |
| 3 | 11011 |

#### Binarization process for inter\_pred\_idc

Input to this process is a request for a binarization for the syntax element inter\_pred\_idc, the current luma coding block width cbWidth and the current luma coding block height cbHeight.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element inter\_pred\_idc is specified in Table 9‑14.

Table 9‑14 – Binarization for inter\_pred\_idc

|  |  |  |  |
| --- | --- | --- | --- |
| **Value of inter\_pred\_idc** | **Name of inter\_pred\_idc** | **Bin string** | |
| ( cbWidth + cbHeight )  !=  8 | ( cbWidth + cbHeight )  = =  8 |
| 0 | PRED\_L0 | 00 | 0 |
| 1 | PRED\_L1 | 01 | 1 |
| 2 | PRED\_BI | 1 | - |

#### Binarization process for cu\_qp\_delta\_abs

Input to this process is a request for a binarization for the syntax element cu\_qp\_delta\_abs.

Output of this process is the binarization of the syntax element.

The binarization of the syntax element cu\_qp\_delta\_abs 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 cu\_qp\_delta\_abs, prefixVal, is derived as follows:

prefixVal = Min( cu\_qp\_delta\_abs, 5 ) (9‑18)

* The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with cMax = 5 and cRiceParam = 0.

When prefixVal is greater than 4, the suffix bin string is present and it is derived as follows:

* The suffix value of cu\_qp\_delta\_abs, suffixVal, is derived as follows:

suffixVal = cu\_qp\_delta\_abs − 5 (9‑19)

* The suffix bin string is specified by invoking the k-th order EGk binarization process as specified in clause 9.5.3.5 for suffixVal with the Exp-Golomb order k set equal to 0.

#### 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 current sub-block index i, and 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).

Output of this process is the binarization of the syntax element.

The variables lastAbsRemainder and lastRiceParam are derived as follows:

* If this process is invoked for the first time for the current sub-block index i, lastAbsRemainder and lastRiceParam are both set equal to 0.
* Otherwise (this process is not invoked for the first time for the current sub-block index i), lastAbsRemainder and lastRiceParam are set equal to the values of abs\_remainder[ n ] and cRiceParam, respectively, that have been derived during the last invocation of the binarization process for the syntax element abs\_remainder[ n ] as specified in this clause.

The variable cRiceParam is derived from lastAbsRemainder and lastRiceParam as follows:

cRiceParam = Min( lastRiceParam + ( ( lastAbsRemainder > ( 3 \* ( 1 << lastRiceParam ) ) ) ? 1 : 0 ), 3 ) (9‑20)

The variable cMax is derived from cRiceParam as:

cMax = 6  <<  cRiceParam (9‑21)

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 ] ) (9‑22)

* 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 6 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 (9‑23)

* The suffix bin string is specified by invoking the limited k-th order EGk binarization process as specified in clause 9.5.3.6 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1 and cRiceParam as input.

#### Binarization process for dec\_abs\_level[ ]

Input to this process is a request for a binarization of the syntax element dec\_abs\_level[ n ], the colour component cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current 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 for dec\_abs\_level[] 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 = 6  <<  cRiceParam (9‑24)

The binarization of dec\_abs\_level[ 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 dec\_abs\_level[ n ], prefixVal, is derived as follows:

prefixVal = Min( cMax, dec\_abs\_level[ n ] ) (9‑25)

* 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 6 with all bits equal to 1, the suffix bin string is present and it is derived as follows:

* The suffix value of dec\_abs\_level[ n ], suffixVal, is derived as follows:

suffixVal = dec\_abs\_level[ n ] − cMax (9‑26)

* The suffix bin string is specified by invoking the limited k-th order EGk binarization process as specified in clause 9.5.3.6 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1 and cRiceParam as input.

### Decoding process flow

#### General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in clause 9.5.3.

Output of this process is the value of the syntax element.

This process specifies how each bin of a bin string is parsed for each syntax element. After parsing each bin, the resulting bin string is compared to all bin strings of the binarization of the syntax element and the following applies:

– If the bin string is equal to one of the bin strings, the corresponding value of the syntax element is the output.

– Otherwise (the bin string is not equal to one of the bin strings), the next bit is parsed.

While parsing each bin, the variable binIdx is incremented by 1 starting with binIdx being set equal to 0 for the first bin.

The parsing of each bin is specified by the following two ordered steps:

1. The derivation process for ctxTable, ctxIdx, and bypassFlag as specified in clause 9.5.4.2 is invoked with binIdx as input and ctxTable, ctxIdx and bypassFlag as outputs.

2. The arithmetic decoding process as specified in clause 9.5.4.3 is invoked with ctxTable, ctxIdx and bypassFlag as inputs and the value of the bin as output.

#### Derivation process for ctxTable, ctxIdx and bypassFlag

##### General

Input to this process is the position of the current bin within the bin string, binIdx.

Outputs of this process are ctxTable, ctxIdx and bypassFlag.

The values of ctxTable, ctxIdx and bypassFlag are derived as follows based on the entries for binIdx of the corresponding syntax element in Table 9‑15:

* If the entry in Table 9‑15 is not equal to "bypass", "terminate" or "na", the values of binIdx are decoded by invoking the DecodeDecision process as specified in clause 9.5.4.3.2 and the following applies:
* ctxTable is specified in Table 9‑5
* The variable ctxInc is specified by the corresponding entry in Table 9‑15 and when more than one value is listed in Table 9‑15 for a binIdx, the assignment process for ctxInc for that binIdx is further specified in the clauses given in parenthesis.
* The variable ctxIdxOffset is specified in Table 9‑5 depending on the current value of initType.
* ctxIdx is set equal to the sum of ctxInc and ctxIdxOffset.
* bypassFlag is set equal to 0.
* Otherwise, if the entry in Table 9‑15 is equal to "bypass", the values of binIdx are decoded by invoking the DecodeBypass process as specified in clause 9.5.4.3.4 and the following applies:
* ctxTable is set equal to 0.
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 1.a
* Otherwise, if the entry in Table 9‑15 is equal to "terminate", the values of binIdx are decoded by invoking the DecodeTerminate process as specified in clause 9.5.4.3.5 and the following applies:
* ctxTable is set equal to 0.
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 0.
* Otherwise (the entry in Table 9‑15 is equal to "na"), the values of binIdx do not occur for the corresponding syntax element.

| Table 9‑15 – Assignment of ctxInc to syntax elements with context coded bins | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Syntax element** | **binIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **>= 5** |
| end\_of\_tile\_one\_bit | terminate | na | na | na | na | na |
| alf\_ctb\_flag[ ][ ][ ] | 0..8 (clause 9.5.4.2.2) | na | na | na | na | na |
| sao\_merge\_left\_flag | 0 | na | na | na | na | na |
| sao\_merge\_up\_flag | 0 | na | na | na | na | na |
| sao\_type\_idx\_luma | 0 | bypass | na | na | na | na |
| sao\_type\_idx\_chroma | 0 | bypass | na | na | na | na |
| sao\_offset\_abs[ ][ ][ ][ ] | bypass | bypass | bypass | bypass | bypass | na |
| sao\_offset\_sign[ ][ ][ ][ ] | bypass | na | na | na | na | na |
| sao\_band\_position[ ][ ][ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| sao\_eo\_class\_luma | bypass | bypass | na | na | na | na |
| sao\_eo\_class\_chroma | bypass | bypass | na | na | na | na |
| split\_cu\_flag | 0..8 (clause 9.5.4.2.2) | na | na | na | na | na |
| split\_qt\_flag | 0..5 (clause 9.5.4.2.2) | na | na | na | na | na |
| mtt\_split\_cu\_vertical\_flag | 0..4 (clause 9.5.4.2.3) | na | na | na | na | na |
| mtt\_split\_cu\_binary\_flag | ( 2 \* mtt\_split\_cu\_vertical\_flag ) + ( mttDepth < = 1 ? 1 : 0 ) | na | na | na | na | na |
| cu\_skip\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| pred\_mode\_flag | 0,1 (clause 9.5.4.2.2) | na | na | na | na | na |
| pred\_mode\_ibc\_flag | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| pcm\_flag[ ][ ] | terminate | na | na | na | na | na |
| intra\_luma\_ref\_idx[ ][ ] | 0 | 1 | na | na | na | na |
| intra\_subpartitions\_mode\_flag | 0 | na | na | na | na | na |
| intra\_subpartition\_split\_flag | 0 | na | na | na | na | na |
| intra\_luma\_mpm\_flag[ ][ ] | 0 | na | na | na | na | na |
| intra\_luma\_mpm\_idx[ ][ ] | bypass | bypass | bypass | bypass | bypass | na |
| intra\_luma\_mpm\_remainder[ ][ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| intra\_chroma\_pred\_mode[ ][ ] sps\_cclm\_enabled\_flag = = 0 | 0 | bypass | bypass | na | na | na |
| intra\_chroma\_pred\_mode[ ][ ]  sps\_cclm\_enabled\_flag = = 1 &&  bin at binIdx equal to 2 = = 0 | 0 | 1 | 2 | bypass | bypass | na |
| intra\_chroma\_pred\_mode[ ][ ]  sps\_cclm\_enabled\_flag = = 1 &&  bin at binIdx equal to 2 = = 1 | 0 | 1 | 2 | 2 | na | na |
| merge\_subblock\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| merge\_subblock\_idx[ ][ ] | 0 | bypass | bypass | bypass | bypass | na |
| merge\_flag[ ][ ] | 0 | na | na | na | na | na |
| mmvd\_flag[ ][ ] | 0 | na | na | na | na | na |
| mmvd\_merge\_flag[ ][ ] | 0 | na | na | na | na | na |
| mmvd\_distance\_idx[ ][ ] | 0 | bypass | bypass | bypass | bypass | bypass |
| mmvd\_direction\_idx[ ][ ] | bypass | bypass | na | na | na | na |
| merge\_triangle\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| merge\_triangle\_split\_dir[ ][ ] | bypass | na | na | na | na | na |
| merge\_triangle\_idx0[ ][ ] | 0 | bypass | bypass | bypass | bypass | na |
| merge\_triangle\_idx1[ ][ ] | 0 | bypass | bypass | bypass | na | na |
| merge\_idx[ ][ ] | 0 | bypass | bypass | bypass | bypass | na |
| ciip\_flag[ ][ ] | 0 | na | na | na | na | na |
| ciip\_luma\_mpm\_flag[ ][ ] | 0 | na | na | na | na | na |
| ciip\_luma\_mpm\_idx[ ][ ] | bypass | bypass | na | na | na | na |
| inter\_pred\_idc[ x0 ][ y0 ] | ( cbWidth + cbHeight ) != 8 ? 7 − ( ( 1 +  Log2( cbWidth ) + Log2( cbHeight ) ) >> 1 )   : 4 | 4 | na | na | na | na |
| inter\_affine\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| cu\_affine\_type\_flag[ ][ ] | 0 | na | na | na | na | na |
| ref\_idx\_l0[ ][ ] | 0 | 1 | bypass | bypass | bypass | bypass |
| ref\_idx\_l1[ ][ ] | 0 | 1 | bypass | bypass | bypass | bypass |
| mvp\_l0\_flag[ ][ ] | 0 | na | na | na | na | na |
| mvp\_l1\_flag[ ][ ] | 0 | na | na | na | na | na |
| amvr\_flag[ ][ ] | inter\_affine\_flag[ ][ ] ? 3 :  ( 0,1,2 (clause 9.5.4.2.2) ) | na | na | na | na | na |
| amvr\_precision\_flag[ ][ ] | 0 | na | na | na | na | na |
| gbi\_idx[ ][ ] NoBackwardPredFlag = = 0 | 0 | 1 | na | na | na | na |
| gbi\_idx[ ][ ]  NoBackwardPredFlag = = 1 | 0 | 1 | 2 | 3 | na | na |
| cu\_cbf | 0 | na | na | na | na | na |
| cu\_sbt\_flag | ( cbWidth \*  cbHeight < 256 ) ? 1 : 0 | na | na | na | na | na |
| cu\_sbt\_quad\_flag | 0 | na | na | na | na | na |
| cu\_sbt\_horizontal\_flag | ( cbWidth = = cbHeight ) ? 0 : ( cbWidth < cbHeight ) ? 1 : 2 | na | na | na | na | na |
| cu\_sbt\_pos\_flag | 0 | na | na | na | na | na |
| abs\_mvd\_greater0\_flag[ ] | 0 | na | na | na | na | na |
| abs\_mvd\_greater1\_flag[ ] | 0 | na | na | na | na | na |
| abs\_mvd\_minus2[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| mvd\_sign\_flag[ ] | bypass | na | na | na | na | na |
| tu\_cbf\_luma[ ][ ][ ] | 0,1,2,3 (clause 9.5.4.2.5) | na | na | na | na | na |
| tu\_cbf\_cb[ ][ ][ ] | trDepth = = 0 ? 0 : 1 | na | na | na | na | na |
| tu\_cbf\_cr[ ][ ][ ] | tu\_cbf\_cb[ ][ ][ ] | na | na | na | na | na |
| cu\_qp\_delta\_abs | 0 | 1 | 1 | 1 | 1 | bypass |
| cu\_qp\_delta\_sign\_flag | bypass | na | na | na | na | na |
| transform\_skip\_flag[ ][ ] | 0 | na | na | na | na | na |
| tu\_mts\_idx[ ][ ] | cqtDepth | 6 | 7 | 8 | na | na |
| last\_sig\_coeff\_x\_prefix | 0..23 (clause 9.5.4.2.4) | | | | | |
| last\_sig\_coeff\_y\_prefix | 0..23 (clause 9.5.4.2.4) | | | | | |
| last\_sig\_coeff\_x\_suffix | bypass | bypass | bypass | bypass | bypass | bypass |
| last\_sig\_coeff\_y\_suffix | bypass | bypass | bypass | bypass | bypass | bypass |
| coded\_sub\_block\_flag[ ][ ] | 0..3 (clause 9.5.4.2.6) | na | na | na | na | na |
| sig\_coeff\_flag[ ][ ] | 0..89 (clause 9.5.4.2.8) | na | na | na | na | na |
| par\_level\_flag[ ] | 0..32 (clause 9.5.4.2.9) | na | na | na | na | na |
| abs\_level\_gt1\_flag[ ] | 0..32 (clause 9.5.4.2.9) | na | na | na | na | na |
| abs\_level\_gt3\_flag[ ] | 0..32 (clause 9.5.4.2.9) | na | na | na | na | na |
| abs\_remainder[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| dec\_abs\_level[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| coeff\_sign\_flag[ ] | bypass | na | na | na | na | na |

[Ed. (BB): In VTM, ctxInc of tu\_cbf\_cb is set to trDepth but trDepth can only takes values 0 and 1 because MaxTbSizeY is always equal to 64 and the maximum CTU size is 128. Thus, the depth resulting from implicit split cannot exceed 1.]

##### Derivation process of ctxInc using left and above syntax elements

Input to this process is the luma location ( x0, y0 ) specifying the top-left luma sample of the current luma block relative to the top-left sample of the current picture, the colour component cIdx, the current coding quadtree depth cqDepth, the width and the height of the current coding block in luma samples cbWidth and cbHeight, and the variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer, allowSplitTtHor, and allowSplitQt as derived in the coding tree semantics in clause 7.4.7.4.

Output of this process is ctxInc.

The location ( xNbL, yNbL ) is set equal to ( x0 − 1, y0 ) and the variable availableL, specifying the availability of the block located directly to the left of the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbL, yNbL ) as inputs, and the output is assigned to availableL.

The location ( xNbA, yNbA ) is set equal to ( x0, y0 − 1 ) and the variable availableA specifying the availability of the coding block located directly above the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbA, yNbA ) as inputs, and the output is assigned to availableA.

The assignment of ctxInc is specified as follows with condL and condA specified in For the syntax element pred\_mode\_flag[ x0 ][ y0 ]:

ctxInc = ( condL  &&  availableL ) | | ( condA  &&  availableA ) (9‑28)

Table 9‑16:

* For the syntax elements alf\_ctb\_flag[ x0 ][ y0 ][ cIdx ], split\_qt\_flag, split\_cu\_flag, cu\_skip\_flag[ x0 ][ y0 ], pred\_mode\_ibc\_flag[ x0 ][ y0 ], amvr\_flag[ x0 ][ y0 ], inter\_affine\_flag[ x0 ][ y0 ], merge\_triangle\_flag[ x0 ][ y0 ] and merge\_subblock\_flag[ x0 ][ y0 ]:

ctxInc = ( condL  &&  availableL ) + ( condA  &&  availableA ) + ctxSetIdx \* 3 (9‑27)

* For the syntax element pred\_mode\_flag[ x0 ][ y0 ]:

ctxInc = ( condL  &&  availableL ) | | ( condA  &&  availableA ) (9‑28)

Table 9‑16 – Specification of ctxInc using left and above syntax elements

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax element** | **condL** | **condA** | **ctxSetIdx** |
| alf\_ctb\_flag[ x0 ][ y0 ][ cIdx ] | alf\_ctb\_flag[ xNbL ][ yNbL ][ cIdx ] | alf\_ctb\_flag[ xNbA ][ yNbA ][cIdx ] | cIdx |
| split\_qt\_flag | cqtDepth[ xNbL ][ yNbL ] > cqtDepth | cqtDepth[ xNbA ][ yNbA ] > cqtDepth | ( cqtDepth < 2) ? 0 : 1 |
| split\_cu\_flag | CbHeight[ xNbL ][ yNbL ] < cbHeight | CbWidth[ xNbA ][ yNbA ] < cbWidth | ( allowSplitBtVer +   allowSplitBtHor +   allowSplitTtVer +   allowSplitTtHor +   2 \* allowSplitQt − 1 ) / 3 |
| cu\_skip\_flag[ x0 ][ y0 ] | cu\_skip\_flag[ xNbL ][ yNbL ] | cu\_skip\_flag[ xNbA ][ yNbA ] | 0 |
| pred\_mode\_flag[ x0 ][ y0 ] | CuPredMode[ xNbL ][ yNbL ] = = MODE\_INTRA | CuPredMode[ xNbA ][ yNbA ] = = MODE\_INTRA | 0 |
| pred\_mode\_ibc\_flag[ x0 ][ y0 ] | CuPredMode[ xNbL ][ yNbL ] = = MODE\_IBC | CuPredMode[ xNbA ][ yNbA ] = = MODE\_IBC | 0 |
| amvr\_flag[ x0 ][ y0 ] | amvr\_flag[ xNbL ][ yNbL ] | amvr\_flag[ xNbA ][ yNbA ] | 0 |
| merge\_subblock\_flag[ x0 ][ y0 ] | merge\_subblock\_flag[ xNbL ][ yNbL ] | inter\_affine\_flag[ xNbL ][ yNbL ] | merge\_subblock\_flag[ xNbA ][ yNbA ] | inter\_affine\_flag[ xNbA ][ yNbA ] | 0 |
| merge\_triangle\_flag[ x0 ][ y0 ] | merge\_triangle\_flag [ xNbL ][ yNbL ] | merge\_triangle\_flag [ xNbA ][ yNbA ] | 0 |
| inter\_affine\_flag [ x0 ][ y0 ] | merge\_subblock\_flag[ xNbL ][ yNbL ] | inter\_affine\_flag[ xNbL ][ yNbL ] | merge\_subblock\_flag[ xNbA ][ yNbA ] | inter\_affine\_flag[ xNbA ][ yNbA ] | 0 |

##### Derivation process of ctxIncfor the syntax element mtt\_split\_cu\_vertical\_flag

Input to this process is the luma location ( x0, y0 ) specifying the top-left luma sample of the current luma block relative to the top-left sample of the current picture, the width and the height of the current coding block in luma samples cbWidth and cbHeight, and the variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer, allowSplitTtHor, and allowSplitQt as derived in the coding tree semantics in clause 7.4.7.4.

Output of this process is ctxInc.

The location ( xNbL, yNbL ) is set equal to ( x0 − 1, y0 ) and the variable availableL, specifying the availability of the block located directly to the left of the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbL, yNbL ) as inputs, and the output is assigned to availableL.

The location ( xNbA, yNbA ) is set equal to ( x0, y0 − 1 ) and the variable availableA specifying the availability of the coding block located directly above the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbA, yNbA ) as inputs, and the output is assigned to availableA.

The assignment of ctxInc is specified as follows:

* If allowSplitBtVer + allowSplitBtHor is greater than allowSplitTtVer + allowSplitTtHor, ctxInc is set equal to 4.
* Otherwise, if allowSplitBtVer + allowSplitBtHor is less than allowSplitTtVer + allowSplitTtHor, ctxInc is set equal to 4.
* Otherwise, the following applies:
* The variables dA and dL are derived as follows

dA = cbWidth / ( availableA  ?  CbWidth[ xNbA ][ yNbA ]  :  1 ) (9‑29)

dL = cbHeight / ( availableL  ?  CbHeight[ xNbL ][ yNbL ]  :  1 ) (9‑30)

* If any of the following conditions is true, ctxInc is set equal to 0:
* dA is equal to dL,
* availableA is equal to FALSE,
* availableL is equal to FALSE.
* Otherwise, if dA is less then dL, ctxInc is set equal to 1.
* Otherwise, ctxInc is set equal to 0.

##### Derivation process of ctxInc for the syntax elements last\_sig\_coeff\_x\_prefix and last\_sig\_coeff\_y\_prefix

Inputs to this process are the variable binIdx, the colour component index cIdx, the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable log2TbSize is derived as follows:

* If the syntax element to be parsed is last\_sig\_coeff\_x\_prefix, log2TbSize is set equal to log2TbWidth.
* Otherwise (the syntax element to be parsed is last\_sig\_coeff\_y\_prefix), log2TbSize is set equal to log2TbHeight.

The variables ctxOffset and ctxShift are derived as follows:

* If cIdx is equal to 0, ctxOffset is set equal to 3 \* ( log2TbSize − 2 ) + ( ( log2TbSize − 1 )  >>  2 ) and ctxShift is set equal to ( log2TbSize + 1 )  >>  2.
* Otherwise (cIdx is greater than 0), ctxOffset is set equal to 21 and ctxShift is set equal to Clip3( 0, 2, 2log2TbSize >> 3 ).

The variable ctxInc is derived as follows:

ctxInc = ( binIdx >> ctxShift ) + ctxOffset (9‑31)

##### Derivation process of ctxInc for the syntax element tu\_cbf\_luma

Inputs to this process are the variable binIdx, the colour component index cIdx, the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight and the current transform tree depth trDepth.

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

* If IntraSubpartitionSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0, the following applies:

ctxInc = trDepth = = 0 ? 1 : 0 (9‑32)

* Otherwise ( IntraSubpartitionSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 ), the following applies:
* The variable prevTuCbfY is derived as follows:
* If the current transform unit is the first one to be parsed in a coding unit, prevTuCbfY is set equal to 0.
* Otherwise, prevTuCbfY is set equal to the value of tu\_cbf\_luma of the previous luma transform unit in the current coding unit.
* The variable ctxInc is derived as follows:

ctxInc = 2 + prevTuCbfY (9‑33)

##### Derivation process of ctxInc for the syntax element coded\_sub\_block\_flag

Inputs to this process are the colour component index cIdx, the current sub-block scan location ( xS, yS ), the previously decoded bins of the syntax element coded\_sub\_block\_flag and the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable csbfCtx is derived using the current location ( xS, yS ), two previously decoded bins of the syntax element coded\_sub\_block\_flag in scan order, log2TbWidth and log2TbHeight, as follows:

* The variables log2SbWidth and log2SbHeight are dervied as follows:

log2SbWidth = ( Min( log2TbWidth, log2TbHeight ) < 2 ? 1 : 2 ) (9‑34)

log2SbHeight = log2SbWidth (9‑35)

* The variables log2SbWidth and log2SbHeight are modifed as follows:
* If log2TbWidth is less than 2 and cIdx is equal to 0, the following applies

log2SbWidth = log2TbWidth (9‑36)

log2SbHeight = 4 − log2SbWidth (9‑37)

* Otherwise, if log2TbHeight is less than 2 and cIdx is equal to 0, the following applies

log2SbHeight = log2TbHeight (9‑38)

log2SbWidth = 4 − log2SbHeight (9‑39)

* csbfCtx is initialized with 0 as follows:

csbfCtx = 0 (9‑40)

* When xS is less than ( 1  <<  ( log2TbWidth − log2SbWidth ) ) − 1, csbfCtx is modified as follows:

csbfCtx += coded\_sub\_block\_flag[ xS + 1 ][ yS ] (9‑41)

* When yS is less than ( 1  <<  ( log2TbHeight − log2SbHeight ) ) − 1, csbfCtx is modified as follows:

csbfCtx += coded\_sub\_block\_flag[ xS ][ yS + 1 ] (9‑42)

The context index increment ctxInc is derived using the colour component index cIdx and csbfCtx as follows:

* If cIdx is equal to 0, ctxInc is derived as follows:

ctxInc = Min( csbfCtx, 1 ) (9‑43)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 2 + Min( csbfCtx, 1 ) (9‑44)

##### 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 ] (9‑45)  
 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 9.5.4.2.7 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 ) ) (9‑46)

* 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 ) (9‑47)

##### Derivation process of ctxInc for the syntax elements par\_level\_flag, abs\_level\_gt1\_flag, and abs\_level\_gt3\_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 9.5.4.2.7 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 ) (9‑48)

* 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 ) ) ) (9‑49)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 22 + ctxOffset + ( d  = =  0  ?  5  :  0 ) (9‑50)

#### Arithmetic decoding process

##### General

Inputs to this process are ctxTable, ctxIdx, and bypassFlag, as derived in subclause 9.5.4.2, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

Figure 9‑3 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table ctxTable, the ctxIdx and the bypassFlag are passed to the arithmetic decoding process DecodeBin( ctxTable, ctxIdx, bypassFlag ), which is specified as follows:

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in subclause 9.5.4.3.4 is invoked.

– Otherwise, if bypassFlag is equal to 0, ctxTable is equal to 0, and ctxIdx is equal to 0, DecodeTerminate( ) as specified in subclause 9.5.4.3.5 is invoked.

– Otherwise (bypassFlag is equal to 0 and ctxTable is not equal to 0), DecodeDecision( ctxTable, ctxIdx ) as specified in subclause 9.5.4.3.2 is invoked.



Figure 9‑3 – Overview of the arithmetic decoding process for a single bin (informative)

NOTE – Arithmetic coding is based on the principle of recursive interval subdivision. Given a probability estimation p( 0 ) and p( 1 ) = 1 − p( 0 ) of a binary decision ( 0, 1 ), an initially given code sub-interval with the range ivlCurrRange will be subdivided into two sub-intervals having range p( 0 ) \* ivlCurrRange and ivlCurrRange − p( 0 ) \* ivlCurrRange, respectively. Depending on the decision, which has been observed, the corresponding sub-interval will be chosen as the new code interval, and a binary code string pointing into that interval will represent the sequence of observed binary decisions. It is useful to distinguish between the most probable symbol(MPS) and the least probable symbol(LPS), so that binary decisions have to be identified as either MPS or LPS, rather than 0 or 1. Given this terminology, each context is specified by the probability pLPS of the LPS and the value of MPS (valMps), which is either 0 or 1. The arithmetic core engine in this Specification has three distinct properties:

– The probability estimation is performed by means of a finite-state machine with a table-based transition process between 64 different representative probability states { pLPS( pStateIdx ) | 0  <=  pStateIdx < 64 } for the LPS probability pLPS. The numbering of the states is arranged in such a way that the probability state with indexpStateIdx = 0 corresponds to an LPS probability value of 0.5, with decreasing LPS probability towards higher state indices.

– The range ivlCurrRange representing the state of the coding engine is quantized to a small set {Q1,…,Q4} of pre-set quantization values prior to the calculation of the new interval range. Storing a table containing all 64x4 pre-computed product values of Qi \* pLPS( pStateIdx ) allows a multiplication-free approximation of the product ivlCurrRange \* pLPS( pStateIdx ).

– For syntax elements or parts thereof for which an approximately uniform probability distribution is assumed to be given a separate simplified encoding and decoding bypass process is used.

##### Arithmetic decoding process for a binary decision

###### General

Inputs to this process are the variables ctxTable, ctxIdx, ivlCurrRange, and ivlOffset.

Outputs of this process are the decoded value binVal, and the updated variables ivlCurrRange and ivlOffset.

Figure 9‑4 shows the flowchart for decoding a single decision (DecodeDecision):

1. The value of the variable ivlLpsRange is derived as follows:

– Given the current value of ivlCurrRange, the variable qRangeIdx is derived as follows:

qRangeIdx = ivlCurrRange >> 5 (9‑51)

– Given qRangeIdx, pStateIdx0 and pStateIdx1 associated with ctxTable and ctxIdx, valMps and ivlLpsRange are derived as follows:

pState = pStateIdx1 + 16 \* pStateIdx0  
valMps = pState >> 14  
ivlLpsRange = ( qRangeIdx \* ( (valMps ? 32767 − pState : pState ) >> 9 ) >> 1 ) + 4 (9‑52)

1. The variable ivlCurrRange is set equal to ivlCurrRange − ivlLpsRange and the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 − valMps, ivlOffset is decremented by ivlCurrRange, and ivlCurrRange is set equal to ivlLpsRange.

– Otherwise, the variable binVal is set equal to valMps.

Given the value of binVal, the state transition isperformed as specified in subclause 9.5.4.3.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in subclause 9.5.4.3.3.

###### State transition process

Inputs to this process are the current pStateIdx0 and pStateIdx1, and the decoded value binVal.

Outputs of this process are the updated pStateIdx0 and pStateIdx1 of the context variable associated with ctxIdx.

The variable ctxIdxOffset is specified in Table 9‑5 depending on the current value of initType and ctxInc set equal to ctxIdx − ctxIdxOffset.

The variables shift0 and shift1 are derived from the shiftIdx value associated with ctxTable and ctxInc in clause 9.5.2.2.

shift0 = (shiftIdx >> 2) + 2   
shift1 = (shiftIdx & 3) + 3 + shift0 (9‑53)

Depending on the decoded value binVal, the update of the two variables pStateIdx0 and pStateIdx1 associated with ctxIdx is derived as follows:

pStateIdx0 = pStateIdx0 − (pStateIdx0 >> shift0) + (1023 \* binVal >> shift0)  
pStateIdx1 = pStateIdx1 − (pStateIdx1 >> shift1) + (16383 \* binVal >> shift1) (9‑54)



Figure 9‑4 – Flowchart for decoding a decision

##### Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from tile group data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset.

A flowchart of the renormalization is shown in Figure 9‑5. The current value of ivlCurrRange is first compared to 256 and further steps are specified as follows:

– If ivlCurrRange is greater than or equal to 256, no renormalization is needed and the RenormD process is finished;

– Otherwise (ivlCurrRange is less than 256), the renormalization loop is entered. Within this loop, the value of ivlCurrRange is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ).

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.



Figure 9‑5 – Flowchart of renormalization

##### Bypass decoding process for binary decisions

Inputs to this process are bits from tile group data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variable ivlOffset and the decoded value binVal.

The bypass decoding process is invoked when bypassFlag is equal to 1. Figure 9‑6 shows a flowchart of the corresponding process.

First, the value of ivlOffset is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ). Then, the value of ivlOffset is compared to the value of ivlCurrRange and further steps are specified as follows:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 and ivlOffset is decremented by ivlCurrRange.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0*.*

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.



Figure 9‑6 – Flowchart of bypass decoding process

##### Decoding process for binary decisions before termination

Inputs to this process are bits from tile group data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset, and the decoded value binVal.

This decoding process applies to decoding of end\_of\_tile \_one\_bit and pcm\_flag corresponding to ctxTable equal to 0 and ctxIdx equal to 0. Figure 9‑7 shows the flowchart of the corresponding decoding process, which is specified as follows:

First, the value of ivlCurrRange is decremented by 2. Then, the value of ivlOffset is compared to the value of ivlCurrRange and further steps are specified as follows:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated. The last bit inserted in register ivlOffset is equal to 1. When decoding end\_of\_tile \_one\_bit, this last bit inserted in register ivlOffset is interpreted as rbsp\_stop\_one\_bit.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0 and renormalization is performed as specified in subclause 9.5.4.3.3.

NOTE – This procedure may also be implemented using DecodeDecision( ctxTable, ctxIdx, bypassFlag ) with ctxTable = 0, ctxIdx = 0 and bypassFlag = 0. In the case where the decoded value is equal to 1, seven more bits would be read by DecodeDecision( ctxTable, ctxIdx, bypassFlag ) and a decoding process would have to adjust its bitstream pointer accordingly to properly decode following syntax elements.



Figure 9‑7 – Flowchart of decoding a decision before termination

# Sub-bitstream extraction process

Inputs to this process are a bitstream and a target highest TemporalId value tIdTarget.

Output of this process is a sub-bitstream.

It is a requirement of bitstream conformance for the input bitstream that any output sub-bitstream that is the output of the process specified in this clause with the bitstream, tIdTarget equal to any value in the range of 0 to 6, inclusive, and that satisfies the following condition shall be a conforming bitstream:

– The output sub-bitstream contains at least one VCL NAL unit with TemporalId equal to tIdTarget.

NOTE – A conforming bitstream contains one or more coded tile group NAL units with TemporalId equal to 0.

The output sub-bitstream is derived as follows:

– Remove all NAL units with TemporalId greater than tIdTarget.