ISO/IEC JTC 1/SC 29/WG 11

Coding of moving pictures and audio

Document type: **Approved WG 11 document**

Title: **Text of ISO/IEC CD 23094-1, Essential Video Coding**

Status: **Approved**

Date of document: **2019-07-22**

Source: **Video**

Expected action: **CD Ballot**

No. of pages: **292**

Email of convenor: leonardo@chiariglione.org

Committee URL: mpeg.chiariglione.org

INTERNATIONAL ORGANISATION FOR STANDARDISATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND AUDIO

ISO/IEC JTC1/SC29/WG11 N18568

July 2019, Gothenburg, Sweden

Source	**Video Subgroup**
Status	**Committee Draft**
Title	**Text of ISO/IEC CD 23094-1, Essential Video Coding**
Editors	**Kiho Choi, Jianle Chen, Dmytro Rusanovskyy, Min Woo Park, Chernyak Roman, Jonatan Samuelsson, Xiang Li, Xiaozhong Xu, Ken McCann**

Abstract
This document provides Committee Draft of Essential Video Coding.

ISO/IEC CD 23094-1

Error! Reference source not found.Error! Reference source not found./SC **Error! Reference source not found.**/WG **Error! Reference source not found.**

Secretariat: **Error! Reference source not found.**

Information technology – General Video Coding – Part 1: Essential Video Coding

CD stage

Warning for WDs and CDs

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword i
1 Scope 1
2 Normative references 1
2.1 International Standards 1
2.2 Additional references 1
3 Definitions 1
4 Abbreviations 5
5 Conventions 7
5.1 General 7
5.2 Arithmetic operators 7
5.3 Logical operators 7
5.4 Relational operators 7
5.5 Bit-wise operators 7
5.6 Assignment operators 8
5.7 Range notation 8
5.8 Mathematical functions 8
5.9 Order of operation precedence 9
5.10 Variables, syntax elements and tables 10
5.11 Text description of logical operations 11
5.12 Processes 12
6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 13
6.1 Bitstream formats 13
6.2 Source, decoded and output picture formats 13
6.3 Partitioning of pictures, slices, tiles, and CTUs 15
6.3.1 Partitioning of pictures into slices and tiles 15
6.3.2 Spatial or component-wise partitionings 15
6.4 Availability processes 16
6.4.1 Derivation process for neighbouring block availability 16
6.4.2 Derivation process for left and right neighbouring blocks availabilities 16
6.4.3 Derivation process for neighbouring block motion vector candidate availability 17
6.5 Scanning processes 17
6.5.1 CTB raster and tile scanning process 17
6.5.2 Zig-zag scan order 2D array initialization process 19
6.5.3 Zig-zag scan order 1D array initialization process 19
6.5.4 Inverse scan order 1D array initialization process 20
7 Syntax and semantics 21
7.1 Method of specifying syntax in tabular form 21
7.2 Specification of syntax functions and descriptors 22
7.3 Syntax in tabular form 23
7.3.1 NAL unit syntax 23
7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 24
7.3.3 Supplemental enhancement information message syntax 28
7.3.4 Slice header syntax 29
7.3.5 Reference picture list structure syntax 32
7.3.6 Slice data syntax 33
7.4 Semantics 45
7.4.1 General 45
7.4.2 NAL unit semantics 45
7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 47
7.4.4 Supplemental enhancement information message semantics 55
7.4.5 Slice header semantics 55

7.4.6 Reference picture list structure semantics 62
7.4.7 Slice data semantics 63
8 Decoding process 74
8.1 General decoding process 74
8.2 NAL unit decoding process 74
8.3 Slice decoding process 74
8.3.1 Decoding process for picture order count 74
8.3.2 Decoding process for reference picture lists construction 77
8.3.3 Decoding process for reference picture marking 79
8.3.4 Decoding process for collocated picture 80
8.4 Decoding process for coding units coded in intra prediction mode 80
8.4.1 General decoding process for coding units coded in intra prediction mode 80
8.4.2 Derivation process for luma intra prediction mode 81
8.4.3 Derivation process for chroma intra prediction mode 89
8.4.4 Decoding process of intra prediction 89
8.5 Decoding process for coding units coded in inter prediction mode 102
8.5.1 General decoding process for coding units coded in inter prediction mode 102
8.5.2 Derivation process for motion vector components and reference indices 105
8.5.3 Derivation process for affine motion vector components and reference indices 138
8.5.4 Decoding process for inter prediction samples 157
8.5.5 Decoder-Side Motion vector refinement process 167
8.5.6 Decoding process for the residual signal of coding units coded in inter prediction mode 172
8.6 Decoding process for coding units coded in ibc prediction mode 174
8.6.1 General decoding process for coding units coded in ibc prediction mode 174
8.6.2 Derivation process for motion vector components 175
8.6.3 Decoding process for ibc blocks 177
8.7 Scaling, transformation and array construction process prior to deblocking filter process 177
8.7.1 Derivation process for quantization parameters 177
8.7.2 Scaling and transformation process 178
8.7.3 Scaling process for transform coefficients 179
8.7.4 Transformation process for scaled transform coefficients 180
8.7.5 Picture construction process 188
8.7.6 Post-reconstruction filter process 188
8.8 In-loop filter process 191
8.8.1 General 191
8.8.2 Deblocking filter process 191
8.8.3 Enhanced deblocking filter process 197
8.8.4 Adaptive Loop Filter 202
9 Parsing process 206
9.1 General 206
9.2 Parsing process for 0-th order Exp-Golomb codes 207
9.2.1 General 207
9.2.2 Mapping process for signed Exp-Golomb codes 208
9.3 CABAC parsing process for slice data 209
9.3.1 General 209
9.3.2 Initialization process 209
9.3.3 Binarization process 226
9.3.4 Decoding process flow 231
Annex A Profiles and levels 242
A.1 Overview of profiles and levels 242
A.2 Requirements on video decoder capability 242
A.3 Profiles 242
A.3.1 General 242
A.3.2 Baseline profile 242
A.3.3 Main profile 243
A.4 Levels 243
A.4.2 Profile-specific level limits 245

A.4.3 Effect of level limits on picture rate for the video profiles (informative) 246
Annex B Byte stream format 250
B.1 General 250
B.2 Byte stream NAL unit syntax and semantics 250
B.2.1 Byte stream NAL unit syntax 250
B.2.2 Byte stream NAL unit semantics 250
Annex C Hypothetical reference decoder 251
C.1 General 251
C.2 Operation of coded picture buffer (CPB) 253
C.2.1 General 253
C.2.2 Timing of bitstream arrival 253
C.2.3 Timing of coded picture removal 254
C.3 Operation of the decoded picture buffer (DPB) 255
C.3.1 General 255
C.3.2 Removal of pictures from the DPB 255
C.3.3 Picture decoding and output 255
C.3.4 Current decoded picture marking and storage 256
C.4 Bitstream conformance 256
C.5 Decoder conformance 257
C.5.1 General 257
C.5.2 Operation of the output order DPB 258
C.5.2.1 General 258
C.5.2.2 Removal of pictures from the DPB 258
C.5.2.3 Current picture decoding, storage, and marking 258
Annex D Supplemental enhancement information 260
D.1 General 260
D.2 SEI payload syntax 260
D.2.1 General SEI message syntax 260
D.2.2 Buffering period SEI message syntax 261
D.2.3 Picture timing SEI message syntax 261
D.2.4 Recovery point SEI message syntax 262
D.2.5 Mastering display colour volume SEI message syntax 262
D.2.6 Content light level information SEI message syntax 263
D.3 SEI payload semantics 263
D.3.1 General SEI payload semantics 263
D.3.2 Buffering period SEI message semantics 263
D.3.3 Picture timing SEI message semantics 264
D.3.4 Recovery point SEI message semantics 267
D.3.5 Mastering display colour volume SEI message semantics 268
D.3.6 Content light level information SEI message semantics 269
Annex E Video usability information 270
E.1 General 270
E.2 VUI syntax 270
E.2.1 VUI parameters syntax 270
E.2.2 HRD parameters syntax 272
E.3 VUI semantics 272
E.3.1 VUI parameters semantics 272
E.3.2 HRD parameters semantics 283

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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

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

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

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

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

A list of all parts in the ISO/IEC 23094 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Information technology – General Video Coding – Part 1: Essential Video Coding

1 Scope

This International Standard specifies essential video coding.

2 Normative references

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

2.1 International Standards

– [Ed. Add references as needed.]

2.2 Additional references

– [Ed. Add references as needed.]

3 Definitions

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

3.1 **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*.

3.2 **bi-predictive (B) slice**: A *slice* that is decoded using *intra prediction* or using *inter prediction* with at most two *motion vectors* and *reference indices* to *predict* the sample values of each *block*.

3.3 **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)*.

3.4 **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.

3.5 **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.

3.6 **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.

3.7 **byte stream**: An encapsulation of a *NAL unit stream* containing *NAL unit length field* and *NAL units* as specified in Annex B.

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

3.9 **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 are often associated with the term chrominance.

3.10 **coded picture**: A *coded representation* of a *picture* containing all *CTUs* of the *picture*.

3.11 **coded picture buffer (CPB)**: A first-in first-out buffer containing *access units* in *decoding order* specified in the *hypothetical reference decoder* in Annex C.

3.12 **coded representation**: A data element as represented in its coded form.

3.13 **coded video sequence (CVS)**: A sequence of *access units* that consists, in *decoding order*, of an *IDR access unit*, followed by zero or more *access units* that are not *IDR access units*, including all subsequent *access units* up to but not including any subsequent *access unit* that is an *IDR access unit*.

3.14 **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*.

3.15 **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*.

3.16 **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.

3.17 **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.

3.18 **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.

3.19 **decoded picture**: A *decoded picture* is derived by decoding a *coded picture*.

3.20 **decoded picture buffer (DPB)**: A buffer holding *decoded pictures* for reference, output reordering, or output delay specified for the *hypothetical reference decoder* in Annex C.

3.21 **decoder**: An embodiment of a *decoding process*.

3.22 **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.

3.23 **decoding process**: The process specified in this document that reads a *bitstream* and derives *decoded pictures* from it.

3.24 **decoder under test (DUT)**: A *decoder* that is tested for conformance to this International Standard by operating the *hypothetical stream scheduler* to deliver a conforming *bitstream* to the *decoder* and to the *hypothetical reference decoder* and comparing the values and timing of the output of the two *decoders*.

3.25 **encoder**: An embodiment of an *encoding process*.

3.26 **encoding process**: A process not specified in this document that produces a *bitstream* conforming to this document.

3.27 **flag**: A variable or single-bit *syntax element* that can take one of the two possible values: 0 and 1.

3.28 **hypothetical reference decoder (HRD)**: A hypothetical *decoder* model that specifies constraints on the variability of conforming *NAL unit streams* or conforming *byte streams* that an encoding process may produce.

3.29 **hypothetical stream scheduler (HSS)**: A hypothetical delivery mechanism for the timing and data flow of the input of a *bitstream* into the *hypothetical reference decoder*. The HSS is used for checking the conformance of a *bitstream* or a *decoder*.

3.30 **informative**: A term used to refer to content provided in this document that does not establish any mandatory requirements for conformance to this document and thus is not considered an integral part of this document.

3.31 **instantaneous decoding refresh (IDR) access unit**: An *access unit* in which the *coded picture* is an *IDR picture*.

3.32 **instantaneous decoding refresh (IDR) picture**: A *coded picture* for which each *VCL NAL unit* has NalUnitType equal to IDR_NUT.

3.33 **inter coding**: Coding of a *coding block*, *slice*, or *picture* that uses *inter prediction*.

3.34 **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.

3.35 **intra coding**: Coding of a *coding block*, *slice*, or *picture* that uses *intra prediction*.

3.36 **intra prediction**: A *prediction* derived from only data elements (e.g., sample values) of the same decoded *slice* without referring to a *reference picture*.

3.37 **intra (I) slice**: A *slice* that is decoded using *intra prediction* only.

3.38 **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this document, 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.

3.39 **list 0 (list 1) motion vector**: A *motion vector* associated with a *reference index* pointing into a *reference picture list 0* (*list 1*).

3.40 **list 0 (list 1) prediction**: *Inter prediction* of the content of a *slice* using a *reference index* pointing into a *reference picture list 0* (*list 1*).

3.41 **long-term reference picture**: A *picture* that is marked as "used for long-term reference".

3.42 **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 are 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.

3.43 **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.

3.44 **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*.

3.45 **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 document (used exclusively in an *informative* context).

3.46 **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*.

3.47 **network abstraction layer (NAL) unit stream**: A sequence of *NAL units*.

3.48 **non-IDR picture**: A *coded picture* that is not an *IDR picture*.

3.49 **non-VCL NAL unit**: A *NAL unit* that is not a *VCL NAL unit*.

3.50 **note**: A term that is used to prefix *informative* remarks (used exclusively in an *informative* context).

3.51 **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*).

3.52 **parameter**: A *syntax element* of an *SPS* or *PPS*, or the second word of the defined term *quantization parameter*.

3.53 **partitioning**: The division of a set into subsets such that each element of the set is in exactly one of the subsets.

3.54 **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.

3.55 **picture parameter set (PPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *coded pictures* as determined by a *syntax element* found in each *slice header*.

3.56 **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*.

3.57 **prediction**: An embodiment of the *prediction process*.

3.58 **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.

3.59 **predictive (P) slice**: A *slice* that is decoded using *intra prediction* or using *inter prediction* with at most one *motion vector* and *reference index* to *predict* the sample values of each *block*.

3.60 **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.

3.61 **profile**: A specified subset of the syntax of this document.

3.62 **quantization parameter**: A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.

3.63 **random access**: The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.

3.64 **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.

3.65 **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.

3.66 **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*.

3.67 **reference index**: An index into a *reference picture list*.

3.68 **reference picture**: A *picture* that is a *short-term reference picture* or *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.

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

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

3.70 **reference picture list 0**: The *reference picture list* used for *inter prediction* of a *P* or the first *reference picture list* used for *inter prediction* of a *B slice*.

3.71 **reference picture list 1**: The second *reference picture list* used for *inter prediction* of a *B slice.*

3.72 **reserved**: A term that may be used to specify that some values of a particular *syntax element* are for future use by ISO/IEC and shall not be used in *bitstreams* conforming to this version of this document, but may be used in bitstreams conforming to future extensions of this document by ISO/IEC.

3.73 **sequence parameter set (SPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *CVSs* as determined by the content of a *syntax element* found in the *PPS* referred to by a *syntax element* found in each *slice header.*

3.74 **shall**: A term used to express mandatory requirements for conformance to this document.

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 document conforms to the decoding process requirements of this document.

3.75 **short-term reference picture**: A *picture* that is marked as "used for short-term reference".

3.76 **should**: A term used to refer to the behaviour of an implementation that is encouraged to be followed under anticipated ordinary circumstances, but is not a mandatory requirement for conformance to this document.

3.77 **source**: A term used to describe the video material or some of its attributes before encoding.

3.78 **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.

3.79 **syntax element**: An element of data represented in the *bitstream*.

3.80 **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order.

3.81 **tile**: A rectangular region of *CTUs* within a particular *tile column* and a particular *tile row* in a *picture*.

3.82 **tile column**: A rectangular region of *CTUs* having a height equal to the height of the *picture* and width specified by *syntax elements* in the *PPS*.

3.83 **slice**: An integer number of *tiles* of a *picture* in the tile raster scan of the *picture* and that are exclusively contained in a single *NAL unit*.

3.84 **slice header**: A part of a coded *slice* containing the data elements pertaining to the first or all *tiles* represented in the *slice*.

3.85 **tile row**: A rectangular region of *CTUs* having a height specified by *syntax elements* in the *PPS* and a width equal to the width of the *picture*.

3.86 **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*.

3.87 **transform block**: A rectangular MxN *block* of samples resulting from a *transform* in the *decoding process*.

3.88 **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*.

3.89 **tree**: A tree is a finite set of nodes with a unique root node.

3.90 **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 document and will not have a specified meaning in the future as an integral part of future versions of this document.

3.91 **video coding layer (VCL) NAL unit**: A collective term for *coded slice NAL units* and the subset of *NAL units* that have *reserved* values of NalUnitType that are classified as VCL NAL units in this document.

4 Abbreviations

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

APS	Adaptation Parameter Set
ATS	Adaptive Transform Selection
B	Bi-predictive
CBR	Constant Bit Rate
CPB	Coded Picture Buffer
CTB	Coding Tree Block
CTU	Coding Tree Unit
CVS	Coded Video Sequence
DPB	Decoded Picture Buffer
HRD	Hypothetical Reference Decoder
HSS	Hypothetical Stream Scheduler
I	Intra
IDR	Instantaneous Decoding Refresh
LSB	Least Significant Bit
LTRP	Long-Term Reference Picture
MMVD	Merge with Motion Vector Difference
MSB	Most Significant Bit
NAL	Network Abstraction Layer
P	Predictive
POC	Picture Order Count
PPS	Picture Parameter Set
QP	Quantization Parameter
RBSP	Raw Byte Sequence Payload
RGB	Same as GBR
SAR	Sample Aspect Ratio
SEI	Supplemental Enhancement Information

SODB	String Of Data Bits
SPS	Sequence Parameter Set
STRP	Short-Term Reference Picture
VBR	Variable Bit Rate
VCL	Video Coding Layer

5 Conventions

5.1 General

NOTE – The mathematical operators used in this document 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.

5.2 Arithmetic operators

The following arithmetic operators are defined as follows:

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

5.3 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.

5.4 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.

5.5 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.

5.6 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).

5.7 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.

5.8 Mathematical functions

The following mathematical functions are defined:

$$\text{Abs}(x) = \begin{cases} x & ; \quad x >= 0 \\ -x & ; \quad x < 0 \end{cases} \tag{5-1}$$

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)

$$\text{Atan2}(y, x) = \begin{cases} \text{Atan}\left(\frac{y}{x}\right) & ; \quad x > 0 \\ \text{Atan}\left(\frac{y}{x}\right) + \pi & ; \quad x < 0 \ \&\& \ y >= 0 \\ \text{Atan}\left(\frac{y}{x}\right) - \pi & ; \quad x < 0 \ \&\& \ y < 0 \\ +\frac{\pi}{2} & ; \quad x == 0 \ \&\& \ y >= 0 \\ -\frac{\pi}{2} & ; \quad \text{otherwise} \end{cases} \tag{5-4}$$

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

$Clip1_Y(x) = Clip3(0, (1 << BitDepth_Y) - 1, x)$ (5-6)

$Clip1_C(x) = Clip3(0, (1 << BitDepth_C) - 1, x)$ (5-7)

$$Clip3(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases} \quad (5\text{-}8)$$

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

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

$$GetCurrMsb(a, b, c, d) = \begin{cases} c + d & ; \quad b - a >= d / 2 \\ c - d & ; \quad a - b > d / 2 \\ c & ; \quad \text{otherwise} \end{cases} \quad (5\text{-}11)$$

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

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

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

$$Min(x, y) = \begin{cases} x & ; \quad x <= y \\ y & ; \quad x > y \end{cases} \quad (5\text{-}15)$$

$$Max(x, y) = \begin{cases} x & ; \quad x >= y \\ y & ; \quad x < y \end{cases} \quad (5\text{-}16)$$

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

$$Sign(x) = \begin{cases} 1 & ; \quad x > 0 \\ 0 & ; \quad x == 0 \\ -1 & ; \quad x < 0 \end{cases} \quad (5\text{-}18)$$

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

$Sqrt(x) = \sqrt{x}$ (5-20)

Swap(x, y) = (y, x) (5-21)

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

5.9 Order of operation precedence

When the 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 document is the same as used in the C programming language.

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

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

5.10 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 s_{yx}. A single column of a matrix may be referred to as a list and denoted by the 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.

5.11 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

5.12 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), the 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.

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

6.1 Bitstream formats

This clause specifies the relationship between the network abstraction layer (NAL) unit stream and byte stream, either of which is 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 NAL unit length field to form a stream of bytes. Methods of framing the NAL units in a manner other than the use of the byte stream format are outside the scope of this document. The byte stream format is specified in Annex B.

6.2 Source, decoded and output picture formats

This clause specifies the relationship between source and decoded pictures that are 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 the convenience of notation and terminology in this document, 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 E.

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. Other values of chroma_format_idc, SubWidthC and SubHeightC may be specified in the future by ISO/IEC.

Table 6-1 – SubWidthC and SubHeightC values derived from chroma_format_idc

chroma_format_idc	Chroma format	SubWidthC	SubHeightC
0	Monochrome	1	1
1	4:2:0	2	2
2	4:2:2	2	1
3	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, each of the two chroma arrays has the same height and width as the luma array.

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 E).

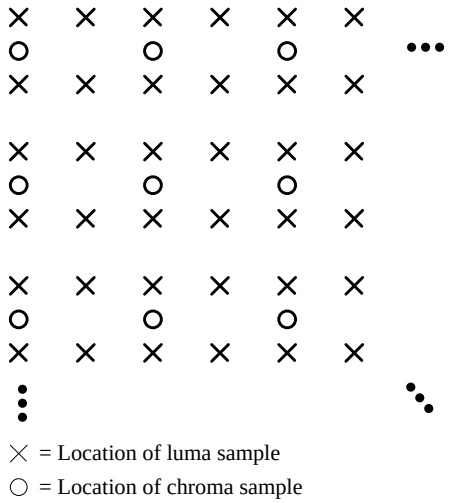

Figure 6-1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture

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.

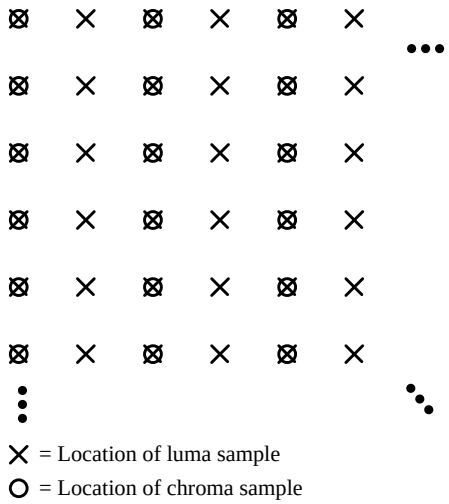

Figure 6-2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture

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.

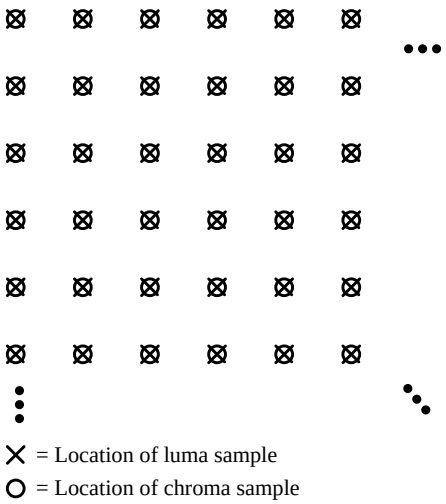

Figure 6-3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture

6.3 Partitioning of pictures, slices, tiles, and CTUs

6.3.1 Partitioning of pictures into slices and tiles

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

Pictures are divided into slices and tiles. A tile is a group of CTUs that cover a rectangular region of a picture. A slice is a group of tiles that cover a rectangular region of a picture.

For example, a picture may be divided into 24 tiles (6 tile columns and 4 tile rows) and 9 slices as shown in Figure 6-4.

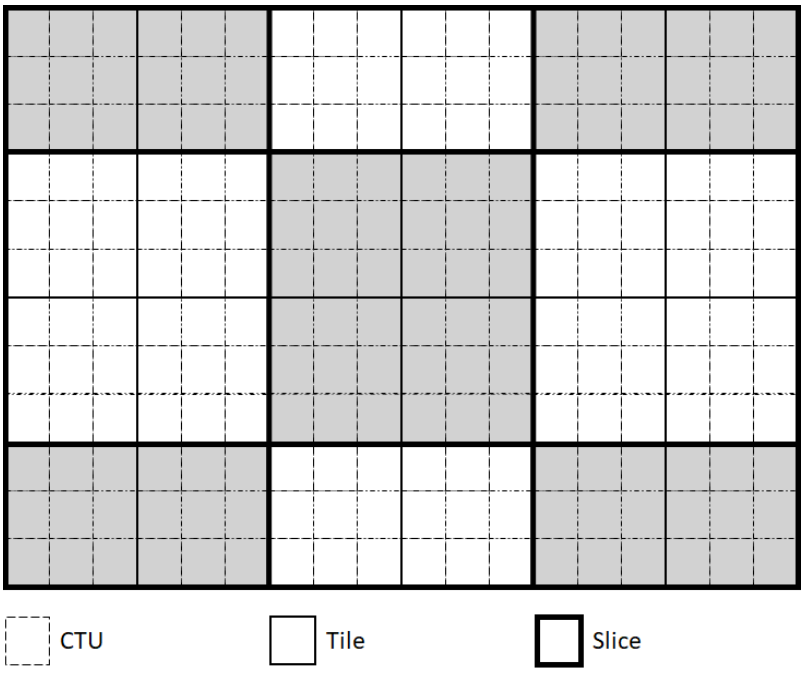

Figure 6-4 – A picture with 18 by 12 luma CTUs that is partitioned into 24 tiles and 9 slices (informative)

6.3.2 Spatial or component-wise partitionings

The following divisions of processing elements of this document 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 slices
– The division of each slice into tiles
– The division of each slice into CTUs
– The division of each CTU into CTBs

6.4 Availability processes

6.4.1 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 or if (xNbY, yNbY) is located out of current picture boundary or if the neighboring block has not been coded, availableN is set equal to FALSE

6.4.2 Derivation process for left and right neighbouring blocks availabilities

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,
– variables nCbW and nCbH specifying the width and height of the current block.

Output of this process is left and right availabilities of the neighbouring blocks, denoted as availLR

A variable availLR, which indicates the left and right neighbouring blocks availabilities, is derived as follows:

– The left luma location (xNbL, yNbL) inside the neighbouring luma coding block is set equal to (xCurr - 1, yCurr).
– The right luma location (xNbR, yNbR) inside the neighbouring luma coding block is set equal to (xCurr + nCbW, yCurr).
– Clause 6.4.1 is invoked with the left luma location (xNbL, yNbL) as inputs, and the output is assigned to the coding block availability flag availableL.
– Clause 6.4.1 is invoked with the right luma location (xNbR, yNbR) as inputs, and the output is assigned to the coding block availability flag availableR.
– The left and right neighbouring availability availLR is derived by availLR = availableL + availableR * 2

availLR can be equal to LR_00, LR_10, LR_01, LR_11, where LR_00 denotes the availabilty availLR equal to 0 when both left and right neighbouring blocks are not available; LR_10 denotes the availabilty availLR equal to 1 when left neighbouring block is available but right block is not available; LR_01 denotes the availabilty availLR equal to 1 when left neighbouring block is not available but the right block is available; LR_11 denotes the availabilty availLR equal to 1 when both left and right neighbouring block are available.

6.4.3 Derivation process for neighbouring block motion vector candidate 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 slice or tile than the current block or if (xNbY, yNbY) is located out of current picture boundary or if the neighboring block has not been coded or if the neighboring block is coded in intra mode, availableN is set equal to FALSE

– Otherwise, availableN is set equal to TRUE.

6.5 Scanning processes

6.5.1 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 ] ] =
                        explicit_tile_id_flag ? tile_id_val[ i ][ j ] : tileIdx
```

The list NumCtusInTile[tileIdx] for tileIdx ranging from 0 to PicSizeInCtbsY − 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 set TileIdToIdx[tileId] for a set of NumTilesInPic tileId values specifying the conversion from a tile ID to a tile index and 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 ) {
        TileIdToIdx[ TileId[ ctbAddrTs ] ] = tileIdx
        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.

6.5.2 Zig-zag scan order 2D array initialization process

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

Output of this process is the array zigZagScan[sPos][sComp]. The array index sPos specifies 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 zigZagScan is derived as follows:

```
i = 0
x = 0
y = 0
diagScan[ i ][ 0 ] = x
diagScan[ i ][ 1 ] = y
i++
for( line = 1; line < (blkWidth + blkHeight – 1); line++ ) {
    if( line % 2 ) {
        x = Min( line, blkWidth – 1 )
        y = Max( 0, line – ( blkWidth – 1 ) )                    (6-10)
        while( x >= 0 && y < blkHeight ) {
            zigZagScan[ i ][ 0 ] = x
            zigZagScan[ i ][ 1 ] = y
            i++
            x– –
            y++
        }
    }
    else {
        y = Min( line, blkHeight – 1 )
        x = Max( 0, line – ( blkHeight – 1) )
        while( y >= 0 && x < blkWidth ) {
            zigZagScan[ i ][ 0 ] = x
            zigZagScan[ i ][ 1 ] = y
            i++
            x++
            y– –
        }
    }
}
```

6.5.3 Zig-zag scan order 1D array initialization process

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

Output of this process is the array scan[pos]. The array index sPos specifies the scan position ranging from 0 to (blkWidth * blkHeight) – 1. Depending on the value of blkWidth and blkHeight, the array zigZagScan is derived as follows:

```
pos = 0
num_line = blkWidth + blkHeight - 1
zigZagScan pos] = 0
pos++
for(l = 1; l < num_line; l++)
{
    if(l % 2) {
```

```
        x = Min(l, size_x - 1)
        y = Max(0, l - (size_x - 1))
        while(x >= 0 && y < size_y){
            scan[pos] = y * size_x + x
            pos++
            x--
            y++
        }
    }
    else{
        y = Min(l, size_y - 1)
        x = Max(0, l - (size_y - 1))
        while(y >= 0 && x < size_x){
            scan[pos] = y * size_x + x
            pos++
            x++
            y--
        }
    }
}
```

6.5.4 Inverse scan order 1D array initialization process

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

Output of this process is the array inverseScan[scanPos]. The array index scanPos specifies the position in the scan order ranging from 0 to (blkWidth * blkHeight) – 1. Depending on the value of blkWidth and blkHeight, the array inverseScan is derived as follows:

- The variables forwardScan is derived by invoking zig-zag scan order 1D array initialization process as specified in clause 6.5.3 with input parameters blkWidth and blkHeight.
- The output variables inverseScan is derived as follows:

```
for ( pos = 0; pos < blkWidth * blkHeight; pos ++){
    inverseScan[ forwardScan[ pos ] ] = pos
}
```


7 Syntax and semantics

7.1 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 document.

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(v)
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	

7.2 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 B, 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 B 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 B 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 9.3.

– 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

– 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 document when the current position in the bitstream is a byte-aligned position.

– 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.

7.3 Syntax in tabular form

7.3.1 NAL unit syntax

7.3.1.1 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)
}	

7.3.1.2 NAL unit header syntax

nal_unit_header( ) {	**Descriptor**
forbidden_zero_bit	f(1)
nal_unit_type_plus1	u(6)
nuh_temporal_id	u(3)
nuh_reserved_zero_5bits	u(5)
nuh_extension_flag	u(1)
}	

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

7.3.2.1 SPS RBSP syntax

seq_parameter_set_rbsp( ) {	**Descriptor**
sps_seq_parameter_set_id	ue(v)
profile_idc	u(7)
level_idc	u(8)
chroma_format_idc	ue(v)
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)
sps_btt_flag	u(1)
if(sps_btt_flag)	
log2_ctu_size_minus2	ue(v)
log2_diff_ctu_max_11_cb_size	ue(v)
log2_diff_max_11_min_11_cb_size	ue(v)
log2_diff_max_11_max_12_cb_size	ue(v)
log2_diff_min_11_min_12_cb_size_minus1	ue(v)
log2_diff_max_12_max_14_cb_size_minus1	ue(v)
log2_diff_min_12_min_14_cb_size_minus1	ue(v)
log2_diff_max_11_max_tt_cb_size_minus1	ue(v)
log2_diff_min_11_min_tt_cb_size_minus2	ue(v)
}	
sps_suco_flag	u(1)
if(sps_suco_flag) {	
log2_diff_ctu_size_max_suco_cb_size	ue(v)
log2_diff_max_suco_min_suco_cb_size	ue(v)
}	
sps_amvr_flag	u(1)
sps_mmvd_flag	u(1)
sps_affine_flag	u(1)
sps_dmvr_flag	u(1)
sps_alf_flag	u(1)
sps_admvp_flag	u(1)

sps_eipd_flag	u(1)
sps_adcc_flag	u(1)
sps_amis_flag	u(1)
sps_ibc_enabled_flag	u(1)
if(sps_ibc_enabled_flag)	
max_ibc_cand_size_minus2	ue(v)
sps_iqt_flag	u(1)
sps_htdf_flag	u(1)
sps_addb_flag	u(1)
sps_cm_init_flag	u(1)
sps_ats_flag	u(1)
sps_rpl_flag	u(1)
sps_pocs_flag	u(1)
sps_dquant_flag	u(1)
if(sps_pocs_flag)	
log2_max_pic_order_cnt_lsb_minus4	ue(v)
if(!sps_pocs_flag || !sps_rpl_flag) {	
log2_sub_gop_length	ue(v)
if(log2_sub_gop_length = = 0)	
log2_ref_pic_gap_length	ue(v)
}	
sps_max_dec_pic_buffering_minus1	ue(v)
if(!sps_rpl_flag)	
max_num_tid0_ref_pics	ue(v)
else {	
long_term_ref_pics_flag	u(1)
rpl_candidates_present_flag	u(1)
if(rpl_candidates_present_flag) {	
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, long_term_ref_pics_flag)	
}	
}	
}	
picture_cropping_flag	u(1)
if(picture_cropping_flag) {	
picture_crop_left_offset	ue(v)
picture_crop_right_offset	ue(v)
picture_crop_top_offset	ue(v)
picture_crop_bottom_offset	ue(v)
}	
vui_parameters_present_flag	u(1)
if(vui_parameters_present_flag)	
vui_parameters( )	
rbsp_trailing_bits( )	
}	

7.3.2.2 PPS RBSP syntax

pic_parameter_set_rbsp( ) {	**Descriptor**
pps_pic_parameter_set_id	ue(v)
pps_seq_parameter_set_id	ue(v)
for(i = 0; i < 2; i++)	
num_ref_idx_default_active_minus1[i]	ue(v)
if(sps_rpl_flag) {	
if(long_term_ref_pics_flag)	
additional_lt_poc_lsb_len	ue(v)
if(rpl_candidates_present_flag)	
rpl1_idx_present_flag	u(1)
}	
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)
}	
loop_filter_across_tiles_enabled_flag	u(1)
tile_offset_lens_minus1	ue(v)
}	
tile_id_len_minus1	ue(v)
explicit_tile_id_flag	u(1)
if(explicit_tile_id_flag)	
for(i = 0; i <= num_tile_rows_minus1; i++)	
for(j = 0; j <= num_tile_columns_minus1; j++)	
tile_id_val[i][j]	u(v)
arbitrary_slice_present_flag	u(1)
constrained_intra_pred_flag	u(1)
cu_qp_delta_enabled_flag	u(1)
if(cu_qp_delta_enabled_flag)	
log2_cu_qp_delta_area_minus6	ue(v)
rbsp_trailing_bits( )	
}	

7.3.2.3 APS RBSP 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( )	
}	

7.3.2.4 Filler data RBSP syntax

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

7.3.2.5 Supplemental enhancement information RBSP syntax

sei_rbsp( ) {	**Descriptor**
do	
sei_message( )	
while(more_rbsp_data( ))	
rbsp_trailing_bits( )	
}	

7.3.2.6 Slice layer RBSP syntax

slice_layer_rbsp( ) {	**Descriptor**
slice_header( )	
slice_data( )	
rbsp_slice_trailing_bits( )	
}	

7.3.2.7 RBSP slice trailing bits syntax

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

7.3.2.8 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)
}	

7.3.2.9 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)
}	

7.3.3 Supplemental enhancement information message syntax

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

7.3.4 Slice header syntax

slice_header( ) {	**Descriptor**
slice_pic_parameter_set_id	ue(v)
single_tile_in_slice_flag	u(1)
first_tile_id	u(v)
if(!single_tile_in_slice_flag) {	
if(arbitrary_slice_present_flag)	
arbitrary_slice_flag	u(1)
if(!arbitrary_slice_flag)	
last_tile_id	u(v)
else {	
num_remaining_tiles_in_slice_minus1	ue(v)
for(i = 0; i < NumTilesInSlice − 1; i++)	
delta_tile_id_minus1[i]	ue(v)
}	
}	
slice_type	ue(v)
if(nal_unit_type = = IDR_NU)	
no_output_of_prior_pics_flag	u(1)
if(sps_mmvd_flag && slice_group_type = = B)	
mmvd_group_enable_flag	u(1)
if(sps_alf_flag) {	
slice_alf_enabled_flag	u(1)
if(slice_alf_enabled_flag) {	
slice_aps_id	u(5)
slice_alf_map_signalled	u(1)
}	
}	
if(NalUnitType != IDR_NUT) {	
if(sps_pocs_flag)	
slice_pic_order_cnt_lsb	u(v)
if(sps_rpl_flag) {	
for(i = 0; i < 2; i++) {	
if(rpl_candidates_present_flag && (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], long_term_ref_pics_flag)	
for(j = 0; j < num_ltrp_entries[i][SliceRplsIdx[i]]; j++) {	
additional_poc_lsb_present_flag[i][j]	u(1)
if(additional_poc_lsb_present_flag[i][j])	
additional_poc_lsb_val[i][j]	u(v)
}	

}	
if(slice_type = = P || slice_type = = B) {	
num_ref_idx_active_override_flag	u(1)
if(num_ref_idx_active_override_flag)	
for(i = 0; i < (slice_type = = B ? 2: 1); i++)	
num_ref_idx_active_minus1[i]	ue(v)
if (addmvp_flag) {	
temporal_mvp_asigned_flag	u(1)
if(temporal_mvp_asigned_flag) {	
if(slice_type = = B) {	
col_pic_list_idx	u(1)
col_source_mvp_list_idx	u(1)
}	
col_pic_ref_idx	ue(v)
}	
}	
}	
}	
slice_deblocking_filter_flag	u(1)
if(slice_deblocking_filter_flag && sps_addb_flag) {	
slice_alpha_offset	se(v)
slice_beta_offset	se(v)
}	
slice_qp	u(6)
slice_cb_qp_offset	se(v)
slice_cr_qp_offset	se(v)
if(!single_tile_in_slice_flag)	
for(i = 0; i < NumTilesInSlice − 1; i++)	
entry_point_offset_minus1[i]	u(v)
byte_alignment( )	
}	

7.3.4.1 Adaptive loop filter data syntax

alf_data( ) {	**Descriptor**
alf_luma_filter_signal_flag	u(1)
alf_chroma_idc	tu(v)
if(alf_luma_filter_signal_flag) {	
alf_luma_num_filters_signalled_minus1	tb(v)
alf_luma_type_flag	u(1)
if(alf_luma_num_filters_signalled_minus1 > 0) {	
for(i = 0; i < NumAlfFilters; i++)	
alf_luma_coeff_delta_idx[i]	tb(v)
}	
alf_luma_fixed_filter_usage_pattern	uek(v)
if(alf_luma_fixed_filter_usage_pattern = = 2) {	
for (i = 0; i < NumAlfFilters; i++)	
alf_luma_fixed_filter_usage[i **]**	u(1)
}	
if (alf_luma_fixed_filter_usage_pattern > 0) {	
for(i = 0; i < NumAlfFilters; i++) {	
if(alf_luma_fixed_filter_usage[i])	
alf_luma_coeff_delta_idx[i]	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 < max_golomb_idx; i++)	
alf_luma_eg_order_increase_flag[i]	u(1)

if(alf_luma_coeff_delta_flag) {	
for(i = 0; i < NumAlfFilters; i++)	
alf_luma_coeff_flag[i]	u(1)
}	
for(i = 0; i < NumAlfFilters; i++) {	
if(alf_luma_coeff_flag[i]) {	
for (j = 0; j < NumAlfCoefs – 1; j++) {	
alf_luma_coeff_delta_abs[i][j]	uek(v)
if(alf_luma_coeff_delta_abs[i][j])	
alf_luma_coeff_delta_sign[i][j]	u(1)
}	
}	
}	
}	
if(alf_chroma_idc > 0) {	
alf_chroma_min_eg_order_minus1	ue(v)
for(i = 0; i < max_golomb_idx; 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)
}	
}	
}	

7.3.5 Reference picture list structure syntax

ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) {	**Descriptor**
num_strp_entries[listIdx][rplsIdx]	ue(v)
if(ltrpFlag)	
num_ltrp_entries[listIdx][rplsIdx]	ue(v)
for(i = 0; i < NumEntriesInList[listIdx][rplsIdx]; i++) {	
if(num_ltrp_entries[listIdx][rplsIdx] > 0)	
lt_ref_pic_flag[listIdx][rplsIdx][i]	u(1)
if(!lt_ref_pic_flag[listIdx][rplsIdx][i]) {	
delta_poc_st[listIdx][rplsIdx][i]	ue(v)
if(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)
}	
}	

7.3.6 Slice data syntax

7.3.6.1 General slice data syntax

slice_data( ) {	Descriptor
for(i = 0; i < NumTilesInSlice; i++) {	
NumHmvpCand = 0	
ctbAddrInTs = FirstCtbAddrTs[SliceTileIdx[i]]	
for(j = 0; j < NumCtusInTile[SliceTileIdx[i]]; j++, ctbAddrInTs++) {	
CtbAddrInRs = CtbAddrTsToRs[ctbAddrInTs]	
coding_tree_unit( )	
}	
end_of_tile_one_bit /* equal to 1 */	ae(v)
if(i < NumTilesInSlice − 1)	
byte_alignment( )	
}	
}	

7.3.6.2 Coding tree unit syntax

coding_tree_unit( ) {	Descriptor
xCtb = (CtbAddrInRs % PicWidthInCtbsY) << CtbLog2SizeY	
yCtb = (CtbAddrInRs / PicWidthInCtbsY) << CtbLog2SizeY	
if(slice_alf_enabled_flag && slice_alf_map_signalled){	
alf_ctb_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY]	u(1)
}	
split_unit(xCtb, yCtb, CtbLog2SizeY, CtbLog2SizeY, 0, 0)	
}	

7.3.6.3 Split unit syntax

split_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth, splitUnitOrder) {	Descriptor
if(sps_btt_flag = = 0) {	
if(log2CbWidth > 2 | | log2CbHeight > 2) {	
split_cu_flag[x0][y0]	ae(v)
}	
}	
else if(sps_btt_flag = = 1) {	

if(log2CbWidth > 2 | | log2CbHeight > 2 &&
x0 + (1 << log2CbWidth) <= pic_width_in_luma_samples &&
y0 + (1 << log2CbHeight) <= pic_height_in_luma_samples) {	

if(allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor)	
btt_split_flag[x0][y0]	ae(v)
if(btt_split_flag[x0][y0]) {	
if((allowSplitBtVer | | allowSplitTtVer) && (allowSplitBtHor | | allowSplitTtHor))	
btt_split_dir[x0][y0]	ae(v)

if((btt_split_dir[x0][y0] && allowSplitBtVer && allowSplitTtVer) | | (!btt_split_dir[x0][y0] && allowSplitBtHor && allowSplitTtHor))	
btt_split_type[x0][y0]	ae(v)
}	
if (cu_qp_delta_enabled_flag && sps_dquant_flag){	
if(btt_split_flag[x0][y0] = = 0 && log2CbWidth + log2CbHeight >= cuQpDeltaArea)	
cuQpDeltaCode = 1	
else if(log2CbWidth + log2CbHeight + 1 = = cuQpDeltaArea && btt_split_type[x0][y0] = = 1)	
cuQpDeltaCode = 2	
}	
}	
}	
if(sps_suco_flag && allowSplitUnitCodingOrder)	
split_unit_coding_order_flag[x0][y0]	ae(v)
if(split_cu_flag[x0][y0]) {	
x1 = x0 + (1 << (log2CbWidth – 1))	
y1 = y0 + (1 << (log2CbHeight – 1))	
split_unit(x0, y0, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 1, 0)	
if(x1 < pic_width_in_luma_samples)	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 1, 0)	
if(y1 < pic_height_in_luma_samples)	
split_unit(x0, y1, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 1, 0)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
split_unit(x1, y1, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 1, 0)	
}	
else if(SplitMode[x0][y0] = = SPLIT_QUAD) {	
x1 = x0 + (1 << (log2CbWidth – 1))	
y1 = y0 + (1 << (log2CbHeight – 1))	
if(split_unit_coding_order_flag[x0][y0] = = 0) {	
split_unit(x0, y0, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 0)	
if(x1 < pic_width_in_luma_samples)	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 0)	
if(y1 < pic_height_in_luma_samples)	
split_unit(x0, y1, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 0)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
split_unit(x1, y1, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 0)	
}	
else {	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 1)	
split_unit(x0, y0, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 1)	
split_unit(x1, y1, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 1)	
split_unit(x0, y1, log2CbWidth – 1, log2CbHeight – 1, ctDepth + 2, 1)	
}	
}	

else if(SplitMode[x0][y0] = = SPLIT_BT_VER) {	
x1 = x0 + (1 << (log2CbWidth – 1))	
if(split_unit_coding_order_flag[x0][y0] = = 0) {	
split_unit(x0, y0, log2CbWidth – 1, log2CbHeight, ctDepth + 1, 0)	
if(x1 < pic_width_in_luma_samples)	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight, ctDepth + 1, 0)	
}	
else {	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight, ctDepth + 1, 1)	
split_unit(x0, y0, log2CbWidth – 1, log2CbHeight, ctDepth + 1, 1)	
}	
}	
else if(SplitMode[x0][y0] = = SPLIT_BT_HOR) {	
y1 = y0 + (1 << (log2CbHeight – 1))	
split_unit(x0, y0, log2CbWidth , log2CbHeight– 1, ctDepth + 1, splitUnitOrder)	
if(y1 < pic_height_in_luma_samples)	
split_unit(x0, y1, log2CbWidth , log2CbHeight– 1, ctDepth + 1, splitUnitOrder)	
}	
else if(SplitMode[x0][y0] = = SPLIT_TT_VER) {	
x1 = x0 + (1 << (log2CbWidth – 2))	
x2 = x1 + (1 << (log2CbWidth – 1))	
if(split_unit_coding_order_flag[x0][y0] = = 0) {	
split_unit(x0, y0, log2CbWidth – 2, log2CbHeight, ctDepth + 2, 0)	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight, ctDepth + 1, 0)	
split_unit(x2, y0, log2CbWidth – 2, log2CbHeight, ctDepth + 2, 0)	
}	
else {	
split_unit(x2, y0, log2CbWidth – 2, log2CbHeight, ctDepth + 2, 1)	
split_unit(x1, y0, log2CbWidth – 1, log2CbHeight, ctDepth + 1, 1)	
split_unit(x0, y0, log2CbWidth – 2, log2CbHeight, ctDepth + 2, 1)	
}	
}	
else if(SplitMode[x0][y0] = = SPLIT_TT_HOR) {	
y1 = y0 + (1 << (log2CbHeight – 2))	
y2 = y1 + (1 << (log2CbHeight – 1))	
split_unit(x0, y0, log2CbWidth, log2CbHeight – 2, ctDepth + 2, splitUnitOrder)	
split_unit(x0, y1, log2CbWidth, log2CbHeight – 1, ctDepth + 1, splitUnitOrder)	
split_unit(x0, y2, log2CbWidth, log2CbHeight – 2, ctDepth + 2, splitUnitOrder)	
}	
else	
coding_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth)	
}	

7.3.6.4 Coding unit syntax

coding_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth) {	**Descriptor**

if(slice_type != I && ! (sps_amis_flag = = 1 && log2CbWidth = = 2 && log2CbHeight = = 2))	
cu_skip_flag[x0][y0]	ae(v)
if(cu_skip_flag[x0][y0]) {	
if(sps_amis_flag = = 0) {	
mvp_idx_l0[x0][y0]	ae(v)
if(slice_type != B)	
mvp_idx_l1[x0][y0]	ae(v)
}	
else if(sps_amis_flag = = 1) {	
if(sps_mmvd_flag)	
mmvd_flag[x0][y0]	ae(v)
if(mmvd_flag[x0][y0]) {	
if(mmvd_group_enable_flag && log2CbWidth + log2CbHeight > 5)	
mmvd_group_idx[x0][y0]	ae(v)
mmvd_merge_idx[x0][y0]	ae(v)
mmvd_distance_idx[x0][y0]	ae(v)
mmvd_direction_idx[x0][y0]	ae(v)
}	
else {	
if(sps_affine_flag && log2CbWidth >= 3 && log2CbHeight >= 3)	
affine_flag[x0][y0]	ae(v)
if(affine_flag[x0][y0])	
affine_merge_idx[x0][y0]	ae(v)
else	
mvp_idx[x0][y0]	ae(v)
}	
}	
}	
else {	
if(slice_type != I && ! (sps_amis_flag = = 1 && log2CbWidth = = 2 && log2CbHeight = = 2))	
pred_mode_flag[x0][y0]	ae(v)
if(sps_ibc_enabled_flag && log2CbWidth < = log2MaxIbcCandSize && log2CbHeight < = log2MaxIbcCandSize && (slice_type = = I | | CuPredMode[x0][y0] != MODE_INTRA))	
ibc_flag[x0][y0]	ae(v)
if(CuPredMode[x0][y0] = = MODE_INTRA) {	
if(sps_eipd_flag = = 0)	
intra_pred_mode[x0][y0]	ae(v)
else if(sps_eipd_flag = = 1) {	
intra_luma_pred_mpm_flag[x0][y0]	ae(v)
if(intra_luma_pred_mpm_flag[x0][y0])	
intra_luma_pred_mpm_idx[x0][y0]	ae(v)
else {	
intra_luma_pred_pims_flag[x0][y0]	ae(v)
if(intra_luma_pred_pims_flag[x0][y0])	

intra_luma_pred_pims_idx[x0][y0]	ae(v)
else	
intra_luma_pred_rem_mode[x0][y0]	ae(v)
}	
if(ChromaArrayType != 0)	
intra_chroma_pred_mode[x0][y0]	ae(v)
}	
}	
else { /* (CuPredMode[x0][y0] != MODE_INTRA) */	
if(ibc_flag)	
{	
abs_mvd_l0[x0][y0][0]	ae(v)
if(abs_mvd_l0[x0][y0][0])	
mvd_l0_sign_flag[x0][y0][0]	ae(v)
abs_mvd_l0[x0][y0][1]	ae(v)
if(abs_mvd_l0[x0][y0][1])	
mvd_l0_sign_flag[x0][y0][1]	ae(v)
}	
else{	
if(sps_amvr_flag)	
amvr_idx[x0][y0]	ae(v)
if(slice_type = = B && sps_amis_flag = = 0)	
direct_mode_flag[x0][y0]	ae(v)
else if(sps_amis_flag = = 1) {	
if(amvr_idx[x0][y0] = = 0)	
merge_mode_flag[x0][y0]	ae(v)
if(merge_mode_flag[x0][y0]) {	
if(sps_mmvd_flag)	
mmvd_flag[x0][y0]	ae(v)
if(mmvd_flag[x0][y0]) {	
if(mmvd_group_enable_flag && log2CbWidth + log2CbHeight > 5)	
mmvd_group_idx[x0][y0]	ae(v)
mmvd_merge_idx[x0][y0]	ae(v)
mmvd_distance_idx[x0][y0]	ae(v)
mmvd_direction_idx[x0][y0]	ae(v)
}	
else {	
if(sps_affine_flag && log2CbWidth >= 3 && log2CbHeight >= 3)	
affine_flag[x0][y0]	ae(v)
if(affine_flag[x0][y0]) {	
affine_merge_idx[x0][y0]	ae(v)
}	
else	
mvp_idx[x0][y0]	ae(v)
}	
}	
}	
if(direct_mode_flag[x0][y0] = = 0 && merge_mode_flag[x0][y0] = = 0) {	

if(slice_type = = B && ! (log2CbWidth < 3 && log2CbHeight < 3))	
inter_pred_idc[x0][y0]	
if(sps_amis_flag = = 0) {	
if(inter_pred_idc[x0][y0] != PRED_L1) {	
if(num_ref_idx_active_minus1[0] > 0)	
ref_idx_l0[x0][y0]	ae(v)
mvp_idx_l0[x0][y0]	ae(v)
abs_mvd_l0[x0][y0][0]	ae(v)
if(abs_mvd_l0[x0][y0][0])	
mvd_l0_sign_flag[x0][y0][0]	ae(v)
abs_mvd_l0[x0][y0][1]	ae(v)
if(abs_mvd_l0[x0][y0][1])	
mvd_l0_sign_flag[x0][y0][1]	ae(v)
}	
if(inter_pred_idc[x0][y0] != PRED_L0) {	
if(num_ref_idx_active_minus1[1] > 0)	
ref_idx_l1[x0][y0]	ae(v)
mvp_idx_l1[x0][y0]	ae(v)
abs_mvd_l1[x0][y0][0]	ae(v)
if(abs_mvd_l1[x0][y0][0])	
mvd_l1_sign_flag[x0][y0][0]	ae(v)
abs_mvd_l1[x0][y0][1]	ae(v)
if(abs_mvd_l1[x0][y0][1])	
mvd_l1_sign_flag[x0][y0][1]	ae(v)
}	
}	
else if(sps_amis_flag= = 1) {	
if(sps_affine_flag && log2CbWidth >= 4 && log2CbHeight >= 4 && amvr_idx[x0][y0] = = 0)	
affine_flag[x0][y0]	ae(v)
if(affine_flag[x0][y0]) {	
affine_mode[x0][y0]	ae(v)
vertexNum = 1+ affine_flag[x0][y0] + affine_mode[x0][y0]	
if(inter_pred_idc[x0][y0] != PRED_L0 || inter_pred_idc[x0][y0] = = PRED_BI) {	
ref_idx_l0[x0][y0]	ae(v)
mvp_idx_l0[x0][y0]	ae(v)
mvd_flag_l0[x0][y0]	ae(v)
for(vertex = 0; vertex < vertexNum; vertex++) {	
if (mvd_flag_l0[x0][y0]) {	
abs_mvd_l0[vertex][0][x0][y0]	ae(v)
if(abs_mvd_l0[vertex][0][x0][y0])	
mvd_l0_sign_flag[vertex][0][x0][y0]	ae(v)
abs_mvd_l0[vertex][1][x0][y0]	ae(v)
if(abs_mvd_l0[vertex][1][x0][y0])	
mvd_l0_sign_flag[vertex][1][x0][y0]	ae(v)

}	
}	
}	
if(inter_pred_idc[x0][y0] != PRED_L1 ‖ inter_pred_idc[x0][y0] = = PRED_BI) {	
ref_idx_l1[x0][y0]	ae(v)
mvp_idx_l1[x0][y0]	ae(v)
mvd_flag_l1[x0][y0]	ae(v)
for(vertex = 0; vertex < vertexNum; vertex++) {	
if (mvd_flag_l1[x0][y0]) {	
abs_mvd_l1[vertex][0][x0][y0]	ae(v)
if(abs_mvd_l1[vertex][0][x0][y0])	
mvd_l1_sign_flag[vertex][0][x0][y0]	ae(v)
abs_mvd_l1[vertex][1][x0][y0]	ae(v)
if(abs_mvd_l1[vertex][1][x0][y0])	
mvd_l1_sign_flag[vertex][1][x0][y0]	ae(v)
}	
}	
}	
}	
else {	
if(inter_pred_idc[x0][y0] = = PRED_BI)	
bi_pred_idx[x0][y0]	ae(v)
if(inter_pred_idc[x0][y0] != PRED_L1) {	
if(num_ref_idx_active_minus1[0] > 0 && bi_pred_idx[x0][y0] = = 0)	
ref_idx_l0[x0][y0]	ae(v)
if(bi_pred_idx[x0][y0] != 1) {	
abs_mvd_l0[x0][y0][0]	ae(v)
if(abs_mvd_l0[x0][y0][0])	
mvd_l0_sign_flag[x0][y0][0]	ae(v)
abs_mvd_l0[x0][y0][1]	ae(v)
if(abs_mvd_l0[x0][y0][1])	
mvd_l0_sign_flag[x0][y0][1]	ae(v)
}	
}	
if(inter_pred_idc[x0][y0] != PRED_L0) {	
if(num_ref_idx_active_minus1[1] > 0 && bi_pred_idx[x0][y0] = = 0)	
ref_idx_l1[x0][y0]	ae(v)
if(bi_pred_idx[x0][y0] != 2) {	
abs_mvd_l1[x0][y0][0]	ae(v)
if(abs_mvd_l1[x0][y0][0])	
mvd_l1_sign_flag[x0][y0][0]	ae(v)
abs_mvd_l1[x0][y0][1]	ae(v)
if(abs_mvd_l1[x0][y0][1])	
mvd_l1_sign_flag[x0][y0][1]	ae(v)
}	

}	
}	
}	
}	
}	
}	
if(CuPredMode[x0][y0] != MODE_INTRA && merge_mode_flag[x0][y0] = = 0)	
cbf_all	ae(v)
if(cbf_all) {	
isSplit = log2CbWidth > 6 | | log2CbHeight > 6	
log2TbWidth = log2CbWidth > 6 ? 6 : log2CbWidth	
log2TbHeight = log2CbHeight > 6 ? 6 : log2CbHeight	
transform_unit(x0, y0, log2TbWidth, log2TbHeight, isSplit)	
if(log2CbWidth > 6)	
transform_unit(x0 + (1 << log2TbWidth), y0, log2TbWidth, log2TbHeight, isSplit)	
if(log2CbHeight > 6)	
transform_unit(x0, y0 + (1 << log2TbHeight), log2TbWidth, log2TbHeight, isSplit)	
if(log2CbWidth > 6 && log2CbHeight > 6)	
transform_unit(x0 + (1 << logTbWidth), y0 + (1 << log2TbHeight), log2TbWidth, log2TbHeight, isSplit)	
}	
}	
}	

7.3.6.5 Transform unit syntax

transform_unit(x0, y0, log2TbWidth, log2TbHeight, isSplit) {	**Descriptor**
cbf_cb	ae(v)
cbf_cr	ae(v)
if(isSplit | | CuPredMode[x0][y0] != MODE_INTRA | | cbf_cb | | cbf_cr)	
cbf_luma	ae(v)
if(cu_qp_delta_enabled_flag && ((!sps_dquant_flag | | cuQpDeltaCode = = 1) && (cbf_luma | | cbf_cb | | cbf_cr)) | | cuQpDeltaCode = = 2))	
cu_qp_delta	ae(v)
if(sps_adcc_flag = = 0) {	
if(cbf_luma)	
residual_coding_baseline(x0, y0, log2TbWidth, log2TbHeight, 0)	
if(cbf_cb)	
residual_coding_baseline (x0, y0, log2TbWidth – 1, log2TbHeight – 1, 1)	
if(cbf_cr)	
residual_coding_baseline (x0, y0, log2TbWidth – 1, log2TbHeight – 1, 2)	
}	
else{	

if(CuPredMode[x0][y0] = = MODE_INTRA && sps_ats_flag && cIdx = = 0 log2CbWidth <= 5 && log2CbHeight <= 5 && cbf_luma)	
ats_cu_intra_flag[x0][y0]	ae(v)
if(ats_cu_intra_flag[x0][y0] = = 1) {	
ats_hor_mode[x0][y0]	ae(v)
ats_ver_mode[x0][y0]	ae(v)
}	
if(CuPredMode[x0][y0] != MODE_INTRA && sps_ats_flag) {	
if((allowAtsInterVerHalf | | allowAtsInterVerQuad | | allowAtsInterHorHalf | | allowAtsInterHorQuad) && (cbf_cb | | cbf_cr | | cbf_luma)) {	
ats_cu_inter_flag[x0][y0]	ae(v)
}	
if(ats_cu_inter_flag[x0][y0]) {	
if((allowAtsInterVerHalf | | allowAtsInterHorHalf) && (allowAtsInterVerQuad | | allowAtsInterHorQuad))	
ats_cu_inter_quad_flag[x0][y0]	ae(v)
if((ats_cu_inter_quad_flag[x0][y0] && allowAtsInterVerQuad && allowAtsInterHorQuad) | | (!ats_cu_inter_quad_flag[x0][y0] && allowAtsInterVerHalf && allowAtsInterHorHalf)) {	
ats_cu_inter_horizontal_flag[x0][y0]	ae(v)
ats_cu_inter_pos_flag[x0][y0]	ae(v)
}	
}	
if(ats_cu_inter_flag[x0][y0]) {	
if(ats_cu_inter_horizontal_flag[x0][y0]) {	
TrafoX0 = (ats_cu_inter_pos_flag[x0][y0] ? 0 : ((1 << log2TbWidth) – (1 << TrafoLog2Width)))	
TrafoY0 = 0	
TrafoLog2Width = log2TbWidth – (ats_cu_inter_quad_flag[x0][y0] ? 2 : 1)	
TrafoLog2Height = log2TbHeight	
} else {	
TrafoX0 = 0	
TrafoY0 = (ats_cu_inter_pos_flag[x0][y0] ? 0 : ((1 << log2TbHeight) – (1 << TrafoLog2Height)))	
TrafoLog2Width = log2TbWidth	
TrafoLog2Height = log2TbHeight – (ats_cu_inter_quad_flag[x0][y0] ? 2 : 1)	
}	
} else {	
TrafoX0 = 0	
TrafoY0 = 0	
TrafoLog2Width = log2TbWidth	
TrafoLog2Height = log2TbHeight	
}	
if(cbf_luma)	
residual_coding_main(x0 + TrafoX0, y0 + TrafoY0, TrafoLog2Width, TrafoLog2Height, 0)	
if(cbf_cb)	
residual_coding_main(x0 + TrafoX0, y0 +TrafoY0, TrafoLog2Width– 1, TrafoLog2Height– 1, 1)	
if(cbf_cr)	

residual_coding_main(x0 + TrafoX0, y0 + TrafoY0, TrafoLog2Width– 1, TrafoLog2Height– 1, 2)	
}	
}	

7.3.6.6 Run-length residual coding syntax

residual_coding_baseline(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	**Descriptor**
for(yC = 0; yC < 1 << (log2TbHeight); yC++)	
for(xC = 0; xC < 1 << (logTbWidth); xC++)	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = 0	
ScanPos = 0	
do {	
coeff_zero_run	ae(v)
ScanPos += coeff_zero_run	
coeff_abs_level_minus1	ae(v)
coeff_sign_flag	ae(v)
Level = coeff_abs_level_minus1 + * (1 – 2 * coeff_sign_flag)	
xC = ZigZagScanOrder[log2TbWidth][log2TbHeight][ScanPos][0]	
yC = ZigZagScanOrder[log2TbWidth][log2TbHeight][ScanPos][1]	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = Level	
if(ScanPos < (1 << (log2TbWidth + log2TbHeight) – 1)	
coeff_last_flag	ae(v)
} while(coeff_last_flag = = 0)	
}	

7.3.6.7 Enchanced residual coding syntax

residual_coding_main (x0, y0, log2TbWidth, log2TbHeight, cIdx) {	**Descriptor**
last_sig_coeff_x_prefix	ae(v)
last_sig_coeff_y_prefix	ae(v)
if(last_sig_coeff_x_prefix > 3)	
last_sig_coeff_x_suffix	ae(v)
if(last_sig_coeff_y_prefix > 3)	
last_sig_coeff_y_suffix	ae(v)
rasterPosLast = lastX + lastY * (1 << log2TbWidth)	
scanPosLast = invScanOrder[log2TbWidth – 1][log2TbHeight – 1][rasterPosLast]	
lastCoefGroup = scanPosLast >> 4	
iPos = scanPosLast	
for(cgIdx = lastCoefGroup; cgIdx >= 0; cgIdx − −) {	
escapeDataPresent = 0	
numNZ = 0	
subBlockPos = cgIdx << 4	
for(; iPos >= subBlockPos; iPos − −) {	

blkPos = ScanOrder[log2TbWidth – 1][log2TbHeight – 1][iPos]	
xC = blkPos & (width -1)	
yC = blkPos >> log2TbWidth	
if(iPos ! = scanPosLast)	
sig_coeff_flag[xC][yC]	ae(v)
else {	
sig_coeff_flag[xC][yC] = 1	
}	
coef[blkPos] = sig_coeff_flag[xC][yC]	
if(coef[blkPos]) {	
blkPosArray[numNZ] = iPos	
numNZ + +	
}	
}	
if(numNZ > 0) {	
lastGreaterAcanPos = −1	
numC1Flag = min(numNZ, 8)	
for(n = 0; n < numC1Flag; n+ +) {	
blkPos = blkPosArray[n]	
coeff_abs_level_greaterA_flag[n]	ae(v)
coef[blkPos] + = coeff_abs_level_greaterA_flag[n]	
if(coeff_abs_level_greaterA_flag[n])	
if(lastGreaterAScanPos ! = −1)	
lastGreaterAScanPos = n	
else	
escapeDataPresent = 1	
}	
if(lastGreaterAScanPos != −1) {	
blkPos = blkPosArray[lastGreaterAScanPos]	
coeff_abs_level_greaterB_flag[lastGreaterAScanPos]	ae(v)
coef[blkPos] + = coeff_abs_level_greaterB_flag[lastGreaterAScanPos]	
if(coeff_abs_level_greaterB_flag[lastGreaterAScanPos])	
escapeDataPresent = 1	
}	
escapeDataPresent = escapeDataPresent || (numNZ > 8)	
countFirstBCoef = 1	
if(escapeDataPresent) {	
for(n = 0; n < numNZ; n+ +) {	
blkPos = blkPosArray[n]	
baseLevel = (n < 8) ? (2 + countFirstBCoef) : 1	
if(coef[blkPos] >= baseLevel) {	
coeff_abs_level_remaining[n]	ae(v)
coef[blkPos] = baseLevel + coeff_abs_level_remaining	
}	
if(coef[blkPos] >= 2) {	
countFirstBCoef = 0	
}	
}	

}	
coeff_signs_group	ae(v)
coeff_signs_group <<= 32 – numNZ	
for(n = 0; n < numNZ; n+ +) {	
blkPos = blkPosArray[n]	
sign = coeff_signs_group >> 31	
coeff_signs_group <<= 1	
coef[blkPos] = sign > 0 ? – coef[blkPos] : coef[blkPos]	
}	
}	
}	
}	

7.4 Semantics

7.4.1 General

The 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 is specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

7.4.2 NAL unit semantics

7.4.2.1 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 B for the byte stream format. Other methods of demarcation may be specified outside of this document.

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 is preceded by a NAL unit length field in the byte stream format specified in Annex B.

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 a 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.

7.4.2.2 NAL unit header semantics

forbidden_zero_bit shall be equal to 0.

nal_unit_type_plus1 minus 1 specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7-1. The variable NalUnitType is derived as follows:

$$NalUnitType = nal\_unit\_type\_plus1 - 1 \quad (7\text{-}1)$$

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

NalUnitType	Name of NalUnitType	Content of NAL unit and RBSP syntax structure	NAL unit type class
0	NONIDR_NUT	Coded slice of a non-IDR picture slice_layer_rbsp( )	VCL
1	IDR_NUT	Coded slice of an IDR picture slice_layer_rbsp( )	VCL
2-23	RSV_VCL_NUT02.. RSV_VCL_NUT23	Reserved VCL NAL Units	VCL
24	SPS_NUT	SPS seq_parameter_set_rbsp( )	non-VCL
25	PPS_NUT	PPS pic_parameter_set_rbsp( )	non-VCL
26	APS_NUT	Adaptation parameter set adaptation_parameter_set_rbsp( )	non-VCL
27	FD_NUT	Filler data filler_data_rbsp( )	non-VCL
28	SEI_NUT	Supplemental enhancement information sei_rbsp( )	non-VCL
29-55	RSV_NONVCL29.. RSV_NONVCL55	Reserved	non-VCL
56-62	UNSPEC_NUT56.. UNSPEC_NUT62	Unspecified	non-VCL

nuh_temporal_id specifies a temporal identifier for the NAL unit.

The variable TemporalId is derived as follows:

$$\text{TemporalId} = \text{nuh\_temporal\_id} \tag{7-2}$$

When NalUnitType is equal to IDR_NUT, the coded slice belongs to an IDR 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 NalUnitType 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, 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 NalUnitType is equal to PPS_NUT, TemporalId may be greater than or equal to the TemporalId of the containing access unit, as all PPSs (PPSs) may be included at the beginning of a bitstream, wherein the first coded picture has TemporalId equal to 0. When NalUnitType is equal to 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_5bits shall be equal to 0 in bitstreams conforming to this version of this document. Values of nuh_reserved_zero_5bits greater than 0 are reserved for future use by ISO/IEC. Decoders conforming to a profile specified in Annex A shall ignore (i.e., remove from the bitstream and discard) all NAL units with values of nuh_reserved_zero_5bits greater than 0.

nuh_extension_flag shall be equal to 0 in bitstreams conforming to this version of this document. Value of nuh_extesion_flag equal to 1 is reserved for future use by ISO/IEC. Decoders conforming to a profile specified in Annex A shall ignore (i.e., remove from the bitstream and discard) all NAL units with values of nuh_extension_flag equal to 1.

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

7.4.2.3.1 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, E.2 and specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

7.4.2.3.2 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 7.4.2.3.3.

The first access unit of a CVS is an IRAP access unit.

7.4.2.3.3 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 9.

An access unit consists of one coded picture, zero or more VCL NAL units and zero or more non-VCL NAL units.

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

Let firstVclNalUnitInAu be a VCL NAL unit that is the first VCL NAL unit of a coded picture and for which the derived PicOrderCntVal differs from the PicOrderCntVal of the previous coded picture. The first of any of the following NAL units preceding firstVclNalUnitInAu and succeeding the last VCL NAL unit preceding firstVclNalUnitInAu, if any, specifies the start of a new access unit:

– SPS NAL unit (when present),

– PPS NAL unit (when present),

– APS NAL unit (when present),

– SEI NAL unit (when present),

– NAL units with NalUnitType in the range of RSV_VCL_NUT02 to RSV_VCL_NUT23, inclusive (when present),

– NAL units with NalUnitType in the range of UNSPEC_NUT (when present).

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

When there is none of the above NAL units preceding firstVclNalUnitInAu and succeeding the last VCL NAL preceding firstVclNalUnitInAu, if any, firstVclNalUnitInAu 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 any SPS NAL units, PPS NAL units, APS NAL units, SEI NAL units, NAL units with NalUnitType in the range of RSV_VCL_NUT02 to RSV_VCL_NUT23, inclusive, or NAL units with NalUnitType in the range of UNSPEC_NUT56 to UNSPEC_NUT62, inclusive, are present in an access unit, they shall not follow the last VCL NAL unit of the access unit.

– NAL units having NalUnitType equal to SEI_NUT in an access unit shall not precede the first VCL NAL unit of the access unit.

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

7.4.3.1 SPS RBSP semantics

sps_seq_parameter_set_id provides an identifier for the SPS for reference by other syntax elements. The value of sps_seq_parameter_set_id shall be in the range of 0 to 15, inclusive.

profile_idc indicates a profile to which the CVS conforms as specified in Annex A. Bitstreams shall not contain values of profile_idc other than those specified in Annex A. Other values of profile_idc are reserved for future use by ISO/IEC.

level_idc indicates a level to which the CVS conforms as specified in Annex A. Bitstreams shall not contain values of level_idc other than those specified in Annex A. Other values of level_idc are reserved for future use by ISO/IEC.

NOTE 1 – A greater value of level_idc indicates a higher level.

NOTE 2 – When the coded video sequence conforms to multiple profiles, 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 document).

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 2, inclusive.

Depending on the value of chroma_format_idc, the value of the variable ChromaArrayType is assigned as follows:

– If chroma_format_idc is equal to 0, ChromaArrayType is set equal to 0.

– Otherwise, ChromaArrayType is set equal to chroma_format_idc.

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 $BitDepth_Y$ and the value of the luma quantization parameter range offset $QpBdOffset_Y$ as follows:

$$BitDepth_Y = 8 + bit\_depth\_luma\_minus8 \quad (7\text{-}3)$$

$$QpBdOffset_Y = 6 * bit\_depth\_luma\_minus8 \quad (7\text{-}4)$$

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 $BitDepth_C$ and the value of the chroma quantization parameter range offset $QpBdOffset_C$ as follows:

$$BitDepth_C = 8 + bit\_depth\_chroma\_minus8 \quad (7\text{-}5)$$

$$QpBdOffset_C = 6 * bit\_depth\_chroma\_minus8 \quad (7\text{-}6)$$

bit_depth_chroma_minus8 shall be in the range of 0 to 8, inclusive.

log2_ctu_size_minus2 plus 2 specifies the luma coding tree block size of each CTU. When log2_ctu_size_minus2 is not present, the value of log2_ctu_size_minu2 is inferred to be equal to 4.

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

$$CtbLog2SizeY = log2\_ctu\_size\_minus2 + 2 \quad (7\text{-}7)$$

$$CtbSizeY = 1 << CtbLog2SizeY \quad (7\text{-}8)$$

$$MinCbLog2SizeY = 2 \quad (7\text{-}9)$$

$$MinCbSizeY = 1 << MinCbLog2SizeY \quad (7\text{-}10)$$

$$PicWidthInCtbsY = Ceil(pic\_width\_in\_luma\_samples \div CtbSizeY) \quad (7\text{-}11)$$

$$PicHeightInCtbsY = Ceil(pic\_height\_in\_luma\_samples \div CtbSizeY) \quad (7\text{-}12)$$

$$PicSizeInCtbsY = PicWidthInCtbsY * PicHeightInCtbsY \quad (7\text{-}13)$$

$$PicWidthInMinCbsY = pic\_width\_in\_luma\_samples / MinCbSizeY \quad (7\text{-}14)$$

$$PicHeightInMinCbsY = pic\_height\_in\_luma\_samples / MinCbSizeY \quad (7\text{-}15)$$

PicSizeInMinCbsY = PicWidthInMinCbsY * PicHeightInMinCbsY (7-16)

PicSizeInSamplesY = pic_width_in_luma_samples * pic_height_in_luma_samples (7-17)

PicWidthInSamplesC = pic_width_in_luma_samples / SubWidthC (7-18)

PicHeightInSamplesC = pic_height_in_luma_samples / SubHeightC (7-19)

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), CtbWidthC and CtbHeightC are both equal to 0.

– Otherwise, CtbWidthC and CtbHeightC are derived as follows:

CtbWidthC = CtbSizeY / SubWidthC (7-20)

CtbHeightC = CtbSizeY / SubHeightC (7-21)

sps_btt_flag equal to 1 specifies the binary and ternary splits are used. sps_btt_flag equal to 0 specifies the binary and ternary splits are not used and explicit quad split is only used.

log2_diff_ctu_max_11_cb_size specifies the difference between the luma coding tree block size and the maximum coding block size in which its width and height ratio is equal to 1:1. When log2_diff_ctu_max_11_cb_size is not present, it is inferred to be equal to 0.

log2_diff_max_11_min_11_cb_size specifies the difference between the maximum coding block size in which its width and height ratio is equal to 1:1 and the minimum coding block size in which its width and height ratio is equal to 1:1. When log2_diff_max_11_min_11_cb_size is not present, it is inferred to be equal to 0.

log2_diff_max_11_max_12_cb_size specifies the difference between the maximum coding block size in which its width and height ratio is equal to 1:1 and the maximum coding block size in which its width and height ratio is equal to 1:2 or 2:1. When log2_diff_max_11_max_12_cb_size is not present, it is inferred to be equal to 0.

log2_diff_min_11_min_12_cb_size_minus1 plus 1 specifies the difference between the minimum coding block size in which its width and height ratio is equal to 1:1 and the minimum coding block size in which its width and height ratio is equal to 1:2 or 2:1. When log2_diff_min_11_min_12_cb_size_minus1 is not present, it is inferred to be equal to 0.

log2_diff_max_12_max_14_cb_size_minus1 plus 1 specifies the difference between the maximum coding block size in which its width and height ratio is equal to 1:2 or 2:1 and the maximum coding block size in which its width and height ratio is equal to or greater than 1:4 or 4:1. When log2_diff_max_12_max_14_cb_size_minus1 is not present, it is inferred to be equal to 0.

log2_diff_min_12_min_14_cb_size_minus1 plus 1 specifies the difference between the maximum coding block size in which its width and height ratio is equal to 1:2 and 2:1 and the minimum coding block size in which its width and height ratio is equal to or greater than 1:4 or 4:1. When log2_diff_min_12_min_14_cb_size_minus1 is not present, it is inferred to be equal to 0.

log2_diff_max_11_max_tt_cb_size_minus1 plus 1 specifies the difference between the maximum coding block size in which its width and height ratio is equal to 1:1 and the maximum coding block size allowing ternary tree split. When log2_diff_max_11_max_tt_cb_size_minus1 is not present, it is inferred to be equal to 0.

log2_diff_min_11_min_tt_cb_size_minus2 plus 2 specifies the difference between the minimum coding block size in which its width and height ratio is equal to 1:1 and the minimum coding block size allowing ternary tree split. When log2_diff_min_11_min_tt_cb_size_minus2 is not present, it is inferred to be equal to 0.

The variables MaxCbLog2Size11Ratio, MinCbLog2Size11Ratio, MaxCbLog2Size12Ratio, MinCbLog2Size12Ratio, MaxCbLog2Size14Ratio, MinCbLog2Size14Ratio, MaxTtLog2Size and MinTtLog2Size are derived as follows:

MaxCbLog2Size11Ratio = CtbLog2SizeY – log2_diff_ctu_max_11_cb_size (7-22)

MinCbLog2Size11Ratio = Max(MaxCbLog2Size11Ratio – log2_diff_max_11_min_11_cb_size, MinCbLog2SizeY) (7-23)

MaxCbLog2Size12Ratio = MaxCbLog2Size11Ratio – log2_diff_max_11_max_12_cb_size (7-24)

MinCbLog2Size12Ratio = Max(MinCbLog2Size11Ratio + log2_diff_min_11_min_12_cb_size_minus1 + 1, MinCbLog2SizeY) (7-25)

MaxCbLog2Size14Ratio = MaxCbLog2Size12Ratio + log2_diff_max_12_max_14_cb_size_minus1 + 1 (7-26)

MinCbLog2Size14Ratio = Max(MinCbLog2Size12Ratio + log2_diff_min_12_min_14_cb_size_minus1 + 1, MinCbLog2SizeY) (7-27)

MaxTtLog2Size = MaxCbLog2Size11Ratio + log2_diff_max_11_max_tt_cb_size_minus1 + 1 (7-28)

MinTtLog2Size = Max(MinCbLog2Size11Ratio + log2_diff_min_11_min_tt_cb_size_minus2 + 2, MinCbLog2SizeY) (7-29)

sps_suco_flag equal to 1 specifies the split unit coding ordering is used. sps_suco_flag equal to 0 the specifies split unit coding ordering is not used.

log2_diff_ctu_size_max_suco_cb_size specifies the difference between the luma coding tree block size and the maximum coding block size allowing split unit coding order. When log2_diff_ctu_size_max_suco_cb_size is not present, it is inferred to be equal to 0.

log2_diff_max_suco_min_suco_cb_size specifies the difference between the maximum coding block size allowing split unit coding order and the minimum coding block size allowing split unit coding order. When log2_diff_max_suco_min_suco_cb_size is not present, it is inferred to be equal to 0.

The variables MaxSucoLog2Size and MinSucoLog2Size are derived as follows:

MaxSucoLog2Size = Min (CtbLog2SizeY – log2_diff_ctu_size_max_suco_cb_size, 6) (7-30)

MinSucoLog2Size = Max(MaxSucoLog2Size – log2_diff_max_suco_min_suco_cb_size, MinCbLog2SizeY) (7-31)

sps_amvr_flag equal to 1 specifies adaptive motion vector resolution is used. sps_amvr_flag equal to 0 specifies that adaptive motion vector resolution is not used. When sps_amvr_flag is not present, it is inferred to be equal to 0.

sps_mmvd_flag equal to 1 specifies MMVD is used. sps_mmvd_flag equal to 0 specifies that MMVD is not used. When sps_mmvd_flag is not present, it is inferred to be equal to 0.

sps_affine_flag specifies whether the affine model based motion compensation can be used for inter prediction. If sps_affine_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 the coding unit syntax of the CVS. Otherwise (sps_affine_flag is equal to 1), affine model based motion compensation can be used in the CVS.

sps_dmvr_flag specifies whether decoder-side motion vector refinement can be used for inter motion vectors refinement. When sps_dmvr_flag is not presented, it is inferred to be equal to 0.

sps_alf_flag equal to 0 specifies that the adaptive loop filter is disabled. sps_alf_flag equal to 1 specifies that the adaptive loop filter is enabled. When sps_alf_flag is not presented, it is inferred to be equal to 0.

sps_addb_flag equal to 1 specifies enchanced deblocking filter is applied. sps_addb_flag equal to 0 specifies that the enchanced deblocking filter is not applied. When sps_addb_flag is not presented, it is inferred to be equal to 0.

sps_admvp_flag equal to 0 specifies that the advanced motion vector prediction is disabled. sps_admvp_flag equal to 1 specifies that the advanced motion vector prediction is enabled. When sps_admvp_flag is not presented, it is inferred to be equal to 0.

sps_eipd_flag equal to 1 specifies the extended intra prediction modes are used. sps_eipd_flag equal to 0 specifies the extended intra prediction modes are not used.

sps_amis_flag equal to 1 specifies the advanced motion signaling and interpolation are used. sps_amis_flag equal to 0 specifies the advanced motion signaling and interpolation are not used.

sps_ibc_enabled_flag equal to 1 specifies the intra block copy is enabled. sps_ibc_enabled_flag equal to 0 specifies the intra block copy is disabled. When sps_ibc_enabled_flag is not presented, it is inferred to be equal to 0.

max_ibc_cand_size_minus2 specifies the maximum block size of the intra block copy mode as follows:

log2MaxIbcCandSize = 2 + log2_max_ibc_cand_size_minus2 (7-32)

log2_max_ibc_cand_size_minus2 shall be in the range of 0 to 4, inclusive.

sps_iqt_flag equal to 1 specifies the improved quantization and transform are used. sps_iqt_flag equal to 0 specifies the improved quantization and transform are not used.

sps_htdf_flag equal to 1 specifies the hadamard transform domain filter is used. sps_htdf_flag equal to 0 specifies the hadamard transform domain filter is not used.

sps_cm_init_flag equal to 1 specifies the context modeling and initialization processes are used. sps_cm_init_flag equal to 0 specifies the context modeling and initialization processes are not used.

sps_adcc_flag equal to 1 specifies the enchanced residual coding is used. sps_adcc_flag equal to 0 specifies bo is used.

The array ScanOrder[sPos] specifies the mapping of the zig-zag scan position sPos, ranging from 0 to (1 << log2TbHeight) * (1 << log2TbWidth) − 1, inclusive to a raster scan pozition rPos. The array invScanOrder[rPos] specifies the mapping of the raster scan position rPos, ranging from 0 to (1 << log2TbHeight) * (1 << log2TbWidth) − 1, inclusive to a the zig-zag scan position sPos.

The array ScanOrder is derived by invoking clause 6.5.3 with input parameters blkWidth=(1<< log2TbWidth) and blkHeight = (1<< log2TbHeight) and the array invScanOrder is derived by invoking clause 6.5.4 with input parameters blkWidth=(1<< log2TbWidth) and blkHeight = (1<< log2TbHeight).

sps_ats_flag equal to 1 specifies that ats_cu_intra_flag and ats_cu_inter_flag may be present in the residual coding syntax of the CVS. sps_ats_flag equal to 0 specifies that ats_cu_intra_flag and ats_cu_inter_flag are not present in the residual coding syntax of the CVS. When not present, the value of sps_ats_flag is inferred to be equal to 0.

sps_rpl_flag equal to 1 specifies that syntax related to reference picture lists is present. sps_rpl_flag equal to 0 specifies that syntax related to reference picture lists is not present.

sps_pocs_flag equal to 1 specifies that syntax related to picture order count is present. sps_pocs_flag equal to 0 specifies that syntax related to picture order count is not present.

sps_dquant_flag equal to 1 specifies the improved delta qp signaling processes is used. sps_dquant_flag equal to 0 specifies the improved delta qp signaling process is not used.

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

$$MaxPicOrderCntLsb = 2^{(log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4)} \quad (7\text{-}33)$$

The value of log2_max_pic_order_cnt_lsb_minus4 shall be in the range of 0 to 12, inclusive. When not present, the value of log2_max_pic_order_cnt_lsb_minus4 is inferred to be equal to 0.

log2_sub_gop_length, when present, specifies the value of the variable SubGopLength that is used in the decoding process for picture order count as follows:

$$SubGopLength = 2^{(log2\_sub\_gop\_length)} \quad (7\text{-}34)$$

The value of log2_sub_gop_length shall be in the range of 0 to 5, inclusive. When not present, the value of log2_sub_gop_length is inferred to be equal to 0.

log2_ref_pic_gap_length, when present, specifies the value of the variable RefPicGapLength that is used in the decoding process for reference picture marking as follows:

$$RefPicGapLength = 2^{(log2\_ref\_pic\_gap\_length)} \quad (7\text{-}35)$$

The value of log2_ref_pic_gap_length shall be in the range of 0 to 5, inclusive. When not present, the value of log2_ref_pic_gap_length is inferred to be equal to 0.

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 in clause A.4. When not present, the value of sps_max_dec_pic_buffering_minus1 is inferred to be equal to SubGopLength + max_num_tid0_ref_pics – 1.

max_num_tid0_ref_pics, when sps_rpl_flag is equal to 0, specifies the maximum number of reference pictures in temporal layer 0 that may be used by the decoding process for inter prediction of any picture in the CVS. The value of

max_num_tid0_ref_pics is also used to determine the size of the Decoded Picture Buffer The value of max_num_tid0_ref_pics shall be in the range of 0 to 5. When not present, the value of max_num_tid0_ref_pics is inferred to be equal to 0.

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.

rpl_candidates_present_flag equal to 1 specifies that information about reference picture list candidates is present. rpl_candidates_present_flag equal to 0 specifies that information about reference picture list candidates is not present.

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, ltrpFlag) 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, ltrpFlag) is inferred to be equal to the value of corresponding syntax element in ref_pic_list_struct(0, rplsIdx, ltrpFlag) 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, ltrpFlag) 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. When not present, the value of num_ref_pic_lists_in_sps[i] is inferred to be equal to 0.

NOTE 3 – 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, ltrpFlag) syntax structures since there may be one ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure directly signalled in the slice headers of a current picture.

picture_cropping_flag equal to 1 specifies that the picture cropping offset parameters follow next in the SPS. picture_cropping_flag equal to 0 specifies that the picture cropping offset parameters are not present.

pic_crop_left_offset, **pic_crop_right_offset**, **pic_crop_top_offset** and **pic_crop_bottom_offset** specify the samples of pictures in the CVS that are output from the decoding process, in terms of a rectangular region specified in picture coordinates for output. When picture_cropping_flag is equal to 0, the value of pic_crop_left_offset, pic_crop_right_offset, pic_crop_top_offset and pic_crop_bottom_offset are inferred to be equal to 0.The conformance cropping window contains the luma samples with horizontal picture coordinates from SubWidthC * pic_crop_left_offset to pic_width_in_luma_samples − (SubWidthC * pic_crop_right_offset + 1) and vertical picture coordinates from SubHeightC * pic_crop_top_offset to pic_height_in_luma_samples − (SubHeightC * pic_crop_bottom_offset + 1), inclusive.

The value of SubWidthC * (pic_crop_left_offset + pic_crop_right_offset) shall be less than pic_width_in_luma_samples, and the value of SubHeightC * (pic_crop_top_offset + pic_crop_bottom_offset) shall be less than pic_height_in_luma_samples.

When ChromaArrayType is not equal to 0, the corresponding specified samples of the two chroma arrays are the samples having picture coordinates (x / SubWidthC, y / SubHeightC), where (x, y) are the picture coordinates of the specified luma samples.

vui_parameters_present_flag equal to 1 specifies that the vui_parameters( ) syntax structure as specified in Annex E is present. vui_parameters_present_flag equal to 0 specifies that the vui_parameters( ) syntax structure as specified in Annex E is not present.

7.4.3.2 PPS 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.

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 slices 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 slices 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.

additional_lt_poc_lsb_len specifies the value of the variable MaxLtPicOrderCntLsb that is used in the decoding process for reference picture lists as follows:

$$MaxLtPicOrderCntLsb = 2^{(log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 + additional\_lt\_poc\_lsb\_len)} \quad (7\text{-}36)$$

The value of additional_lt_poc_lsb_len shall be in the range of 0 to 32 − log2_max_pic_order_cnt_lsb_minus4 − 4, inclusive.

When not present, the value of additional_lt_poc_lsb_len is inferred to be equal to 0.

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 slice 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 slice headers.

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.

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 and adaptive loop filter operations. When not present, the value of loop_filter_across_tiles_enabled_flag is inferred to be equal to 1.

tile_offset_len_minus1 plus 1 specifies the length, in bits, of the entry_point_offset_minus1[i] syntax elements in the slice headers referring to the PPS. The value of tile_offset_len_minus1 shall be in the range of 0 to 31, inclusive.

tile_id_len_minus1 plus 1 specifies the number of bits used to represent the syntax element tile_id_val[i][j], when present, in the PPS, and the syntax element first_tile_id and last_tile_id in slice headers referring to the PPS. The value of tile_id_len_minus1 shall be in the range of Ceil(Log2(NumTilesInPic) to 15, inclusive.

explicit_tile_id_flag equal to 1 specifies that tile ID for each tile is explicitly signalled. explicit_tile_id_flag equal to 0 specifies that tile IDs are not explicitly signalled.

tile_id_val[i][j] specifies the tile ID of the tile of the i-th tile row and the j-th tile column. The length of tile_id_val[i][j] is tile_id_len_minus1 + 1 bits.

For any integer m in the range of 0 to num_tile_columns_minus1, inclusive, and any integer n in the range of 0 to num_tile_rows_minus1, inclusive, tile_id_val[i][j] shall not be equal to tile_id_val[m][n] when i is not equal to m or j is not equal to n, and tile_id_val[i][j] shall be less than tile_id_val[m][n] when j * (num_tile_columns_minus1 + 1) + i is less than n * (num_tile_columns_minus1 + 1) + m.

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 PicSizeInCtbsY − 1, inclusive, specifying the conversion from a tile index to the number of CTUs in the tile,
– the set TileIdToIdx[tileId] for a set of NumTilesInPic tileId values specifying the conversion from a tile ID to a tile index and 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 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.

arbitrary_slice_present_flag equal to 1 specifies that the syntax element arbitrary_slice_flag is present in slice headers referring to the PPS. arbitrary_slice_present_flag equal to 0 specifies that the syntax element arbitrary_slice_flag is not present in slice headers referring to the PPS.

constrained_intra_pred_flag equal to 0 specifies that intra prediction allows usage of decoded samples of neighbouring coding blocks coded using either intra or inter prediction modes. constrained_intra_pred_flag equal to 1 specifies constrained intra prediction, in which case intra prediction only uses decoded samples from neighbouring coding blocks coded using intra prediction modes.

cu_qp_delta_enabled_flag equal to 1 specifies that the log2_cu_qp_delta_area_minus6 syntax element is present in the PPS and that cu_qp_delta may be present in the transform unit syntax. cu_qp_delta_enabled_flag equal to 0 specifies that that the log2_cu_qp_delta_area_minus6 syntax element is not present in the PPS and cu_qp_delta is not present in the transform unit syntax.

log2_cu_qp_delta_area_minus6 specifies the cuQpDeltaArea value of coding units that convey cu_qp_delta as follows:

$$cuQpDeltaArea = log2\_cu\_qp\_delta\_area\_minus6 + 6 \quad (7\text{-}37)$$

7.4.3.3 APS RBSP 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 slices 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.

7.4.3.4 Filler data RBSP semantics

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

ff_byte is a byte equal to 0xFF.

7.4.3.5 Supplemental enhancement information RBSP semantics

Supplemental enhancement information (SEI) contains information that is not necessary to decode the samples of coded pictures from VCL NAL units. An SEI RBSP contains one or more SEI messages.

7.4.3.6 Slice layer RBSP semantics

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

7.4.3.7 RBSP slice trailing bits semantics

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

7.4.3.8 RBSP trailing bits semantics

rbsp_stop_one_bit shall be equal to 1.

rbsp_alignment_zero_bit shall be equal to 0.

7.4.3.9 Byte alignment semantics

alignment_bit_equal_to_one shall be equal to 1.

alignment_bit_equal_to_zero shall be equal to 0.

7.4.4 Supplemental enhancement information message semantics

Each SEI message consists of the variables specifying the type payloadType and size payloadSize of the SEI message payload. SEI message payloads are specified in Annex D. The derived SEI message payload size payloadSize is specified in bytes and shall be equal to the number of RBSP bytes in the SEI message payload.

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

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

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

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

7.4.5 Slice header semantics

When present, the value of the slice header syntax elements slice_pic_parameter_set_id, no_output_of_prior_pics_flag, slice_pic_order_cnt_lsb, and ref_pic_flag shall be the same in all slice headers of a coded picture.

slice_pic_parameter_set_id specifies the value of pps_pic_parameter_set_id for the PPS in use. The value of slice_pic_parameter_set_id shall be in the range of 0 to 63, inclusive.

single_tile_in_slice_flag equal to 1 specifies that there is only one tile in the slice. single_tile_in_slice_flag equal to 0 specifies that there is more than one tile in the slice.

first_tile_id specifies the tile ID of the first tile of the slice. The length of first_tile_id is tile_id_len_minus1 + 1 bits bits. The value of first_tile_id shall not be equal to the value of first_tile_id of any other coded slice of the same coded picture.

When there is more than one slice in a picture, the decoding order of the slices in the picture shall be in increasing value of first_tile_id.

arbitrary_slice_flag equal to 1 specifies that the tiles in the slice may be any set of two or more tiles of the picture and the tile ID of each of the tiles is explicitly signalled in the slice header. arbitrary_slice_flag equal to 0 specifies the tiles in the slice are identified by the syntax elements first_tile_id and last_tile_id. When not present, the value of arbitrary_slice_flag is inferred to be equal to 0.

last_tile_id specifies the tile ID of the last tile of the slice. The length of last_tile_id is tile_id_len_minus1 + 1 bits. When not present, the value of last_tile_id is inferred to be equal to first_tile_id.

When arbitrary_slice_flag is equal to 0, the variables numTileRowsInSlice, numTileColumnsInSlice, and NumTilesInSlice are derived as follows:

```
firstTileIdx = TileIdToIdx[ first_tile_id ]
firstTileColumnIdx = firstTileIdx % ( num_tile_columns_minus1 + 1 )
```

```
lastTileIdx = TileIdToIdx[ last_tile_id ]
lastTileColumnIdx = lastTileIdx % ( num_tile_columns_minus1 + 1 )

deltaTileIdx = lastTileIdx − firstTileIdx
if( lastTileIdx < firstTileIdx ) {
    if( firstTileColumnIdx > lastTileColumnIdx )                                    (7-38)
        deltaTileIdx  +=  NumTilesInPic + num_tile_columns_minus1 + 1
    else
        deltaTileIdx  +=  NumTilesInPic
} else if( firstTileColumnIdx > lastTileColumnIdx )
    deltaTileIdx  +=  num_tile_columns_minus1 + 1
numTileRowsInSlice = ( deltaTileIdx / ( num_tile_columns_minus1 + 1 ) ) + 1
numTileColumnsInSlice = ( deltaTileIdx % ( num_tile_columns_minus1 + 1 ) ) + 1
NumTilesInSlice = numTileRowsInSlice * numTileColumnsInSlice
```

When arbitrary_slice_flag is equal to 0, the variable SliceTileIdx[i] specifies the tile index of the i-th tile in the slice and is derived as follows:

```
tileIdx = TileIdToIdx[ first_tile_id ]
for( j = 0, cIdx = 0; j < numTileRowsInSlice; j++, tileIdx  +=  num_tile_columns_minus1 + 1 ) {
  tileIdx = tileIdx % NumTilesInPic
  for( i = 0, currTileIdx = tileIdx; i < numTileColumnsInSlice; i++, currTileIdx++, cIdx++ ) {   (7-39)
      if( currTileIdx / ( num_tile_columns_minus1 + 1 ) > tileIdx / ( num_tile_columns_minus1 + 1 ) )
          SliceTileIdx[ cIdx ] = currTileIdx − ( num_tile_columns_minus1 + 1 )
      else
          SliceTileIdx[ cIdx ] = currTileIdx
  }
}
```

num_remaining_tiles_in_slice_minus1 specifies the number of tiles in the slice excluding the first tile of the slice. The value of num_remaining_tiles in_slice_minus1 shall be in the range of 0 to NumTilesInPic − 1, inclusive.

When arbitrary_slice_flag is equal to 1, the variable NumTilesInSlice is derived as follows:

```
NumTilesInSlice = num_remaining_tiles_in_slice_minus1 + 2                           (7-40)
```

delta_tile_id_minus1[i] specifies the difference between the tile ID of the (i + 1)-th tile and the tile ID of i-th tile in the slice. The value of delta_tile_id_minus1[i] shall be in the range of 0 to $2^{(tile\_id\_len\_minus1 + 1)} - 1$, inclusive.

When arbitrary_slice_flag is equal to 1, the variable SliceTileIdx[i] for i in the range 0 to NumTilesInSlice – 1, inclusive, is derived as follows:

```
sliceTileId[ i ] = i > 0 ? ( sliceTileId[ i − 1 ] + delta_tile_id_minus1[ i − 1 ] + 1 ) : first_tile_id
SliceTileIdx[ i ] = TileIdToIdx[ sliceTileId[ i ] ]                                 (7-41)
```

slice_type specifies the coding type of the slice according to Table 7-2.

Table 7-2 – Name association to slice_type

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

When NalUnitType is equal to IDR_NUT, i.e., the picture is an IDR picture, slice_type shall be equal to 2.

no_output_of_prior_pics_flag affects the output of previously-decoded pictures in the decoded picture buffer after the decoding of an IDR picture that is not the first picture in the bitstream as specified in Annex C.

mmvd_group_enable_flag equal to 1 specifies that the syntax element mmvd_group_idx is present. mmvd_group_enable_flag equal to 0 specifies that the syntax element mmvd_group_idx is not present. When not present, the value of mmvd_group_enable_flag is inferred to be equal to 0.

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

slice_aps_id specifies the adaptation_parameter_set_id of the APS that the slice refers to. The TemporalId of the APS NAL unit having adaptation_parameter_set_id equal to slice_aps_id shall be less than or equal to the TemporalId of the coded slice NAL unit.

When multiple APSs with the same value of adaptation_parameter_set_id are referred to by two or more slices of the same picture, the multiple APSs with the same value of adaptation_parameter_set_id shall have the same content.

slice_alf_map_signalled equal to 1 specifies that adaptive loop filter applicability map for Y component is signalled. slice_alf_enabled_flag equal to 0 specifies that adaptive loop filter applicability map for Y is not signalled. When not present, the value of slice_alf_map_signalled is inferred to be equal to 0.

slice_pic_order_cnt_lsb, when present, specifies the picture order count modulo MaxPicOrderCntLsb for the current picture. The length of the slice_pic_order_cnt_lsb syntax element is log2_max_pic_order_cnt_lsb_minus4 + 4 bits. When not present, the value of slice_pic_order_cnt_lsb is inferred to be equal to 0.

ref_pic_list_sps_flag[i] equal to 1 specifies that reference picture list i of the current picture is derived based on one of the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) 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 picture is derived based on the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure with listIdx equal to i that is directly included in the slice 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] shall be equal to 0. When rpl1_idx_present_flag is equal to 0 and ref_pic_list_sps_flag[0] is present, the value of ref_pic_list_sps_flag[1] is inferred to be equal to ref_pic_list_sps_flag[0]. When not present, the value of rep_pic_list_sps_flag[i] is inferred to be equal to 0.

ref_pic_list_idx[i] specifies the index, into the list of the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structures with listIdx equal to i included in the active SPS, of the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) 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 rpl1_idx_present_flag is equal to 0 and ref_pic_list_sps_flag[0] is present, the value of ref_pic_list_idx[1] is inferred to be equal to ref_pic_list_idx[0].

The variable SliceRplsIdx[i] is derived as follows:

$$\text{SliceRplsIdx}[i] = \text{ref\_pic\_list\_sps\_flag}[i] \; ? \; \text{ref\_pic\_list\_idx}[i] : \text{num\_ref\_pic\_lists\_in\_sps}[i] \quad (7\text{-}42)$$

additional_poc_lsb_present_flag[i][j] equal to 1 specifies that additional_poc_lsb_val[i][j] is present. additional_poc_lsb_present_flag[i][j] equal to 0 specifies that additional_poc_lsb_val[i][j] is not present.

additional_poc_lsb_val[i][j] specifies the value of FullPocLsbLt[i][j] as follows:

$$\text{FullPocLsbLt}[i][\text{SliceRplsIdx}[i]][j] = \text{additional\_poc\_lsb\_val}[i][j] * \text{MaxPicOrderCntLsb} + \text{poc\_lsb\_lt}[i][\text{SliceRplsIdx}[i]][j] \quad (7\text{-}43)$$

The syntax element additional_poc_lsb_val[i][j] is represented by additional_lt_poc_lsb_len bits. When not present, the value of additional_poc_lsb_val[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 slices and that the syntax element num_ref_idx_active_minus1[1] is present for B slices. 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], when present, specifies the value of the variable NumRefIdxActive[i] as follows:

$$\text{NumRefIdxActive}[i] = \text{num\_ref\_idx\_active\_minus1}[i] + 1 \quad (7\text{-}44)$$

The value of num_ref_idx_active_minus1[i] shall be in the range of 0 to 14, inclusive.

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

For i equal to 0 or 1, when the current slice is a B slice and num_ref_idx_active_override_flag is equal to 0, NumRefIdxActive[i] is inferred to be equal to num_ref_idx_default_active_minus1[i] + 1.

When the current slice is a P slice and num_ref_idx_active_override_flag is equal to 0, NumRefIdxActive[0] is inferred to be equal to num_ref_idx_default_active_minus1[0] + 1.

When the current slice is a P slice, NumRefIdxActive[1] is inferred to be equal to 0.

When the current slice is an I slice, both NumRefIdxActive[0] and NumRefIdxActive[1] are inferred to be equal to 0.

temporal_mvp_asigned_flag specifies whether temporal motion vector predictors derivation process is to be configured with additional parameters signaled in the bitstream. When not present, the value of temporal_mvp_asigned_flag is inferred to be equal to 0.

col_pic_list_idx specifies the reference picture list for the derivation of collocated picture for purposes of temporal motion vector prediction. When col_pic_list_idx is not present, and slice_type is equal to P the col_pic_list_idx is inferred to be equal to 0. When col_pic_list_idx is not present, and slice_type is equal to B the col_pic_list_idx is inferred to be equal to 1.

col_pic_ref_idx specifies the reference index of the collocated picture used for temporal motion vector prediction. collocated_ref_idx refers to a picture in RefPicList[col_pic_list_idx], and the value of collocated_ref_idx shall be in the range of 0 to num_ref_idx_active_minus1[col_pic_list_idx], inclusive. When col_pic_ref_idx is not present, col_pic_ref_idx is inferred to be equal to 0.

col_source_mvp_list_idx specifies the reference picture list for the derivation of motion vector candidate for purposes of temporal motion vector prediction. When col_source_mvp_list_idx is not present, col_source_mvp_list_idx is inferred to be equal to 0.

slice_deblocking_filter_flag equal to 1 specifies that the operation of the deblocking filter is applied for the current slice. slice_deblocking_filter_flag equal to 0 specifies that the operation of the deblocking filter is not applied for the current slice.

slice_alpha_offset specifies the offset used in accessing the α and tC0 deblocking filter tables for deblocking filtering operations the slice. From this value, the offset that shall be applied when addressing these tables shall be computed as:

$$FilterOffsetA = slice\_alpha\_offset \tag{7-45}$$

The value of slice_alpha_offset shall be in the range of -12 to +12, inclusive. When slice_alpha_offset is not present in the slice header, the value of slice_alpha_offset shall be inferred to be equal to 0.

slice_beta_offset specifies the offset used in accessing the β deblocking filter table for deblocking filtering operations within the slice. From this value, the offset that is applied when addressing the β table of the deblocking filter shall be computed as:

$$FilterOffsetB = slice\_beta\_offset \tag{7-46}$$

The value of slice_beta_offset shall be in the range of -12 to +12, inclusive. When slice_beta_offset is not present in the slice header the value of slice_beta_offset shall be inferred to be equal to 0.

slice_qp specifies the luma quantization parameter value, QPy to be used for the coding blocks in the slice. The value of slice_qp shall be in the range of 0 to 51, inclusive.

slice_cb_qp_offset and **slice_cr_qp_offset** specify the offsets to the luma quantization parameter QPy used for deriving QPcb and QPcr, respectively. The values of slice_cb_qp_offset and slice_cr_qp_offset shall be in the range of −12 to +12, inclusive.

entry_point_offset_minus1[i] plus 1 specifies the i-th entry point offset in bytes, and is represented by tile_offset_len_minus1 plus 1 bits. The slice data that follow the slice header consists of NumTilesInSlice subsets, with subset index values ranging from 0 to NumTilesInSlice − 1, inclusive. The first byte of the slice data is considered byte 0. When present, emulation prevention bytes that appear in the slice data portion of the coded slice NAL unit are counted as part of the slice data for purposes of subset identification. Subset 0 consists of bytes 0 to entry_point_offset_minus1[0], inclusive, of the coded slice data, subset k, with k in the range of 1 to NumTilesInSlice − 2, inclusive, consists of bytes firstByte[k] to lastByte[k], inclusive, of the coded slice data with firstByte[k] and lastByte[k] defined as:

$$firstByte[\ k\ ] = \sum_{n=1}^{k} (entry\_point\_offset\_minus\ 1[n-1]+1) \tag{7-47}$$

lastByte[k] = firstByte[k] + entry_point_offset_minus1[k] (7-48)

The last subset (with subset index equal to NumTilesInSlice − 1) consists of the remaining bytes of the coded slice data.

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

7.4.5.1 Adaptive loop filter data semantics

alf_luma_filter_signal_flag equal to 1 specifies that a luma filter set is signalled. alf_luma_filter_signal_flag equal to 0 specifies that a luma filter set is not signalled.

alf_chroma_idc equal to 0 specifies that the chroma adaptive loop filter set is not is signalled and not applied to Cb and Cr colour components. alf_chroma_idc larger than 0 indicates that the chroma adaptive loop filter set is signalled. 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 for luma adaptive loop filter set and set equal to 1 for chroma adaptive loop filter set.

alf_luma_num_filters_signalled_minus1 plus 1 specifies the number of adaptive loop filter classes for which luma coefficients can be signalled. The value of alf_luma_num_filters_signalled_minus1 shall be in the range of 0 to NumAlfFilters − 1, inclusive.

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

alf_luma_type_flag specifies a type luma filter. The value of **alf_luma_type_flag** shall be in a range of 0 to 1, inclusive. When alf_luma_type_flag is equal to 0, the value max_golomb_idx is set equal to 2 and the variable NumAlfCoefs is set equal to 7, when alf_luma_type_flag is equal to 1, the value max_golomb_idx is set equal to 3 and the variable NumAlfCoefs is set equal to 13.

alf_luma_coeff_delta_idx[i] specifies the indices of the signalled adaptive loop filter luma coefficient deltas for the filter class indicated by i ranging from 0 to NumAlfFilters − 1. When alf_luma_coeff_delta_idx[i] is not present, it is inferred to be equal to 0.

alf_luma_fixed_filter_usage_pattern equal to 0 specifies that new filters do not use fixed filter and all entries alf_luma_fixed_filter_usage[i] are set equal to 0. alf_luma_fixed_filter_usage_pattern equal to 1 specifies all new filters use fixed filter and all entries of alf_luma_fixed_filter_usage[i] are set equal to 1. alf_luma_fixed_filter_usage_pattern equal to 2 specifies that usage of fixed filter is defined by signalled value of alf_luma_fixed_filter_usage[i] syntax elements. The value i shall be in the range from 0 to NumAlfFilters-1, inclusive.

alf_luma_fixed_filter_usage[i] equal to 1 specifies ith filter uses fixed filter. alf_luma_fixed_filter_usage[i] equal to 0 specifies ith filter does not use fixed filter. When it is not present, alf_luma_fixed_filter_usage[i] is inferred to be equal to 0 if alf_luma_fixed_filter_usage_pattern is equal to 0 and alf_luma_fixed_filter_usage[i] is inferred to be equal to 1 if alf_luma_fixed_filter_usage_pattern is equal to 1.

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[k][j] is derived as follows:

expGoOrderY[i] = alf_luma_min_eg_order_minus1 + 1+ alf_luma_eg_order_increase_flag[i] (7-49)

alf_luma_coeff_flag[i] equal 1 specifies that the coefficients of the luma filter indicated by i are signalled. alf_luma_coeff_flag[i] equal to 0 specifies that all filter coefficients of the luma filter indicated by i are set equal to 0. When not present, alf_luma_coeff_flag[i] is set equal to 1.

alf_luma_coeff_delta_abs[i][j] specifies the absolute value of the j-th coefficient delta of the signalled luma filter indicated by i. When alf_luma_coeff_delta_abs[i][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 } (7-50)

k = expGoOrderY[golombOrderIdxY[j]] (7-51)

alf_luma_coeff_delta_sign[i][j] specifies the sign of the j-th luma coefficient of the filter indicated by i as follows:

– If alf_luma_coeff_delta_sign[i][j] is equal to 0, the corresponding luma filter coefficient has a negative value.

– Otherwise (alf_luma_coeff_delta_sign[i][j] is equal to 1), the corresponding luma filter coefficient has a positive value.

When alf_luma_coeff_delta_sign[i][j] is not present, it is inferred to be equal to 1.

The variable filterCoefficients[i][j] with i = 0..alf_luma_num_filters_signalled_minus1, j = 0..NumAlfCoefs-2 is initialized as follows:

filterCoefficients[i][j] = alf_luma_coeff_delta_abs[i][j] * (1 − 2 * !alf_luma_coeff_delta_sign[i][j]) (7-52)

When alf_luma_coeff_delta_prediction_flag is equal 1, filterCoefficients[i][j] of luma adaptive loop filter set with i = 1..alf_luma_num_filters_signalled_minus1 and j = 0.. NumAlfCoefs-2 are modified as follows:

filterCoefficients[i][j] += filterCoefficients[i − 1][j] (7-53)

The luma filter coefficients $AlfCoeff_L$ with elements $AlfCoeff_L$[filtIdx][j], with filtIdx = 0..NumAlfFilters − 1 and j = 0.. NumAlfCoefs-2 are derived as follows

$AlfCoeff_L$[filtIdx][j] = filterCoefficients[alf_luma_coeff_delta_idx[filtIdx]][j] (7-54)

When alf_luma_fixed_filter_usage_pattern is greater then 0**,** and alf_luma_fixed_filter_usage[filtidx] is equal to 1 , the following applies:

AlfCoeffL [filtIdx][j] = AlfCoeffL [filtIdx][j] + fixedFilterCoeff[classToFilterMapping [filtidx] [alf_luma_fixed_filter_usage[filtidx] - 1]][j] (7-55)

with variables fixedFilterCoef and classToFilterMapping defined as follows:

```
fixedFilterCoeff[64][13] =
{
  { 0,   2,   7, -12,  -4, -11,  -2,  31,  -9,   6,  -4,  30, 444 - (1 << ( 10  - 1)) },
  { -26,   4,  17,  22,  -7,  19,  40,  47,  49, -28,  35,  48,  72 - (1 << ( 10  - 1)) },
  { -24,  -8,  30,  64, -13,  18,  18,  27,  80,   0,  31,  19,  28 - (1 << ( 10  - 1)) },
  { -4, -14,  44, 100,  -7,   6,  -4,   8,  90,  26,  26, -12,  -6 - (1 << ( 10  - 1)) },
  { -17,  -9,  23,  -3, -15,  20,  53,  48,  16, -25,  42,  66, 114 - (1 << ( 10  - 1)) },
  { -12,  -2,   1, -19,  -5,   8,  66,  80,  -2, -25,  20,  78, 136 - (1 << ( 10  - 1)) },
  { 2,   8, -23, -14,  -3, -23,  64,  86,  35, -17,  -4,  79, 132 - (1 << ( 10  - 1)) },
  { 12,   4, -39,  -7,   1, -20,  78,  13,  -8,  11, -42,  98, 310 - (1 << ( 10  - 1)) },
  { 0,   3,  -4,   0,   2,  -7,   6,   0,   0,   3,  -8,  11, 500 - (1 << ( 10  - 1)) },
  { 4,  -7, -25, -19,  -9,   8,  86,  65, -14,  -7,  -7,  97, 168 - (1 << ( 10  - 1)) },
  { 3,   3,   2, -30,   6, -34,  43,  71, -10,   4, -23,  77, 288 - (1 << ( 10  - 1)) },
  { 12,  -3, -34, -14,  -5, -14,  88,  28, -12,   8, -34, 112, 248 - (1 << ( 10  - 1)) },
  { -1,   6,   8, -29,   7, -27,  15,  60,  -4,   6, -21,  39, 394 - (1 << ( 10  - 1)) },
  { 8,  -1,  -7, -22,   5, -41,  63,  40, -13,   7, -28, 105, 280 - (1 << ( 10  - 1)) },
  { 1,   3,  -5,  -1,   1, -10,  12,  -1,   0,   3,  -9,  19, 486 - (1 << ( 10  - 1)) },
```



```
  { 10,  -1, -23, -14,  -3, -27,  78,  24, -14,   8, -28, 102, 288 - (1 << ( 10  - 1)) },
  { 0,   0,  -1,   0,   0,  -1,   1,   0,   0,   0,   0,   1, 512 - (1 << ( 10  - 1)) },
  { 7,   3, -19,  -7,   2, -27,  51,   8,  -6,   7, -24,  64, 394 - (1 << ( 10  - 1)) },
  { 11, -10, -22, -22, -11, -12,  87,  49, -20,   4, -16, 108, 220 - (1 << ( 10  - 1)) },
  { 17,  -2, -69,  -4,  -4,  22, 106,  31,  -7,  13, -63, 121, 190 - (1 << ( 10  - 1)) },
  { 1,   4,  -1,  -7,   5, -26,  24,   0,   1,   3, -18,  51, 438 - (1 << ( 10  - 1)) },
  { 3,   5, -10,  -2,   4, -17,  17,   1,  -2,   6, -16,  27, 480 - (1 << ( 10  - 1)) },
  { 9,   2, -23,  -5,   6, -45,  90, -22,   1,   7, -39, 121, 308 - (1 << ( 10  - 1)) },
  { 4,   5, -15,  -2,   4, -22,  34,  -2,  -2,   7, -22,  48, 438 - (1 << ( 10  - 1)) },
  { 6,   8, -22,  -3,   4, -32,  57,  -3,  -4,  11, -43, 102, 350 - (1 << ( 10  - 1)) },
  { 2,   5, -11,   1,  12, -46,  64, -32,   7,   4, -31,  85, 392 - (1 << ( 10  - 1)) },
  { 5,   5, -12,  -8,   6, -48,  74, -13,  -1,   7, -41, 129, 306 - (1 << ( 10  - 1)) },
  { 0,   1,  -1,   0,   1,  -3,   2,   0,   0,   1,  -3,   4, 508 - (1 << ( 10  - 1)) },
  { -1,   3,  16, -42,   6, -16,   2, 105,   6,   6, -31,  43, 318 - (1 << ( 10  - 1)) },
  { 7,   8, -27,  -4,  -4, -23,  46,  79,  64,  -8, -13,  68, 126 - (1 << ( 10  - 1)) },
  { -3,  12,  -4, -34,  14,  -6, -24, 179,  56,   2, -48,  15, 194 - (1 << ( 10  - 1)) },
  { 8,   0, -16, -25,  -1, -29,  68,  84,   3,  -3, -18,  94, 182 - (1 << ( 10  - 1)) },
  { -3,  -1,  22, -32,   2, -20,   5,  89,   0,   9, -18,  40, 326 - (1 << ( 10  - 1)) },
  { 14,   6, -51,  22, -10, -22,  36,  75, 106,  -4, -11,  56,  78 - (1 << ( 10  - 1)) },
  { 1,  38, -59,  14,   8, -44, -18, 156,  80,  -1, -42,  29, 188 - (1 << ( 10  - 1)) },
  { -1,   2,   4,  -9,   3, -13,   7,  17,  -4,   2,  -6,  17, 474 - (1 << ( 10  - 1)) },
  { 11,  -2, -15, -36,   2, -32,  67,  89, -19,  -1, -14, 103, 206 - (1 << ( 10  - 1)) },
  { -1,  10,   3, -28,   7, -27,   7, 117,  34,   1, -35,  51, 234 - (1 << ( 10  - 1)) },
  { 3,   3,   4, -18,   6, -40,  36,  18,  -8,   7, -25,  86, 368 - (1 << ( 10  - 1)) },
  { -1,   3,   9, -18,   5, -26,  12,  37, -11,   3,  -7,  32, 436 - (1 << ( 10  - 1)) },
  { 0,  17, -38,  -9, -28, -17,  25,  48, 103,   2,  40,  69,  88 - (1 << ( 10  - 1)) },
  { 6,   4, -11, -20,   5, -32,  51,  77,  17,   0, -25,  84, 200 - (1 << ( 10  - 1)) },
  { 0,  -5,  28, -24,  -1, -22,  18,  -9,  17,  -1, -12, 107, 320 - (1 << ( 10  - 1)) },
  { -10,  -4,  17, -30, -29,  31,  40,  49,  44, -26,  67,  67,  80 - (1 << ( 10  - 1)) },
  { -30, -12,  39,  15, -21,  32,  29,  26,  71,  20,  43,  28,  32 - (1 << ( 10  - 1)) },
  { 6,  -7,  -7, -34, -21,  15,  53,  60,  12, -26,  45,  89, 142 - (1 << ( 10  - 1)) },
  { -1,  -5,  59, -58,  -8, -30,   2,  17,  34,  -7,  25, 111, 234 - (1 << ( 10  - 1)) },
  { 7,   1,  -7, -20,  -9, -22,  48,  27,  -4,  -6,   0, 107, 268 - (1 << ( 10  - 1)) },
  { -2,  22,  29, -70,  -4, -28,   2,  19,  94, -40,  14, 110, 220 - (1 << ( 10  - 1)) },
  { 13,   0, -22, -27, -11, -15,  66,  44,  -7,  -5, -10, 121, 218 - (1 << ( 10  - 1)) },
  { 10,   6, -22, -14,  -2, -33,  68,  15,  -9,   5, -35, 135, 264 - (1 << ( 10  - 1)) },
  { 2,  11,   4, -32,  -3, -20,  23,  18,  17,  -1, -28,  88, 354 - (1 << ( 10  - 1)) },
  { 0,   3,  -2,  -1,   3, -16,  16,  -3,   0,   2, -12,  35, 462 - (1 << ( 10  - 1)) },
  { 1,   6,  -6,  -3,  10, -51,  70, -31,   5,   6, -42, 125, 332 - (1 << ( 10  - 1)) },
  { 5,  -7,  61, -71, -36,  -6,  -2,  15,  57,  18,  14, 108, 200 - (1 << ( 10  - 1)) },
  { 9,   1,  35, -70, -73,  28,  13,   1,  96,  40,  36,  80, 120 - (1 << ( 10  - 1)) },
  { 11,  -7,  33, -72, -78,  48,  33,  37,  35,   7,  85,  76,  96 - (1 << ( 10  - 1)) },
  { 4,  15,   1, -26, -24, -19,  32,  29,  -8,  -6,  21, 125, 224 - (1 << ( 10  - 1)) },
  { 11,   8,  14, -57, -63,  21,  34,  51,   7,  -3,  69,  89, 150 - (1 << ( 10  - 1)) },
  { 7,  16,  -7, -31, -38,  -5,  41,  44, -11, -10,  45, 109, 192 - (1 << ( 10  - 1)) },
  { 5,  16,  16, -46, -55,   3,  22,  32,  13,   0,  48, 107, 190 - (1 << ( 10  - 1)) },
  { 2,  10,  -3, -14,  -9, -28,  39,  15, -10,  -5,  -1, 123, 274 - (1 << ( 10  - 1)) },
  { 3,  11,  11, -27, -17, -24,  18,  22,   2,   4,   3, 100, 300 - (1 << ( 10  - 1)) },
  { 0,   1,   7,  -9,   3, -20,  16,   3,  -2,   0,  -9,  61, 410 - (1 << ( 10  - 1)) },
};
classToFilterMapping[25][16] =
{
  { 0,   1,   2,   3,   4,   5,   6,   7,   9,  19,  32,  41,  42,  44,  46,  63 },
  { 0,   1,   2,   4,   5,   6,   7,   9,  11,  16,  25,  27,  28,  31,  32,  47 },
  { 5,   7,   9,  11,  12,  14,  15,  16,  17,  18,  19,  21,  22,  27,  31,  35 },
  { 7,   8,   9,  11,  14,  15,  16,  17,  18,  19,  22,  23,  24,  25,  35,  36 },
  { 7,   8,  11,  13,  14,  15,  16,  17,  19,  20,  21,  22,  23,  24,  25,  27 },
  { 1,   2,   3,   4,   6,  19,  29,  30,  33,  34,  37,  41,  42,  44,  47,  54 },
  { 1,   2,   3,   4,   6,  11,  28,  29,  30,  31,  32,  33,  34,  37,  47,  63 },
  { 0,   1,   4,   6,  10,  12,  13,  19,  28,  29,  31,  32,  34,  35,  36,  37 },
  { 6,   9,  10,  12,  13,  16,  19,  20,  28,  31,  35,  36,  37,  38,  39,  52 },
  { 7,   8,  10,  11,  12,  13,  19,  23,  25,  27,  28,  31,  35,  36,  38,  39 },
  { 1,   2,   3,   5,  29,  30,  33,  34,  40,  43,  44,  46,  54,  55,  59,  62 },
  { 1,   2,   3,   4,  29,  30,  31,  33,  34,  37,  40,  41,  43,  44,  59,  61 },
  { 0,   1,   3,   6,  19,  28,  29,  30,  31,  32,  33,  34,  37,  41,  44,  61 },
```

```
  { 1,   6,  10,  13,  19,  28,  29,  30,  32,  33,  34,  35,  37,  41,  48,  52 },
  { 0,   5,   6,  10,  19,  27,  28,  29,  32,  37,  38,  40,  41,  47,  49,  58 },
  { 1,   2,   3,   4,  11,  29,  33,  42,  43,  44,  45,  46,  48,  55,  56,  59 },
  { 0,   1,   2,   5,   7,   9,  29,  40,  43,  44,  45,  47,  48,  56,  59,  63 },
  { 0,   4,   5,   9,  14,  19,  26,  35,  36,  43,  45,  47,  48,  49,  50,  51 },
  { 9,  11,  12,  14,  16,  19,  20,  24,  26,  36,  38,  47,  49,  50,  51,  53 },
  { 7,   8,  13,  14,  20,  21,  24,  25,  26,  27,  35,  38,  47,  50,  52,  53 },
  { 1,   2,   4,  29,  33,  40,  41,  42,  43,  44,  45,  46,  54,  55,  56,  58 },
  { 2,   4,  32,  40,  42,  43,  44,  45,  46,  54,  55,  56,  58,  59,  60,  62 },
  { 0,  19,  42,  43,  45,  46,  48,  54,  55,  56,  57,  58,  59,  60,  61,  62 },
  { 8,  13,  36,  42,  45,  46,  51,  53,  54,  57,  58,  59,  60,  61,  62,  63 },
  { 8,  13,  20,  27,  36,  38,  42,  46,  52,  53,  56,  57,  59,  61,  62,  63 },
};
```

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

$$AlfCoeff_L[filtIdx][12] = 512 - \Sigma_k (AlfCoeff_L[filtIdx][k] << 1), \text{ with } k = 0..NumAlfCoefs-2 \quad (7\text{-}56)$$

It is a requirement of bitstream conformance that the values of $AlfCoeff_L[filtIdx][j]$ with filtIdx = 0..NumAlfFilters − 1, j = 0..11 shall be in the range of -2^7 to $2^7 - 1$, inclusive and that the values of $AlfCoeff_L[filtIdx][12]$ shall be in the range of 0 to $2^8 - 1$, 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:

$$expGoOrderC[i] = alf\_chroma\_min\_eg\_order\_minus1 + 1 + alf\_chroma\_eg\_order\_increase\_flag[i] \quad (7\text{-}57)$$

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.

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

$$golombOrderIdxC[ ] = \{ 0, 0, 1, 0, 0, 1 \} \quad (7\text{-}58)$$

$$k = expGoOrderC[golombOrderIdxC[j]] \quad (7\text{-}59)$$

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 negative value.

– Otherwise (alf_chroma_coeff_sign[j] is equal to 1), the corresponding chroma filter coefficient has a positive value.

When alf_chroma_coeff_sign[j] is not present, it is inferred to be equal to 1.

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

$$AlfCoeff_C[j] = alf\_chroma\_coeff\_abs[j] * (1 - 2 * !alf\_chroma\_coeff\_sign[j]) \quad (7\text{-}60)$$

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

$$AlfCoeff_C[6] = 512 - \Sigma_k (AlfCoeff_C[k] << 1), \text{ with } k = 0..5 \quad (7\text{-}61)$$

7.4.6 Reference picture list structure semantics

The ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure may be present in an SPS or in a slice header. Depending on whether the syntax structure is included in a slice header or an SPS, the following applies:

– If present in a slice header, the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure specifies reference picture list listIdx of the current picture (the picture containing the slice).

– Otherwise (present in an SPS), the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) 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 slices containing ref_pic_list_idx[listIdx] equal to an index into the list of the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structures included in the SPS, and 2) is in a CVS that has the SPS as the active SPS.

num_strp_entries[listIdx][rplsIdx] specifies the number of STRP entries in the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure.

num_ltrp_entries[listIdx][rplsIdx] specifies the number of LTRP entries in the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure. When not present, the value of num_ltrp_entries[listIdx][rplsIdx] is inferred to be equal to 0.

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

$$\text{NumEntriesInList}[\text{listIdx}][\text{rplsIdx}] = \text{num\_strp\_entries}[\text{listIdx}][\text{rplsIdx}] + \text{num\_ltrp\_entries}[\text{listIdx}][\text{rplsIdx}] \quad (7\text{-}62)$$

The value of NumEntriesInList[listIdx][rplsIdx] shall be in the range of 0 to sps_max_dec_pic_buffering_minus1, inclusive.

lt_ref_pic_flag[listIdx][rplsIdx][i] equal to 1 specifies that the i-th entry in the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure is an LTRP entry. lt_ref_pic_flag[listIdx][rplsIdx][i] equal to 0 specifies that the i-th entry in the ref_pic_list_struct(listIdx, rplsIdx, ltrpFlag) syntax structure is an STRP entry. When not present, the value of lt_ref_pic_flag[listIdx][rplsIdx][i] is inferred to be equal to 0.

It is a requirement of bitstream conformance that the sum of lt_ref_pic_flag[listIdx][rplsIdx][i] for all values of i in the range of 0 to NumEntriesInList[listIdx][rplsIdx] − 1, inclusive, shall be equal to num_ltrp_entries[listIdx][rplsIdx].

delta_poc_st[listIdx][rplsIdx][i], when the i-th entry is the first STRP entry in ref_pic_list_struct(rplsIdx, ltrpFlag) 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(rplsIdx, ltrpFlag) 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, ltrpFlag) syntax structure.

The value of delta_poc_st[listIdx][rplsIdx][i] shall be in the range of -2^{15} to $2^{15} - 1$, 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, ltrpFlag) 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, ltrpFlag) 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 < NumEntriesInList[ listIdx ][ rplsIdx ]; i++ ) {                                    (7-63)
    if( !lt_ref_pic_flag[ i ][ RplsIdx[ i ] ][ j ] ) {
        DeltaPocSt[ listIdx ][ rplsIdx ][ i ] = ( strp_entry_sign_flag[ listIdx ][ rplsIdx ][ i ]) ?
                delta_poc_st[ listIdx ][ rplsIdx ][ i ] : 0 − 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, ltrpFlag) syntax structure. The length of the poc_lsb_lt[listIdx][rplsIdx][i] syntax element is log2_max_pic_order_cnt_lsb_minus4 + 4 bits.

7.4.7 Slice data semantics

7.4.7.1 General slice data semantics

end_of_tile_one_bit shall be equal to 1.

7.4.7.2 Coding tree unit semantics

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

alf_ctb_flag [xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY] equal to 1 specifies that the adaptive loop filter is applied to the coding tree block of the luma component of the coding tree unit at luma location (xCtb, yCtb). alf_ctb_flag [xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY] equal to 0 specifies that the adaptive loop filter is not applied to the coding tree block of the luma of the coding tree unit at luma location (xCtb, yCtb).

When alf_ctb_flag [xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY] is not present, it is inferred to be equal to slice_alf_enabled_flag.

7.4.7.3 Split unit semantics

split_cu_flag[x0][y0] specifies whether a coding unit is split into coding units with half horizontal and vertical size. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When split_cu_flag[x0][y0] is not present, the value of split_cu_flag[x0][y0] is inferred to be equal to 0.

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

- A variable yBoudnaryCtb is set equal to ((pic_width_in_luma_samples >> (CtbLog2SizeY - 1)) << (CtbLog2SizeY - 1))
- A variable xBoudnaryCtb is set equal to ((pic_height_in_luma_samples >> (CtbLog2SizeY - 1)) << (CtbLog2SizeY - 1))
- The variable allowSplitBtVer is derived as follows:
  - A variable log2CbWidthTemp is set equal to log2CbWidth – 1.
  - A variable log2CbSizeLongerSide is set equal to (log2CbWidthTemp > log2CbHeight ? log2CbWidthTemp : log2CbHeight).
  - A variable ratioWH is set equal to Abs(log2CbWidthTemp – log2CbHeight).
  - If one or more of the following conditions are true, allowSplitBtVer is set equal to FALSE.
    - log2CbWidthTemp is less than 4.
    - If ratioWH is greater than 2, log2CbWidth is smaller than log2CbHeight or y0 is smaller than yBoudnaryCtb or x0 is greater than or equal to xBoudnaryCtb.
    - if ratioWH is equal to 0, log2CbSizeLongerSide is greater than MaxCbLog2Size11Ratio or log2CbSizeLongerSize is less than MinCbLog2Size11Ratio.
    - if ratioWH is equal to 1, log2CbSizeLongerSide is greater than MaxCbLog2Size12Ratio or log2CbSizeLongerSize is less than MinCbLog2Size12Ratio.
    - if ratioWH is equal to 2, log2CbSizeLongerSide is greater than MaxCbLog2Size14Ratio or log2CbSizeLongerSize is less than MinCbLog2Size14Ratio.
  - Otherwise, allowSplitBtVer is set equal to TRUE.
- The variable allowSplitBtHor is derived as follows:
  - A variable log2CbHeightTemp is set equal to log2CbHeight – 1.
  - A variable log2CbSizeLongerSide is set equal to (log2CbWidth > log2CbHeightTemp ? log2CbWidth : log2CbHeightTemp).
  - A variable ratioWH is set equal to Abs(log2CbWidth – log2CbHeightTemp).
  - If one or more of the following conditions are true, allowSplitBtHor is set equal to FALSE.
    - log2CbHeightTemp is less than 4.
    - If ratioWH is greater than 2, log2CbHeight is smaller than or equal to log2CbWidth or x0 is smaller than xBoudnaryCtb or y0 is greater than or equal to yBoudnaryCtb.
    - if ratioWH is equal to 0, log2CbSizeLongerSide is greater than MaxCbLog2Size11Ratio or log2CbSizeLongerSize is less than MinCbLog2Size11Ratio.

- if ratioWH is equal to 1, log2CbSizeLongerSide is greater than MaxCbLog2Size12Ratio or log2CbSizeLongerSize is less than MinCbLog2Size12Ratio.
- if ratioWH is equal to 2, log2CbSizeLongerSide is greater than MaxCbLog2Size14Ratio or log2CbSizeLongerSize is less than MinCbLog2Size14Ratio.

- Otherwise, allowSplitBtHor is set equal to TRUE.

- The variable allowSplitTtVer is derived as follows:
  - A variable log2CbWidthTemp is set equal to log2CbWidth – 2.
  - A variable log2CbSizeLongerSide is set equal to (log2CbWidthTemp > log2CbHeight ? log2CbWidthTemp : log2CbHeight).
  - A variable ratioWH is set equal to Abs(log2CbWidthTemp – log2CbHeight).
  - If one or more of the following conditions are true, allowSplitTtVer is set equal to FALSE.
    - log2CbWidthTemp is less than 4.
    - if ratioWH is greater than 2, log2CbWidth is smaller than log2CbHeight or y0 is smaller than yBoudnaryCtb or x0 is greater than or equal to xBoudnaryCtb.
    - log2SizeLongerSide is greater than MaxTtLog2Size or log2SizeLongerSide is less than MinTtLog2Size.
    - if ratioWH is equal to 0, log2CbSizeLongerSide is greater than MaxCbLog2Size11Ratio or log2CbSizeLongerSize is less than MinCbLog2Size11Ratio.
    - if ratioWH is equal to 1, log2CbSizeLongerSide is greater than MaxCbLog2Size12Ratio or log2CbSizeLongerSize is less than MinCbLog2Size12Ratio.
    - if ratioWH is equal to 2, log2CbSizeLongerSide is greater than MaxCbLog2Size14Ratio or log2CbSizeLongerSize is less than MinCbLog2Size14Ratio.
  - Otherwise, allowSplitTtVer is set equal to TRUE.
- The variable allowSplitTtHor is derived as follows:
  - A variable log2CbHeightTemp is set equal to log2CbWidth – 2.
  - A variable log2CbSizeLongerSide is set equal to (log2CbWidth > log2CbHeightTemp ? log2CbWidth : log2CbHeightTemp).
  - A variable ratioWH is set equal to Abs(log2CbWidth – log2CbHeightTemp).
  - If one or more of the following conditions are true, allowSplitTtHor is set equal to FALSE.
    - log2CbHeightTemp is less than 4.
    - if ratioWH is greater than 2, log2CbHeight is smaller than or equal to log2CbWidth or x0 is smaller than xBoudnaryCtb or y0 is greater than or equal to yBoudnaryCtb.
    - log2SizeLongerSide is greater than MaxTtLog2Size or log2SizeLongerSide is less than MinTtLog2Size.
    - if ratioWH is equal to 0, log2CbSizeLongerSide is greater than MaxCbLog2Size11Ratio or log2CbSizeLongerSize is less than MinCbLog2Size11Ratio.
    - if ratioWH is equal to 1, log2CbSizeLongerSide is greater than MaxCbLog2Size12Ratio or log2CbSizeLongerSize is less than MinCbLog2Size12Ratio.
    - if ratioWH is equal to 2, log2CbSizeLongerSide is greater than MaxCbLog2Size14Ratio or log2CbSizeLongerSize is less than MinCbLog2Size14Ratio.
  - Otherwise, allowSplitTtHor is set equal to TRUE.

btt_split_flag[x0][y0] specifies whether a coding unit is split. btt_split_flag[x0][y0] equal to 1 specifies that a coding unit is split into two or three coding units. btt_split_flag[x0][y0] equal to 0 specifies that a coding unit is not split.

When btt_split_flag[x0][y0] is not present, the value of btt_split_flag[x0][y0] is inferred to be equal to 0.

btt_split_dir[x0][y0] equal to 0 specifies that a coding unit is split in horizontal direction. btt_split_dir[x0][y0] equal to 1 specifies that a coding unit is split in vertical direction.

When btt_split_dir[x0][y0] is not present, the value of btt_split_dir[x0][y0] is inferred to be equal to 1 if allowSplitBtVer or allowSplitTtVer is equal to 1, otherwise it is inferred to be equal to 0.

btt_split_type[x0][y0] equal to 0 specifies that a coding unit is split into two coding units. btt_split_type[x0][y0] equal to 1 specifies that a coding unit is split into three coding units.

When btt_split_type[x0][y0] is not present, the value of btt_split_type[x0][y0] is inferred to be equal to derived as follows:

– If one of the follows are true, btt_split_type[x0][y0] is set equal to 1:
  – btt_split_dir[x0][y0] is equal to 0 and allowSplitTtHor is equal to 1.
  – btt_split_dir[x0][y0] is equal to 1 and allowSplitTtVer is equal to 1
– btt_split_type[x0][y0] is set equal to 0.

The array SplitMode[x][y] is derived as follows for x = x0..x0 + nCbW – 1 and y = y0..y0 + nCbH – 1:

– If btt_split_flag[x0][y0] is equal to 1, the following applies:
  – If btt_split_dir[x0][y0] is equal to 0, the following applies:
    – If btt_split_type[x0][y0] is equal to 0, SplitMode[x][y] is set equal to SPLIT_BT_HOR.
    – Otherwise (btt_split_type[x0][y0] is equal to 1), SplitMode[x][y] is set equal to SPLIT_TT_VER.
  – Otherwse (btt_split_dir[x0][y0] is equal to 1),.the following applies:
    – If btt_split_type[x0][y0] is equal to 0, SplitMode[x][y] is set equal to SPLIT_BT_VER.
    – Otherwise (btt_split_type[x0][y0] is equal to 1), SplitMode[x][y] is set equal to SPLIT_TT_VER.
– Otherwise, if x0 + (1 << log2CbWidth) is greater than pic_width_in_luma_samples and y0 + (1 << log2CbHeight) is samller than or equal to pic_height_in_luma_samples and log2CbWidth is equal to 6 and log2CbHeight is equal to 7, SplitMode[x][y] is set equal to SPLIT_BT_HOR.
– Otherwise, if x0 + (1 << log2CbWidth) is greater than pic_width_in_luma_samples and y0 + (1 << log2CbHeight) is samller than or equal to pic_height_in_luma_samples, SplitMode[x][y] is set equal to SPLIT_BT_VER.
– Otherwise, if x0 + (1 << log2CbWidth) is smaller than or equal to pic_width_in_luma_samples and y0 + (1 << log2CbHeight) is greater than pic_height_in_luma_samples and log2CbWidth is equal to 7 and log2CbHeight is equal to 6, SplitMode[x][y] is set equal to SPLIT_BT_VER.
– Otherwise, if x0 + (1 << log2CbWidth) is smaller than or equal to pic_width_in_luma_samples and y0 + (1 << log2CbHeight) is greater than pic_height_in_luma_samples, SplitMode[x][y] is set equal to SPLIT_BT_HOR.
– Otherwise, if x0 + (1 << log2CbWidth) is greater than pic_width_in_luma_samples and y0 + (1 << log2CbHeight) is greater than pic_height_in_luma_samples, SplitMode[x][y] is set equal to SPLIT_QUAD.
– Otherwise, if btt_split_flag[x0][y0] is equal to 0, SplitMode[x][y] is set equal to NO_SPLIT.

The array CtDepth[x][y] specifies the coding tree depth for a luma coding block covering the location (x, y). When split_cu_flag[x0][y0] and btt_split_flag[x0][y0] are equal to 0, CtDepth[x][y] is inferred to be equal to ctDepth for x = x0..x0 + nCbW – 1 and y = y0..y0 + nCbH – 1.

The variable allowSplitUnitCodingOrder is derived as follows:

– A variable log2CbSizeLongerSide is set equal to (log2CbWidth > log2CbHeight ? log2CbWidth : log2CbHeight).
– A variable log2CbSizeShorterSide is set equal to (log2CbWidth > log2CbHeight ? log2CbHeight : log2CbWidth).
– If one or more of the following conditions are true, allowSplitUnitCodingOrder is set equal to FALSE.
  – log2SizeLongerSide is greater than MaxSucoLog2Size or log2CbSizeShorterSide is less than MinSucoLog2Size.
  – x0 + (1 << log2CbWidth) is greater than pic_width_in_luma_samples or y0 + (1 << log2CbHeight) is greater than pic_height_in_luma_samples.

- log2CbWidth is equal to or smaller than log2CbHeight, and Split[x0][y0] is not equal to SPLIT_QUAD.
- SplitMode[x0][y0] is equal to SPLIT_BT_HOR, SPLIT_TT_HOR or NO_SPLIT.

– Otherwise, allowSplitUnitCodingOrder is set equal to TRUE.

split_unit_coding_order_flag[x0][y0] specifies the coding order of split coding units. split_unit_coding_order_flag[x0][y0] equal to 0 specifies the split coding units is coded from left to right. split_unit_coding_order_flag[x0][y0] equal to 1 specifies the split coding units is coded from right to left. When split_unit_coding_order_flag[x0][y0] is not present, it is inferred to be equal to splitUnitOrder.

7.4.7.4 Coding unit semantics

cu_skip_flag[x0][y0] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, no more syntax elements except the motion vector predictor indices mvp_idx_l0[x0][y0] and mvp_idx_l1[x0][y0] are parsed after cu_skip_flag[x0][y0]. cu_skip_flag[x0][y0] equal to 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 related 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.

mvp_idx_l0[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_idx_l0[x0][y0] is not present, it is inferred to be equal to 0.

mvp_idx_l1[x0][y0] has the same semantics as mvp_idx_l0, with l0 and list 0 replaced by l1 and list 1, respectively.

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_group_idx[x0][y0] specifies which direction of the merging candidate is used at MMVD mode. The merge candidate is decided by mmvd_merge_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. When mmvd_group_idx[x0][y0] is not present, it is inferred to be equal to 0.

mmvd_merge_idx[x0][y0] specifies which candidate of 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. When mmvd_merge_idx[x0][y0] is not present, it is inferred to be equal to 0.

mmvd_distance_idx[x0][y0] specifies the index used to derive MmvdDistance[x0][y0] as specified in Table 7-3. 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_distance_idx[x0][y0] is not present, it is inferred to be equal to 0.

Table 7-3 – Specification of MmvdDistance[x0][y0] based on mmvd_distance_idx[x0][y0].

mmvd_distance_idx[x0][y0]	MmvdDistance[x0][y0]
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

mmvd_direction_idx[x0][y0] specifies index used to derive MmvdSign[x0][y0] as specified in Table 7-4. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7-4 – 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 the merge plus MVD offset MmvdOffset[x0][y0] are derived as follows:

MmvdOffset[x0][y0][0] = (MmvdDistance[x0][y0] << 2) * MmvdSign[x0][y0][0] (7-64)

MmvdOffset[x0][y0][1] = (MmvdDistance[x0][y0] << 2) * MmvdSign[x0][y0][1] (7-65)

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

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

mvp_idx[x0][y0] specifies the motion vector predictor candidate index of the motion vector 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 mvp_idx[x0][y0] is not present, it is inferred to be equal to 0.

pred_mode_flag[x0][y0] equal to 0 specifies that the current coding unit is coded in inter prediction mode. pred_mode_flag[x0][y0] 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 + nCbW – 1 and y = y0..y0 + nCbH – 1:

– If pred_mode_flag[x0][y0] is equal to 0, CuPredMode[x][y] is set equal to MODE_INTER.

– Otherwise (pred_mode_flag[x0][y0] is equal to 1), CuPredMode[x][y] is set equal to MODE_INTRA.

When pred_mode_flag[x0][y0] is not present, the variable CuPredMode[x][y] is derived as follows for x = x0..x0 + nCbW − 1 and y = y0..y0 + nCbH − 1:

– If slice_type is equal to I, CuPredMode[x][y] is inferred to be equal to MODE_INTRA.

– Otherwise (slice_type is equal to P or B), the following applies:

  – If when cu_skip_flag[x0][y0] is equal to 1, CuPredMode[x][y] is inferred to be equal to MODE_SKIP.

  – Otherwise, if direct_mode_flag[x0][y0] is equal to 1, CuPredMode[x][y] is inferred to be equal to MODE_DIRECT.

  – Otherwise, CuPredMode[x][y] is inferred to be equal to MODE_INTRA.

intra_pred_mode[x0][y0] specifies the intra prediction mode for both luma and 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.

The syntax elements **intra_luma_pred_mpm_flag**[x0][y0], **intra_luma_pred_mpm_idx**[x0][y0], **intra_luma_pred_pims_flag**[x0][y0], **intra_luma_pred_pims_idx**[x0][y0] and **intra_luma_pred_rem_mode**[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_pred_mpm_flag[x0][y0] or intra_luma_pred_pims_flag[x0][y0] is equal to 1, the intra prediction mode is inferred from neighbouring intra-predicted coding units according to clause 8.4.2.

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

ibc_flag[x0][y0] specifies whether the intra block copy mode is used 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 ibc_flag[x0][y0] is not present, it is inferred to be equal to 0.

The variable CuPredMode[x][y] is derived as follows for x = x0..x0 + nCbW – 1 and y = y0..y0 + nCbH – 1:

– When ibc_flag[x0][y0] is equal to 1, CuPredMode[x][y] is set equal to MODE_IBC.

amvr_idx[x0][y0] specifies the resolution of the motion vector 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 amvr_idx[x0][y0] is not present, it is inferred to be equal to 0.

merge_mode_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_mode_flag[x0][y0] is not present, it is inferred to be equal to 0.

direct_mode_flag[x0][y0] specifies whether the inter prediction parameters for the current coding unit are inferred from the temporal co-located 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 sample of the picture. When direct_mode_flag[x0][y0] is not present, it is inferred to be equal to 0.

inter_pred_idc[x0][y0] specifies whether list 0, list 1, or bi-prediction is used for the current coding unit according to Table 7-5. 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 inter_pred_idc[x0][y0] is not present, it is inferred as follows:

– If slice_type is equal to P, it is inferred to be equal to PRED_L0.

– Otherwise (slice_type is equal to B), it is inferred to be equal to PRED_BI.

Table 7-5 – Name association to inter prediction mode

inter_pred_idc	Name of inter_pred_idc
0	PRED_L0
1	PRED_L1
2	PRED_BI

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 th the top-left luma sample of the picture. When ref_idx_l0[x0][y0] is not present it is inferred to be equal to 0.

abs_mvd_l0[x0][y0][compIdx] specifies the absolute value of a motion vector component difference of list 0 for the current coding unit. When abs_mvd_l0[x0][y0][compIdx] is not present, it is inferred to be equal to 0.

mvd_l0_sign_flag[x0][y0][compIdx] specifies the sign of a motion vector component difference of list 0 for the current coding unit as follows:

– If mvd_l0_sign_flag [x0][y0][compIdx] is equal to 0, the corresponding motion vector component difference has a positive value.

– Otherwise (mvd_l0_sign_flag[x0][y0][compIdx] is equal to 1), the corresponding motion vector component difference has a negative value.

When mvd_l0_sign_flag[x0][y0][compIdx] is not present, it is inferred to be equal to 0.

The variable motion vector difference of list 0 MvdL0[x0][y0][compIdx] for compIdx = 0..1, which specifies the difference between a list 0 motion vector component to be used and its prediction, is derived as follows:

$$MvdL0[x0][y0][compIdx] = abs\_mvd\_l0[x0][y0][compIdx] * (1 - 2 * mvd\_l0\_sign\_flag[x0][y0][compIdx]) \quad (7\text{-}66)$$

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 value of MvdL0X[x0][y0][compIdx] shall be in the range of -2^{15} to $2^{15} - 1$, inclusive. 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.

ref_idx_l1[x0][y0] has the same semantics as ref_idx_l0[x0][y0],, with l0 and list 0 replaced by l1 and list 1, respectively.

abs_mvd_l1[x0][y0][compIdx] has the same semantics as abs_mvd_l0.

mvd_l0[x0][y0][compIdx], with l0 and list 0 replaced by l1 and list 1, respectively.

mvd_l1_sign_flag[x0][y0][compIdx] has the same semantics as and mvd_l0_sign_flag[x0][y0][compIdx], with l0 and list 0 replaced by l1 and list 1, respectively.

The variable MvdL1[x0][y0][compIdx] for compIdx = 0..1, which specifies the difference between a list 1 motion vector component to be used and its prediction, is derived as follows:

$$\text{MvdL1[x0][y0][compIdx]} = \text{abs\_mvd\_l1[x0][y0][compIdx]} * (1 - 2 * \text{mvd\_l1\_sign\_flag[x0][y0][compIdx]}) \quad (7\text{-}67)$$

The value of MvdL1[x0][y0][compIdx] shall be in the range of -2^{15} to $2^{15} - 1$, inclusive. 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.

When sps_amvr_flag is equal to 1, the variables MvdL0[x0][y0][compIdx] and MvdL1[x0][y0][compIdx] for compIdx = 0..1 are modified as follows:

$$\text{MvdL0[x0][y0][compIdx]} = \text{MvdL0[x0][y0][compIdx]} << (\text{amvr\_idx[x0][y0]}) \quad (7\text{-}68)$$

$$\text{MvdL1[x0][y0][compIdx]} = \text{MvdL1[x0][y0][compIdx]} << (\text{amvr\_idx[x0][y0]}) \quad (7\text{-}69)$$

bi_pred_idx[x0][y0] specifies whether the reference indices and the motion vector differences for list 0 or list 1 are present. bi_pred_idx[x0][y0] equal to 0 specifies the reference indices and the motion vector differences for list 0 and list 1 are present. bi_pred_idx[x0][y0] equal to 1 specifies the reference indices for list 0 and list 1 are not present and the motion vector difference for list 0 is not present. bi_pred_idx[x0][y0] equal to 2 specifies the reference indices for list 0 and list 1 are not present and the motion vector difference for list 1 is not present. When bi_pred_idx[x0][y0] is not present, it is inferred to be equal to 0.

cbf_all equal to 1 specifies that the transform_unit( ) syntax structure is present for the current coding unit. cbf_all equal to 0 specifies that transform_unit( ) syntax structure is not present for the current coding unit.

When cbf_all is not present, its value is inferred to be equal to 1.

7.4.7.5 Transform unit semantics

The variables allowAtsInterVerHalf, allowAtsInterHorHalf, allowAtsInterVerQuad and allowAtsInterHorQuad are derived as follows:

- The variable allowAtsInterVerHalf is set equal to TRUE if all of the following conditions are true:
  - A variable log2CbWidth is greater than or equal to 3.
  - A variable log2CbWidth is less than or equal to 6.
  - A variable log2CbHeight is less than or equal to 6.
- Otherwise, allowAtsInterVerHalf is set equal to FALSE.
- The variable allowAtsInterVerQuad is set equal to TRUE if all of the following conditions are true:
  - A variable log2CbWidth is greater than or equal to 4.
  - A variable log2CbWidth is less than or equal to 6.
  - A variable log2CbHeight is less than or equal to 6.

– Otherwise, allowAtsInterVerQuad is set equal to FALSE.

– The variable allowAtsInterHorHalf is set equal to TRUE if all of the following conditions are true:
  – A variable log2CbHeight is greater than or equal to 3.
  – A variable log2CbWidth is less than or equal to 6.
  – A variable log2CbHeight is less than or equal to 6.

– Otherwise, allowAtsInterHorHalf is set equal to FALSE.

– The variable allowAtsInterHorQuad is set equal to TRUE if all of the following conditions are true:
  – A variable log2CbHeight is greater than or equal to 4.
  – A variable log2CbWidth is less than or equal to 6.
  – A variable log2CbHeight is less than or equal to 6.

– Otherwise, allowAtsInterHorQuad is set equal to FALSE.

cbf_cb equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. When cbf_cb is not present, it is inferred to be equal to 1.

cbf_cr equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. When cbf_cr is not present, it is inferred to be equal to 1.

cbf_luma equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. When cbf_luma is not present, it is inferred to be equal to 1.

cu_qp_delta specifies the difference CuQpDeltaVal value between the luma quantization parameter of the current coding unit and its prediction. The decoded value of cu_qp_delta shall be in the range of –26 to +25, inclusive. when it is not present, cu_qp_delta shall be inferred to be equal to 0.When cu_qp_delta is present, the variable cuQpDeltaCode is derived as follows:

cuQpDeltaCode = 0 (7-70)

ats_cu_intra_flag[x0][y0] specifies whether an adaptive transform selection is 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 ats_cu_intra_flag[x0][y0] is not present, it is inferred to be equal to 0.

It is a requirement of bitstream conformance that when CuPredMode[x][y] is equal to MODE_IBC, ats_cu_intra_flag[x0][y0] shall be equal to 0.

ats_hor_mode[x0][y0] specifies which kernel is applied to the residual samples along the horizontal 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 ats_hor_mode[x0][y0] is not present, it is inferred to be equal to 0.

ats_ver_mode[x0][y0] specifies which kernel is applied to the residual samples along the 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 ats_ver_mode [x0][y0] is not present, it is inferred to be equal to 0.

ats_cu_inter_flag[x0][y0] equal to 1 specifies that for the current coding unit, sub-block transform is used. ats_cu_inter_flag[x0][y0] equal to 0 specifies that for the current coding unit, sub-block transform is not used.

When ats_cu_inter_flag[x0][y0] is not present, its value is inferred to be equal to 0.

It is a requirement of bitstream conformance that when CuPredMode[x][y] is equal to MODE_IBC, ats_cu_inter_flag[x0][y0] shall be equal to 0.

ats_cu_inter_quad_flag[x0][y0] equal to 1 specifies that for the current coding unit, the sub-block transform includes a transform unit of 1/4 size of the current coding unit. ats_cu_inter_quad_flag[x0][y0] equal to 0 specifies that for the current coding unit the sub-block transform includes a transform unit of 1/2 size of the current coding unit.

When ats_cu_inter_quad_flag[x0][y0] is not present, its value is inferred to be equal to 0.

ats_cu_inter_horizontal_flag[x0][y0] equal to 1 specifies that the current coding unit is tiled into 2 transform units by a horizontal split. ats_cu_inter_horizontal_flag[x0][y0] equal to 0 specifies that the current coding unit is tiled into 2 transform units by a vertical split.

When ats_cu_inter_horizontal_flag[x0][y0] is not present, its value is derived as follows:

– If ats_cu_inter_quad_flag[x0][y0] is equal to 1, ats_cu_inter_horizontal_flag[x0][y0] is set to be equal to allowAtsInterHorQuad.

– Otherwise (ats_cu_inter_quad_flag[x0][y0] is equal to 0), ats_cu_inter_quad_flag[x0][y0] is set to be equal to allowAtsInterHorHalf.

ats_cu_inter_pos_flag[x0][y0] 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[x0][y0] 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

7.4.7.6 Run-length residual coding semantics

coeff_zero_run specifies the number of zero-valued transform coefficient levels that are located before the position of the next non-zero transform coefficient level in a scan of transform coefficient levels.

coeff_abs_level_minus1 plus 1 specifies the absolute value of a transform coefficient level at the givien scanning position.

coeff_sign_flag specifies the sign of a transform coefficient level for the given scanning position as follows:

– If coeff_sign_flag is equal to 0, the corresponding transform coefficient level has a positive value.

– Otherwise (coeff_sign_flag is equal to 1), the corresponding transform coefficient level has a negative value.

When coeff_sign_flag is not present, it is inferred to be equal to 0.

coeff_last_flag specifies for the given scanning position whether there are non-zero transform coefficient levels for the next subsequent scanning positions to nCbW * nCbH – 1 as follows:

– If coeff_last_flag is equal to 1, all following transform coefficient levels (in scanning order) of the block have a value equal to 0.

– Otherwise (coeff_last_flag is equal to 0), there are further non-zero transform coefficient levels along the scanning path.

7.4.7.7 Enhanced residual coding semantics

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 (log2TrafoWidth << 1) − 1, inclusive.

last_sig_coeff_y_prefix specifies the prefix of the row position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_y_prefix shall be in the range of 0 to (log2TrafoHeight << 1) − 1, inclusive.

last_sig_coeff_x_suffix specifies the suffix of the column position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_x_suffix shall be in the range of 0 to (1 << ((last_sig_coeff_x_prefix >> 1) − 1)) − 1, inclusive.
The column position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffX is derived as follows:

- If last_sig_coeff_x_suffix is not present, the following applies:
  LastSignificantCoeffX = last_sig_coeff_x_prefix (7-71)
- Otherwise (last_sig_coeff_x_suffix is present), the following applies:
  LastSignificantCoeffX = (1 << ((last_sig_coeff_x_prefix >> 1) − 1)) * (2 + (last_sig_coeff_x_prefix & 1)) + last_sig_coeff_x_suffix (7-72)

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 (7-73)
- Otherwise (last_sig_coeff_y_suffix is present), the following applies:
  LastSignificantCoeffY = (1 << ((last_sig_coeff_y_prefix >> 1) − 1)) * (2 + (last_sig_coeff_y_prefix & 1)) + last_sig_coeff_y_suffix (7-74)

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, sig_coeff_flag[xC][yC] is inferred to be equal to 1.
- Otherwise, sig_coeff_flag[xC][yC] is inferred to be equal to 0.

coeff_abs_level_greaterA_flag[n] specifies for the scanning position n whether there are absolute values of transform coefficient levels greater than 1.

When coeff_abs_level_greaterA_flag[n] is not present, it is inferred to be equal to 0.

coeff_abs_level_greaterB_flag[n] specifies for the scanning position n whether there are absolute values of transform coefficient levels greater than 2.

When coeff_abs_level_greaterB_flag[n] is not present, it is inferred to be equal to 0.

coeff_signs_group specifies the binary representation for sign flags for the group transform coefficient. The number of bits used to represent coeff_signs_group is determined by a number of non-zero transform coefficients present in the transform coefficients group and should be in the range 1..16, inclusive.

When coeff_signs_group is not present, it is inferred to be equal to 0.

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

8 Decoding process

8.1 General decoding process

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 slice 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 slice of a picture.
   - At the beginning of the decoding process for each slice 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]) and the decoding process for collocated picture specified in clause 8.3.4 is invoked for derivation of the variables ColPic.
   - 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 slice of a picture.
3. The processes in clauses 8.4, 8.5, 8.6 and 8.8 specify decoding processes using syntax elements in all syntax structure layers. It is the requirement of bitstream conformance that the coded slices of the picture shall contain slice data for every CTU of the picture, such that the division of the picture into slices, the division of the slices into tiles and the division of the tiles into CTUs each forms a partitioning of the picture.
4. After all slices of the current picture have been decoded, the current decoded picture is marked as "used for short-term reference".

8.2 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.

8.3 Slice decoding process

8.3.1 Decoding process for picture order count

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

Picture order counts are used to identify pictures, for deriving motion parameters in merge mode and motion vector prediction, and for decoder conformance checking.

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

When the current picture is not an IDR picture, the variables prevPicOrderCntLsb and prevPicOrderCntMsb are derived as follows:

- Let prevTid0Pic be the previous picture in decoding order that has TemporalId equal to 0.
- The variable prevPicOrderCntLsb is set equal to slice_pic_order_cnt_lsb of prevTid0Pic.
- The variable prevPicOrderCntMsb is set equal to PicOrderCntMsb of prevTid0Pic.

If sps_pocs_flag is equal to 1, the following applies:

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

- If the current picture is an IDR picture, PicOrderCntMsb is set equal to 0.

Otherwise, PicOrderCntMsb is derived as follows:


```
if( ( slice_pic_order_cnt_lsb < prevPicOrderCntLsb )  &&
      ( ( prevPicOrderCntLsb − slice_pic_order_cnt_lsb )  >=  ( MaxPicOrderCntLsb / 2 ) ) )
      PicOrderCntMsb = prevPicOrderCntMsb + MaxPicOrderCntLsb                    (8-1)
else if( (slice_pic_order_cnt_lsb > prevPicOrderCntLsb )  &&
      ( ( slice_pic_order_cnt_lsb − prevPicOrderCntLsb ) > ( MaxPicOrderCntLsb / 2 ) ) )
      PicOrderCntMsb = prevPicOrderCntMsb − MaxPicOrderCntLsb
else
      PicOrderCntMsb = prevPicOrderCntMsb
```

PicOrderCntVal is derived as follows:

```
PicOrderCntVal = PicOrderCntMsb + slice_pic_order_cnt_lsb                        (8-2)
```

Otherwise (sps_pocs_flag is equal to 0), the following applies:

1. Let prevPicOrderCntVal be the PicOrderCntVal of prevTid0Pic
2. If TemporalId equals 0, then PicOrderCntVal and DocOffset are derived as follows:

   PicOrderCntVal = prevPicOrderCntVal + SubGopLength

   DocOffset = 0
3. Otherwise (TemporalId is not equal to 0), the following applies:
   - Let prevDocOffset be the DocOffset of the previous picture in decoding order.
   - DocOffset is derived as follows:

     DocOffset = (prevDocOffset + 1) % SubGopLength
   - If DocOffset equals 0, the following applies:

     prevPicOrderCntVal = prevPicOrderCntVal + SubGopLength

     ExpectedTemporalId = 0
   - Otherwise (DocOffset is not equal to 0), the following applies:

     ExpectedTemporalId = 1 + Floor(Log2(DocOffset))
   - PicOrderCntVal is derived as follows:

```
while( TemporalId != ExpectedTemporalId ) {
    DocOffset = ( DocOffset + 1 ) % SubGopLength
    if( DocOffset = = 0)
        ExpectedTemporalId = 0
    else                                                                    (8-3)
        ExpectedTemporalId = 1 + Floor( Log2( DocOffset ) )
}
PocOffset = DocToPoc[ DocOffset ][ SubGopLength ]
PicOrderCntVal = prevPicOrderCntVal + PocOffset
```

Table 8-1 – Specification of DocToPoc[i][j]

i	j					
	1	2	4	8	16	32
0	0	0	0	0	0	0
1	n/a	−1	−2	−4	−8	−16
2	n/a	n/a	−3	−6	−12	−24
3	n/a	n/a	−1	−2	−4	−8
4	n/a	n/a	n/a	−7	−14	−28
5	n/a	n/a	n/a	−5	−10	−20
6	n/a	n/a	n/a	−3	−6	−12
7	n/a	n/a	n/a	−1	−2	−4
8	n/a	n/a	n/a	n/a	−15	−30
9	n/a	n/a	n/a	n/a	−13	−26
10	n/a	n/a	n/a	n/a	−11	−22
11	n/a	n/a	n/a	n/a	-9	−18
12	n/a	n/a	n/a	n/a	−7	−14
13	n/a	n/a	n/a	n/a	−5	−10
14	n/a	n/a	n/a	n/a	−3	−6
15	n/a	n/a	n/a	n/a	−1	−2
16	n/a	n/a	n/a	n/a	n/a	−31
17	n/a	n/a	n/a	n/a	n/a	−29
18	n/a	n/a	n/a	n/a	n/a	−27
19	n/a	n/a	n/a	n/a	n/a	−25
20	n/a	n/a	n/a	n/a	n/a	−23
21	n/a	n/a	n/a	n/a	n/a	−21
22	n/a	n/a	n/a	n/a	n/a	−19
23	n/a	n/a	n/a	n/a	n/a	−17
24	n/a	n/a	n/a	n/a	n/a	−15
25	n/a	n/a	n/a	n/a	n/a	−13
26	n/a	n/a	n/a	n/a	n/a	−11
27	n/a	n/a	n/a	n/a	n/a	-9
28	n/a	n/a	n/a	n/a	n/a	−7
29	n/a	n/a	n/a	n/a	n/a	−5
30	n/a	n/a	n/a	n/a	n/a	−3
31	n/a	n/a	n/a	n/a	n/a	−1

NOTE 1 – All IDR pictures will have PicOrderCntVal equal to 0 since slice_pic_order_cnt_lsb is inferred to be 0 for IDR pictures and prevPicOrderCntLsb and prevPicOrderCntMsb are both set equal to 0.

The value of PicOrderCntVal shall be in the range of -2^{31} to $2^{31} - 1$, inclusive. In one CVS, the PicOrderCntVal values for any two coded pictures shall not be the same.

At any moment during the decoding process, the values of PicOrderCntVal & (MaxLtPicOrderCntLsb − 1) for any two reference pictures in the DPB shall not be the same.

The function PicOrderCnt(picX) is specified as follows:

PicOrderCnt(picX) = PicOrderCntVal of the picture picX (8-4)

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

DiffPicOrderCnt(picA, picB) = PicOrderCnt(picA) − PicOrderCnt(picB) (8-5)

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 -2^{15} to $2^{15} - 1$, 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.

8.3.2 Decoding process for reference picture lists construction

8.3.2.1 Decoding process for reference picture lists construction when sps_rpl_flag is equal to 1

When sps_rpl_flag is equal to 1, this process is invoked at the beginning of the decoding process for each slice 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 slice, no reference picture list is used in decoding of the slice data. When decoding a P slice, only reference picture list 0 (i.e., RefPicList[0]) is used in decoding of the slice data. When decoding a B slice, both reference picture list 0 (i.e., RefPicList[0]) and the reference picture list 1 (i.e., RefPicList[1]) are used in decoding of the slice data.

At the beginning of the decoding process for each slice 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 slice data.

NOTE 1 – For an I slice of a non-IDR picture that it is not the first slice 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 slice that it is not the first slice 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, pocBase = PicOrderCntVal; j < NumEntriesInList[ i ][ SliceRplsIdx[ i ] ]; j++) {
        if( !lt_ref_pic_flag[ i ][ SliceRplsIdx[ i ] ][ j ] ) {
            RefPicPocList[ i ][ j ] = pocBase − DeltaPocSt[ i ][ SliceRplsIdx[ 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"
            pocBase = RefPicPocList[ i ][ j ]                                          (8-6)
        } else {
            if( there is a reference picA in the DPB with PicOrderCntVal & ( MaxLtPicOrderCntLsb − 1 )
                                    equal to FullPocLsbLt[ i ][ SliceRplsIdx[ i ] ][ j ] )
                RefPicList[ i ][ j ] = picA
            else
                RefPicList[ i ][ j ] = "no reference picture"
        }
    }
```

For each i equal to 0 or 1, the following applies:

- 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].
- Each entry in RefPicList[i][j] for j in the range of 0 to NumEntriesInList[i][SliceRplsIdx[i]] − 1, inclusive, is referred to as an STRP entry if lt_ref_pic_flag[i][SliceRplsIdx[i]][j] is equal to 0, and as an LTRP entry otherwise.

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, NumEntriesInList[i] [SliceRplsIdx[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.
- An STRP entry in RefPicList[0] or RefPicList[1] of a slice of a picture and an LTRP entry in RefPicList[0] or RefPicList[1] of the same slice or a different slice 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 2^{24}.
- 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 slices of a picture.

8.3.2.2 Decoding process for reference picture lists construction when sps_rpl_flag is equal to 0

8.3.2.2.1 General

When sps_rpl_flag is equal to 0, this process is invoked at the beginning of the decoding process for each P or B slice:

The reference picture list RefPicList[0] is constructed as follows:

1. The decoding process for filling a reference picture list with lower PicOrderCntVal in clause 8.3.2.2.2 is invoked with i set to 0 and startIdx set to 0, and the output is the variable nextIdx.
2. When nextIdx is less than NumRefIdxActive[0], the decoding process for filling a reference picture list with higher PicOrderCntVal in clause 8.3.2.2.3 is invoked with i set to 0 and startIdx set to nextIdx, and the output is the variable nextIdx.
3. When nextIdx is less than NumRefIdxActive[0], NumRefIdxActive[0] is set equal to nextIdx.

For B slices, the reference picture list RefPicList[1] is constructed as follows:

1. The decoding process for filling a reference picture list with higher PicOrderCntVal in clause 8.3.2.2.3 is invoked with i set to 0 and startIdx set to 0, and the output is the variable nextIdx.
2. When nextIdx is less than NumRefIdxActive[1], the decoding process for filling a reference picture list with lower PicOrderCntVal in clause 8.3.2.2.2 is invoked with i set to 0 and startIdx set to nextIdx, and the output is the variable nextIdx.
3. When nextIdx is less than NumRefIdxActive[1], NumRefIdxActive[1] is set equal to nextIdx.

8.3.2.2.2 Decoding process for filling a reference picture list with lower PicOrderCntVal pictures

Input to this process are:

- a reference picture list identifier i,
- a start index position startIdx.

Output of this process is a modified reference picture list and the variable nextIdx, representing the number of positions filled in the reference picture list.

The varaiable nextIdx is set equal to startIdx.

The varaiable nextTemporalId is set equal to Max(TemporalId – 1, 0).

Let minPoc be set equal to the lowest value of PictureOrderCountVal of all reference pictures in the DPB.

The reference picture list RefPicList[i] is filled with lower PicOrderCntVal pictures as follows:

```
for( j = PicOrderCntVal; j >= minPoc && nextIdx < NumRefIdxActive[ i ]; j-- ) {
    if( there is a reference picture picA in the DPB with PicOrderCntVal equal to j and
        with TemporalId <= nextTemporalId ) {
            RefPicList[ i ][ nextIdx++ ] = picA                                    (8-7)
            nextTemporalId = Max( TemporalId of picA – 1, 0 )
        }
    }
```

8.3.2.2.3 Decoding process for filling a reference picture list with higher PicOrderCntVal

Input to this process are:

– a reference picture list identifier i,

– a start index position startIdx.

Output of this process is a modified reference picture list and the variable nextIdx, representing the number of positions filled in the reference picture list.

The varaiable nextIdx is set equal to startIdx. fd

The varaiable nextTemporalId is set equal to Max(TemporalId – 1, 0).

Let maxPoc be set equal to the highest value of PictureOrderCountVal of all reference pictures in the DPB.

The reference picture list RefPicList[i] is filled with higher PicOrderCntVal pictures as follows:

```
for( j = PicOrderCntVal; j <= maxPoc && nextIdx < NumRefIdxActive[ i ]; j++ ) {
    if( there is a reference picture picA in the DPB with PicOrderCntVal equal to j and
        with TemporalId <= nextTemporalId ) {
            RefPicList[ i ][ nextIdx++ ] = picA                                    (8-8)
            nextTemporalId = Max( TemporalId of picA – 1, 0 )
        }
    }
```

8.3.3 Decoding process for reference picture marking

8.3.3.1 Decoding process for reference picture marking when sps_rpl_flag is equal to 1

When sps_rpl_flag is equal to 1, this process is invoked once per picture, after decoding of a slice header and the decoding process for reference picture list construction for the slice as specified in clause 8.3.2, but prior to the decoding of the slice 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).

When the current picture is an IDR picture, all reference pictures currently in the DPB (if any) are marked as "unused for reference".

STRPs are identified by their PicOrderCntVal values. LTRPs are identified by the log2_max_pic_order_cnt_lsb_minus4 + 4 + additional_lt_poc_lsb LSBs of their PicOrderCntVal values.

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".

8.3.3.2 Decoding process for reference picture marking when sps_rpl_flag is equal to 0

When sps_rpl_flag is equal to 0, this process is invoked for decoded pictures when TemporalId is equal to 0. The process is invoked once, after decoding of a slice header, but prior to the decoding process for reference picture list construction for the slice as specified in clause 8.3.2 and prior to the decoding of the slice data.

Let minPoc be set equal to the lowest value of PictureOrderCountVal of all reference pictures in the DPB.

The variable idx is set equal to 0.

If log2_sub_gop_length is greater than 0, the decoded reference picture marking is performed as follows:

```
for( j = PicOrderCntVal - 1; j >= minPoc; j-- ) {
    if( there is a reference picture picA in the DPB with PicOrderCntVal equal to j) {
        if( TemporalId of picA = = 0 && idx < max_num_tid0_ref_pics ) {
            the picture picA is marked as "used for short-term reference"
            idx++
        } else                                                                    (8-9)
            the picture picA is marked as "unused for reference"
      }
  }
```

Otherwise (log2_sub_gop_length equals 0), the decoded reference picture marking is performed as follows:

```
for( j = PicOrderCntVal - 1; j >= minPoc; j-- ) {
    if( there is a reference picture picA in the DPB with PicOrderCntVal equal to j) {
        picApoc is set equal to the PicOrderCntVal of picA
        if(( picApoc = = PicOrderCntVal – 1 || (picApoc % RefPicGapLength) = = 0 ) &&
            idx < max_num_tid0_ref_pics ) {
            the picture picA is marked as "used for short-term reference"
            idx++                                                                 (8-10)
        } else
            the picture picA is marked as "unused for reference"
      }
  }
```

8.3.4 Decoding process for collocated picture

This process is invoked at the beginning of the decoding process for each P or B slice, after the invocation of the decoding process for reference picture list construction for the slice as specified in clause 8.3.2 , but prior to the decoding of any coding unit. The variable ColPic is set equal to RefPicList[col_pic_list_idx][col_pic_ref_idx].

8.4 Decoding process for coding units coded in intra prediction mode

8.4.1 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 luma coding block relative to the top-left luma sample of the current picture,

– a variable log2CbWidth specifying the width of the current luma coding block,

– a variable log2CbHeight specifying the height of the current luma coding block.

Output of this process is a modified reconstructed block before post-reconstruction and in-loop filtering.

The derivation process for the luma intra prediction mode as specified in clause 8.4.2 is invoked with the luma location (xCb, yCb) and the width of the current coding block log2BlkWidth set to log2CbWidth as inputs and the output is the variable IntraPredModeY[xCb][yCb].

The general decoding process for intra blocks as specified in clause 8.4.4 is invoked with the luma location (xCb, yCb), the variables log2BlkWidth and log2BlkHeight set equal to log2CbWidth and log2CbHeight, 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 block before post-reconstruction filtering.

When the value of sps_htdf_flag is equal to 1, the post-reconstruction filtering process prior to in-loop filtering for a luma component as specified in clause 8.7.6 is invoked with the luma coding block location (xCb, yCb), the variable nCbW set equal to (1 << log2CbWidth), the variable nCbH set equal to (1 << log2CbWidth), and the output is a modified reconstructed block.

When ChromaArrayType is not equal to 0, the following decoding process for chroma samples applies.

The variables log2CbWidthC and log2CbHeightC are set equal to log2CbWidth – (ChromaArrayType = = 3 ? 0 : 1) and log2CbHeight – (ChromaArrayType = = 3 ? 0 : 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) as input, and the output is the variable IntraPredModeC.

The general decoding process for intra blocks as specified in clause 8.4.4 is invoked with the chroma location (xCb / SubWidthC, yCb / SubHeightC), the variables log2BlkWidth and log2BlkHeight set equal to log2CbWidthC and log2CbHeightC, the variable predModeIntra set equal to IntraPredModeC and the variable cIdx set equal to 1 as inputs, and the output is a modified reconstructed picture before in-loop filtering.

The general decoding process for intra blocks as specified in clause 8.4.4 is invoked with the chroma location (xCb / SubWidthC, yCb / SubHeightC), the variables log2BlkWidth and log2BlkHeight set equal to log2CbWidthC and log2CbHeightC, the variable predModeIntra set equal to IntraPredModeC and the variable cIdx set equal to 2 as inputs, and the output is a modified reconstructed picture before in-loop filtering.

8.4.2 Derivation process for luma intra prediction mode

Inputs of this process are:

– a sample location (xCbCmp, yCbCmp) specifying the top-left sample of the current block relative to the top-left sample of the current picture,

– a variable log2BlkWidth specifying the width of the current coding block.

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

When sps_eipd_flag is equal to 0, the following applies:

Table 8-2 specifies the value for the intra prediction mode and the associated names.

Table 8-2 – Specification of intra prediction mode and associated names

Intra prediction mode	Associated name
0	INTRA_DC
1	INTRA_HOR
2	INTRA_VER
3	INTRA_UL
4	INTRA_UR

IntraPredModeY[xCb][yCb] is derived by the intra prediction mode of neighbouring blocks as follows:

IntraPredModeY[xCb][yCb] = IntraPredModeList[intra_pred_mode[xCb][yCb]] (8-11)

Table 8-3 – Derivation of IntraPredModeList[i] from the intra prediction mode of the neighbouring blocks

		IntraPredModeY[xCb][yCb – 1]					
		unavailable	**0**	**1**	**2**	**3**	**4**
IntraPredModeY [xCb – 1][yCb]	**unavailable**	0, 3, 1, 2, 4	0, 2, 1, 3, 4	0, 2, 1, 3, 4	2, 0, 1, 3, 4	0, 2, 1, 3, 4	0, 1, 2, 3, 4
	0	1, 0, 2, 3, 4	0, 1, 2, 3, 4	0, 1, 2, 3, 4	2, 0, 1, 3, 4	0, 1, 3, 2, 4	0, 2, 1, 4, 3
	1	1, 0, 2, 3, 4	1, 0, 2, 3, 4	1, 0, 2, 3, 4	1, 2, 0, 3, 4	1, 0, 3, 2, 4	0, 1, 2, 4, 3
	2	1, 0, 2, 3, 4	0, 2, 1, 3, 4	1, 0, 2, 3, 4	2, 0, 1, 3, 4	0, 1, 2, 3, 4	0, 2, 1, 4, 3
	3	0, 1, 2, 3, 4	0, 3, 2, 1, 4	1, 0, 2, 3, 4	2, 0, 1, 3, 4	3, 0, 1, 2, 4	0, 2, 1, 4, 3
	4	0, 1, 2, 3, 4	0, 1, 2, 4, 3	0, 1, 2, 4, 3	0, 2, 1, 4, 3	0, 1, 2, 3, 4	0, 1, 2, 4, 3

Otherwise (when sps_eipd_flag is equal to 1), the following applies:

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

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. When sps_suco_flag is equal to 1, the neighbouring location (xNbC, yNbC) is set equal to (xCb + (1 << log2BlkWidth), yCb).
2. For X being replaced by either A or B, the variables candIntraPredModeX are derived as follows (when sps_suco_flag is equal to 1, X is replaced by among A, B, or C.):
   - The availability derivation process for a block in z-scan order as specified in clause 6.4.1 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 availableX is equal to FALSE, candIntraPredModeX is set equal to INTRA_DC.
     - Otherwise, candIntraPredModeX is set equal to IntraPredModeY[xNbX][yNbX].
3. When sps_suco_flag is equal to 1, a variable validRight is derived, and candIntraPredModeA and candIntraPredModeB are modified as follows:
   - The variable validRight is set equal to FALSE.
   - If availableC equal to TRUE, the following applies.
     - If both availableA and availableB are equal to TRUE, the following applies.
       - If candIntraPredModeA is equal to candIntraPredModeB, candIntraPredModeB is set equal to candIntraPredModeC.
       - Otherwise, if both candIntraPredModeA and candIntraPredModeB are not equal to candIntraPredModeC, the variable validRight is set equal to TRUE.
     - Otherwise, if availableA is equal to FALSE, candIntraPredModeA is set equal to candIntraPredModeC.
     - Otherwise, if availableB is equal to FALSE, candIntraPredModeB is set equal to candIntraPredModeC.
4. The candModeList[x] with x = 0, 1 is derived as follows:

   candModeList[0] = Min(candIntraPredModeA, candIntraPredModeB) (8-12)

   candModeList[1] = Max(candIntraPredModeA, candIntraPredModeB) (8-13)

   - If candModeList[1] is equal to candModeList[0], the following applies:

     candModeList[0] = INTRA_DC (8-14)

     candModeList[1] = (candModeList[1] = = INTRA_DC) ? INTRA_BI : candModeList[1] (8-15)

5. The extCandModeList[x] with x = 0..7 is derived as follows:
   - When sps_suco_flag is equal to 0 or validRight is equal to FALSE, the following applies:
     - If both candModeList[0] and candModeList[1] are less than 3 (i.e., equal to INTRA_PLN, INTRA_DC, or INTRA_BI), extCandModeList[x] with x = 0..7 is derived as follows:
       - The extCandModeList[0] is a mode which is not included in candModeList[x] with x = 0, 1 among the INTRA_PLN, INTRA_DC, and INTRA_BI modes. The extCandModeList[x] with x = 1..7 is derived as follows:

         extCandModeList[1] = INTRA_VER (8-16)

extCandModeList[2] = INTRA_HOR (8-17)

extCandModeList[3] = INTRA_DIA_R (i.e., INTRA_VER + INTRA_HOR) >> 1) (8-18)

extCandModeList[4] = INTRA_DIA_L (8-19)

extCandModeList[5] = INTRA_DIA_U (8-20)

extCandModeList[6] = INTRA_VER + 4 (8-21)

extCandModeList[7] = INTRA_HOR − 4 (8-22)

– If candModeList[0] is less than 3 and candModeList[1] are equal or greater than 3, extCandModeList[x] with x = 0..7 is derived as follows:

  – If candModeList[0] is equal to INTRA_PLN, extCandModeList[x] with x = 0, 1 is derived as follows:

    extCandModeList[0] = INTRA_BI (8-23)

    extCandModeList[1] = INTRA_DC (8-24)

  – Otherwise, extCandModeList[x] with x = 0, 1 is derived as follows:

    extCandModeList[0] = (candModeList[0] = = INTRA_BI) ? INTRA_DC : INTRA_BI (8-25)

    extCandModeList[1] = INTRA_PLN (8-26)

  – If candModeList[1] is greater than 30, extCandModeList[x] with x = 2..7 is derived as follows:

    extCandModeList[2] = (candModeList[1] = = 32) ? 31 : 32 (8-27)

    extCandModeList[3] = 30 (8-28)

    extCandModeList[4] = 29 (8-29)

    extCandModeList[5] = 28 (8-30)

    extCandModeList[6] = INTRA_HOR (8-31)

    extCandModeList[7] = INTRA_DIA_R (8-32)

  – Otherwise, if candModeList[1] is less than 5, extCandModeList[x] with x = 2..7 is derived as follows:

    extCandModeList[2] = (candModeList[1] = = 3) ? 4 : 3 (8-33)

    extCandModeList[3] = 5 (8-34)

    extCandModeList[4] = 6 (8-35)

    extCandModeList[5] = 7 (8-36)

    extCandModeList[6] = INTRA_VER (8-37)

    extCandModeList[7] = INTRA_DIA_R (8-38)

  – Otherwise, extCandModeList[x] with x = 2..7 is derived as follows:

    extCandModeList[2] = candModeList[1] + 2 (8-39)

    extCandModeList[3] = candModeList[1] − 2 (8-40)

extCandModeList[4] = candModeList[1] + 1 (8-41)

extCandModeList[5] = candModeList[1] − 1 (8-42)

– If candModeList[1] is equal or less than 23 and equal or greater than 13, the following applies:

extCandModeList[6] = candModeList[1] − 5 (8-43)

extCandModeList[7] = candModeList[1] + 5 (8-44)

– Otherwise, the following applies:

extCandModeList[6] = (candModeList[1] > 23) ?
candModeList[1] – 5 : candModeList[1] + 5 (8-45)

extCandModeList[7] = (candModeList[1] > 23) ?
candModeList[1] – 10 : candModeList[1] + 10 (8-46)

– Otherwise, extCandModeList[x] with x = 0..7 is derived as follows:

extCandModeList[0] = INTRA_BI (8-47)

extCandModeList[1] = INTRA_DC (8-48)

– The rest of extCandModeList[x] with x = 3..7 is derived by the following ordered steps:

– A list[y] with y = 0..14 is derived as follows:

list[0] = (candModeList[0] = = 3 | | candModeList[0] = = 4) ?
candModeList[0] + 1 : candModeList[0] – 2 (8-49)

list[1] = (candModeList[0] = = 31) ? candModeList[0] – 1 : candModeList[0] + 2 (8-50)

list[2] = (candModeList[1] = = 4) ? candModeList[1] + 1 : candModeList[1] – 2 (8-51)

list[3] = (candModeList[1] = = 32 | | candModeList[1] = = 31) ?
candModeList[1] – 1 : candModeList[1] + 2 (8-52)

list[4] = (candModeList[0] + candModeList[1] + 1) >> 1 (8-53)

list[5] = (list[4] + candModeList[0] + 1) >> 1 (8-54)

list[6] = (list[4] + candModeList[1] + 1) >> 1 (8-55)

list[7] = INTRA_VER (8-56)

list[8] = INTRA_HOR (8-57)

list[9] = INTRA_DIA_R (8-58)

list[10] = INTRA_PLN (8-59)

list[11] = INTRA_DIA_L (8-60)

list[12] = INTRA_DIA_U (8-61)

list[13] = INTRA_VER + 4 (8-62)

list[14] = INTRA_HOR – 4 (8-63)

- A variable iCount is set to 2.
- For i equal to 0 to 14, inclusive, the following steps are performed.
  - For j equal to 0 to iCount, inclusive, when the list[i] is equal to among the extCandModeList[j], candModeList[0], and candModeList[1], the value of i is incremented by one and j is set to 0. If j is equal to (iCount – 1), then extCandModeList[iCount] is set equal to list[i], i and iCount are increased by one and j is set to 0.
  - If iCount is greater than 7, all the process is terminated.

– Otherwise (when sps_suco_flag is equal to 1 and the validRight is equal to TRUE), the following applies:

- If both candModeList[0] and candModeList[1] are less than 3 (i.e., equal to INTRA_PLANAR, INTRA_DC, or INTRA_BI), extCandModeList[x] with x = 0..7 is derived as follows:
  - The extCandModeList[0] is set to a mode which is not included in the candModeList among the INTRA_PLANAR, INTRA_DC, and INTRA_BI modes.
  - If candIntraPredModeC is less than 3, extCandModeList[x] with x = 1..7 is set to equations 8-16 to 8-22.
  - Otherwise, extCandModeList[x] with x = 1..3 is derived as follows:

extCandModeList[1] = candIntraPredModeC (8-64)

extCandModeList[2] = (candIntraPredModeC = = 3 | | candIntraPredModeC = = 4) ?
candIntraPredModeC + 1 : candIntraPredModeC − 2 (8-65)

extCandModeList[3] = (candIntraPredModeC = = 32 | | candIntraPredModeC = = 31) ?
candIntraPredModeC − 1 : candIntraPredModeC + 2 (8-66)

  - The rest of extCandModeList[x] with x = 4..7 is derived by the following ordered steps:
    - A list[y] with y = 0..9 is derived as follows:

list[0] = INTRA_VER (8-67)

list[1] = INTRA_HOR (8-68)

list[2] = INTRA_DIA_R (8-69)

list[3] = INTRA_PLN (8-70)

list[4] = INTRA_DIA_L (8-71)

list[5] = INTRA_DIA_U (8-72)

list[6] = INTRA_VER + 4 (8-73)

list[7] = INTRA_HOR – 4 (8-74)

list[8] = INTRA_VER – 4 (8-75)

list[9] = INTRA_HOR + 4 (8-76)

    - A variable iCount is set to 4.
    - For i equal to 0 to 9, inclusive, the following steps are performed.
      - For j equal to 0 to iCount, inclusive, when the list[i] is equal to among the extCandModeList[j], candModeList[0], and candModeList[1], the value of i is incremented by one and j is set to 0. If j is equal to (iCount – 1), then

extCandModeList[iCount] is set equal to list[i], i and iCount are increased by one and j is set to 0.

- If iCount is greater than 7, all the process is terminated.

- If candModeList[0] is less than 3 and candModeList[1] are equal or greater than 3, extCandModeList[x] with x = 0..7 is derived as follows:
  - If candIntraPredModeC is less than 3, extCandModeList[x] with x = 0..7 is set to equations 8-23 to 8-46.
  - Otherwise, extCandModeList[x] with x = 0..2 is derived as follows:
    - If candModeList[0] is equal to INTRA_PLN, extCandModeList[x] with x = 0..2 is derived as follows:

      extCandModeList[0] = INTRA_BI (8-77)

      extCandModeList[1] = INTRA_DC (8-78)

      extCandModeList[2] = candIntraPredModeC (8-79)

    - Otherwise, extCandModeList[x] with x = 0..2 is derived as follows:

      extCandModeList[0] = (candModeList[0] = = INTRA_BI) ? INTRA_DC : INTRA_BI (8-80)

      extCandModeList[1] = INTRA_PLN (8-81)

      extCandModeList[2] = candIntraPredModeC (8-82)

    - The rest of extCandModeList[x] with x = 3..7 is derived by the following ordered steps:
      - A list[y] with y = 0..14 is derived as follows:

        list[0] = (candIntraPredModeC = = 3 | | candIntraPredModeC = = 4) ?
        candIntraPredModeC + 1 : candIntraPredModeC − 2 (8-83)

        list[1] = (candIntraPredModeC = = 32 | | candIntraPredModeC = = 31) ?
        candIntraPredModeC − 1 : candIntraPredModeC + 2 (8-84)

        list[2] = (candModeList[1] = = 3 | | candModeList[1] = = 4) ?
        candModeList[1] + 1 : candModeList[1] – 2 (8-85)

        list[3] = (candModeList[1] = = 32 | | candModeList[1] = = 31) ?
        candModeList[1] – 1 : candModeList[1] + 2 (8-86)

        list[4] = (candIntraPredModeC + candModeList[1] + 1) >> 1 (8-87)

        list[5] = (candIntraPredModeC + list[4] + 1) >> 1 (8-88)

        list[6] = (candModeList[1] + list[4] + 1) >> 1 (8-89)

        list[7] = INTRA_VER (8-90)

        list[8] = INTRA_HOR (8-91)

        list[9] = INTRA_DIA_R (8-92)

        list[10] = INTRA_PLN (8-93)

        list[11] = INTRA_DIA_L (8-94)

list[12] = INTRA_DIA_U (8-95)

list[13] = INTRA_VER + 4 (8-96)

list[14] = INTRA_HOR – 4 (8-97)

- A variable iCount is set to 3.
- For i equal to 0 to 14, inclusive, the following steps are performed.
  - For j equal to 0 to iCount, inclusive, when the list[i] is equal to among the extCandModeList[j], candModeList[0], and candModeList[1], the value of i is incremented by one and j is set to 0. If j is equal to (iCount – 1), then extCandModeList[iCount] is set equal to list[i], i and iCount are increased by one and j is set to 0.
  - If iCount is greater than 7, all the process is terminated.

– Otherwise, extCandModeList[x] with x = 0..7 is derived as follows:
  - If candIntraPredModeC is less than 3, extCandModeList[x] with x = 0, 1 is derived as follows:

    extCandModeList[0] = candIntraPredModeC (8-98)

    extCandModeList[1] = (candIntraPredModeC = = INTRA_BI) ? INTRA_DC : INTRA_BI (8-99)

  - The rest of extCandModeList[x] with x = 2..7 is derived by the following ordered steps:
    - A list[x] with x = 0..14 is set to equations 8-49 to 8-63.
    - A variable iCount is set to 2.
    - For i equal to 0 to 14, inclusive, the following steps are performed.
      - For j equal to 0 to iCount, inclusive, when the list[i] is equal to among the extCandModeList[j], candModeList[0], and candModeList[1], the value of i is incremented by one and j is set to 0. If j is equal to (iCount – 1), then extCandModeList[iCount] is set equal to list[i], i and iCount are increased by one and j is set to 0.
      - If iCount is greater than 7, all the process is terminated.
  - Otherwise, extCandModeList[x] with x = 0..2 is derived as follows:

    extCandModeList[0] = INTRA_BI (8-100)

    extCandModeList[1] = INTRA_DC (8-101)

    extCandModeList[2] = candIntraPredModeC (8-102)

  - The rest of extCandModeList[x] with x = 3..7 is derived by the following ordered steps:
    - A list[y] with y = 0..15 is derived as follows:

      list[0] = (candModeList[0] = = 3 | | candModeList[0] = = 4) ?
      candModeList[0] + 1 : candModeList[0] – 2 (8-103)

      list[1] = (candModeList[0] = = 31) ? candModeList[0] – 1 : candModeList[0] + 2 (8-104)

      list[2] = (candModeList[1] = = 4) ? candModeList[1] + 1 : candModeList[1] – 2 (8-105)

      list[3] = (candModeList[1] = = 32 | | candModeList[1] = = 31) ?
      candModeList[0] – 1 : candModeList[0] + 2 (8-106)

list[4] = (candIntraPredModeC = = 3 | | candIntraPredModeC = = 4) ?
candIntraPredModeC + 1 : candIntraPredModeC − 2 (8-107)

list[5] = (candIntraPredModeC = = 32 | | candIntraPredModeC = = 31) ?
candIntraPredModeC − 1 : candIntraPredModeC + 2 (8-108)

list[6] = (candIntraPredModeC < candModeList[1]) ? (candModeList[0] +
condIntraPredModeC + 1) >> 1 : (candModeList[0] + candModeList[1] + 1)
>> 1 (8-109)

list[7] = (candIntraPredModeC < candModeList[0]) ? (candModeList[0] +
candModeList[1] + 1) >> 1 : (candIntraPredModeC + candModeList[1] + 1)
>> 1 (8-110)

list[8] = INTRA_VER (8-111)

list[9] = INTRA_HOR (8-112)

list[10] = INTRA_DIA_R (8-113)

list[11] = INTRA_PLN (8-114)

list[12] = INTRA_DIA_L (8-115)

list[13] = INTRA_DIA_U (8-116)

list[14] = INTRA_VER + 4 (8-117)

list[15] = INTRA_HOR – 4 (8-118)

- A variable iCount is set to 3.
- For i equal to 0 to 15, inclusive, the following steps are performed.
  - For j equal to 0 to iCount, inclusive, when the list[i] is equal to among the extCandModeList[j], candModeList[0], and candModeList[1], the value of i is incremented by one and j is set to 0. If j is equal to (iCount – 1), then extCandModeList[iCount] is set equal to list[i], i and iCount are increased by one and j is set to 0.
  - If iCount is greater than 7, all the process is terminated.

6. The remModeList[x] with x = 0..32 is derived by the following ordered steps:
   - The remModeList[x] with x = 0, 1 is set equal to the candModeList[y] with y = 0, 1.
   - The remModeList[x] with x = 2..9 is set equal to the extCandModeList[y] with y = 0..7.
   - The remModeList[x] with x = 10..32 is derived as follows:
     - A variable iCount is set to 10.
     - A defaultModeList[33] is set to follows: { INTRA_DC, INTRA_BI, INTRA_VER, INTRA_PLN, INTRA_HOR, INTRA_VER – 1, INTRA_VER + 1, INTRA_VER – 2, INTRA_VER + 2, INTRA_VER – 3, INTRA_VER + 3, INTRA_HOR – 1, INTRA_HOR + 1, INTRA_HOR – 2, INTRA_HOR + 2, INTRA_HOR – 3, INTRA_HOR + 3, INTRA_VER + 5, INTRA_VER + 4, INTRA_VER – 5, INTRA_VER – 4, INTRA_DIA_R, INTRA_DIA_L, INTRA_DIA_L – 3, INTRA_DIA_L – 2, INTRA_DIA_L – 1, INTRA_DIA_U, INTRA_DIA_U + 1, INTRA_DIA_U + 2, INTRA_HOR – 4, INTRA_HOR – 5, INTRA_HOR + 5, INTRA_HOR + 4 }
     - For i equal to 0 to 32, inclusive, when the defaultModeList[i] is equal to the candModeList[y] with y = 0, 1 or the extCandModeList[z] with z = 0..7, i is increased by one. Otherwise, remModeList[iCount] is set equal to the defaultModeList[i] and the iCount is increased by one.

7. IntraPredModeY[xCb][yCb] is derived by applying the following procedure:
   - If intra_luma_pred_mpm_flag[x0][y0] is equal to 1, the IntraPredModeY[xCb][yCb] is set equal to candModeList[intra_luma_pred_mpm_idx[x0][y0]].
   - Otherwise, if intra_luma_pred_pims_flag[x0][y0] is equal to 1, the IntraPredModeY[xCb][yCb] is set equal to extCandModeList[intra_luma_pred_pims_idx[x0][y0]].
   - Otherwise, the IntraPredModeY[xCb][yCb] is set equal to remModeList[intra_luma_pred_rem_mode[x0][y0] + 2 + 8].

8.4.3 Derivation process for chroma intra prediction mode

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

Output of this process is the variable IntraPredModeC.

The variable modeIdx is derived using intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb][yCb] as specified in clause 8.4.2.

The chroma intra prediction mode IntraPredModeC is derived as follows:

- If intra_chroma_pred_mode[x0][y0] is equal to 0, IntraPredModeC is equal to IntraPredModeY[xCb][yCb].
- Otherwise, IntraPredModeC is derived as follows:
  - If IntraPredModeY[xCb][yCb] is INTRA_DC, INTRA_HOR, INTRA_VER or INTRA_BI, the following applies:
    - A variable modeIdx is derived as follows:
      - If IntraPredModeY[xCb][yCb] is equal to INTRA_BI, modeIdx is set equal to 1.
      - Otherwise, if IntraPredModeY[xCb][yCb] is equal to INTRA_DC, modeIdx is set equal to 2.
      - Otherwise, if IntraPredModeY[xCb][yCb] is equal to INTRA_HOR, modeIdx is set equal to 3.
      - Otherwise, if IntraPredModeY[xCb][yCb] is equal to INTRA_VER, modeIdx is set equal to 4.
    - If intra_chroma_pred_mode[x0][y0] is equal to or greater than modeIdx, IntraPredModeC is set to equal to intra_chroma_pred_mode[x0][y0] + 1.
    - Otherwise, IntraPredModeC is set to equal to intra_chroma_pred_mode[x0][y0].
  - Otherwise, IntraPredModeC is set to equal to intra_chroma_pred_mode[x0][y0].

Table 8-4 – Specification of IntraPredModeC

IntraPredModeC	chroma intra prediction mode
0	IntraPredModeY[xCb][yCb]
1	INTRA_BI
2	INTRA_DC
3	INTRA_HOR
4	INTRA_VER

8.4.4 Decoding process of intra prediction

8.4.4.1 General decoding process for intra blocks

Inputs to this process are:

- a sample location (xCbCmp, yCbCmp) specifying the top-left sample of the current block relative to the top-left sample of the current picture,
- a variable predModeIntra specifying the intra prediction mode,
- variables log2BlkWidth and log2BlkHeight specifying the width and height of the current coding block,

– a variable cIdx specifying the colour component of the current coding block.

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

The variables nCbW and nCbH are set equal to 1 << log2BlkWidth and 1 << log2BlkHeight.

When sps_suco_flag is equal to 1, the following applies:

The nCbW * 3 + nCbH * 3 + 1 neighbouring samples p[x][y] that are constructed samples after the post-reconstruction filtering process, with x = −1, y = −1..nCbH + nCbW − 1, x = 0..nCbW + nCbH − 1, y = −1, and x = nCbW, y = 0..nCbH + nCbW – 1 are derived as follows:

– The neighbouring location (xNbCmp, yNbCmp) is specified as follows:

(xNbCmp, yNbCmp) = (xCbCmp + x, yCbCmp + y) (8-119)

– The current luma location (xCbY, yCbY) and the neighbouring luma location (xNbY, yNbY) are derived as follows:

(xCbY, yCbY) =
(cIdx == 0) ? (xCbCmp, yCbCmp) : (xCbCmp * SubWidthC, yCbCmp * SubHeightC) (8-120)

(xNbY, yNbY) =
(cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp * SubWidthC, yNbCmp * SubHeightC) (8-121)

– The availability derivation process for a block in z-scan order as specified in clause 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCbY, yCbY) and the neighbouring luma location (xNbY, yNbY) as inputs, and the output is assigned to availableN.

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

  – If one or more of the following conditions are TRUE, the sample p[x][y] is marked as "not available for intra prediction".

    – The variable availableN is equal to FALSE.

    – CuPredMode[xNbX][xNbY] is not equal to MODE_INTRA and constrained_intra_pred_flag is equal to 1.

  – Otherwise, the sample p[x][y] is marked as "available for intra prediction" and the sample at the location (xNbCmp, yNbCmp) is assigned to p[x][y].

– When at least one sample p[x][y] with x = −1, y = −1..nCbH + nCbW – 1 and x = 0..nCbW + nCbH − 1, y = −1 and x = nCbW, y = 0..nCbH + nCbW – 1 is marked as "not available for intra prediction", the reference sample substitution process for intra sample prediction in clause 8.4.4.2 is invoked with the samples p[x][y] with x = −1, y = −1..nCbH + nCbW – 1 and x = 0..nCbW + nCbH − 1, y = −1 and x = nCbW, y = 0..nCbH + nCbW – 1, nCbW, nCbH and cIdx as inputs, and the modified samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 and x = nCbW, y = 0..nCbH + nCbW – 1 as output.

Otherwise, when sps_suco_flag is equal to 0, the following applies:

The nCbW * 2 + nCbH * 2 + 1 neighbouring samples p[x][y] that are constructed samples after the post-reconstruction filtering process, with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1, are derived as follows:

– The neighbouring location (xNbCmp, yNbCmp) is specified as follows:

(xNbCmp, yNbCmp) = (xCbCmp + x, yCbCmp + y) (8-122)

– The current luma location (xCbY, yCbY) and the neighbouring luma location (xNbY, yNbY) are derived as follows:

(xCbY, yCbY) =
(cIdx == 0) ? (xCbCmp, yCbCmp) : (xCbCmp * SubWidthC, yCbCmp * SubHeightC) (8-123)

(xNbY, yNbY) =
(cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp * SubWidthC, yNbCmp * SubHeightC) (8-124)

– The availability derivation process for a block in z-scan order as specified in clause 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCbY, yCbY) and the neighbouring luma location (xNbY, yNbY) as inputs, and the output is assigned to availableN.

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

  – If one or more of the following conditions are TRUE, the sample p[x][y] is marked as "not available for intra prediction".

    – The variable availableN is equal to FALSE.

    – CuPredMode[xNbX][xNbY] is not equal to MODE_INTRA and constrained_intra_pred_flag is equal to 1.

  – Otherwise, the sample p[x][y] is marked as "available for intra prediction" and the sample at the location (xNbCmp, yNbCmp) is assigned to p[x][y].

– When at least one sample p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 is marked as "not available for intra prediction", the reference sample substitution process for intra sample prediction in clause 8.4.4.2 is invoked with the samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1, nCbW, nCbH and cIdx as inputs, and the modified samples p[x][y] with x = −1, y = −1..nCbH + nCbw − 1 and x = 0..nCbW + nCbH − 1, y = −1 as output.

Depending on the value of predModeIntra, the following applies:

– If predModeIntra is equal to INTRA_DC, the corresponding intra prediction mode specified in clause 8.4.4.3 is invoked with the sample array p, the coding block width nCbW and height nCbH as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_HOR, the corresponding intra prediction mode specified in clause 8.4.4.4 is invoked with the sample array p, the coding block width nCbW and height nCbH as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_VER, the corresponding intra prediction mode specified in clause 8.4.4.5 is invoked with the sample array p, the coding block width nCbW and height nCbH as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_UL, the corresponding intra prediction mode specified in clause 8.4.4.6 is invoked with the sample array p, the coding block width nCbW and height nCbH as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_UR, the corresponding intra prediction mode specified in clause 8.4.4.7 is invoked with the sample array p, the coding block width nCbW and height nCbH as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_BI, the corresponding intra prediction mode specified in clause 8.4.4.8 is invoked with the sample array p, the block size nCbW, nCbH, log2BlkWidth, and log2BlkHeight as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_PLN, the corresponding intra prediction mode specified in clause 8.4.4.9 is invoked with the sample array p, the block size nCbW, nCbH, log2BlkWidth, and log2BlkHeight as inputs, and the output is the predicted sample array predSamples.

– Otherwise, the corresponding intra prediction mode specified in clause 8.4.4.10 is invoked with the sample array p, the coding block width nCbW and height nCbH as inputs, and the output is the predicted sample array predSamples.

8.4.4.2 Reference sample substitution process for intra sample prediction

Inputs to this process are:

– reference samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 for intra sample prediction,

– reference samples p[x][y] with x = nCbW, y = 0..nCbH + nCbW – 1 if sps_suco_flag is equal to 1,

– variables nCbW and nCbH specifying the width and height of the current coding block,

– a variable cIdx specifying the colour component of the current coding block.

Outputs of this process are:

– the modified reference samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 for intra sample prediction,

– the modified reference samples p[x][y] with x = nCbW, y = 0..nCbH + nCbW – 1 if sps_suco_flag is equal to 1.

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to $BitDepth_Y$.

– Otherwise, bitDepth is set equal to $BitDepth_C$.

When sps_eipd_flag is equal to 0, the following applies:

The values of the samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 are modified as follows:

– If p[−1][−1] is marked as "not available for intra prediction", the value of 1 << (bitDepth − 1) is is assigned to p[−1][−1].

– The following ordered steps are applied:

  1. Search sequentially starting from x = 0, y = –1 to x = nCbW + nCbH − 1, y = –1. When a sample p[x][y] marked as "not available for intra prediction", the value of 1 << (bitDepth − 1) is assigned to p[x][y].

  2. Search sequentially starting from x = −1, y = 0 to x = −1, y = nCbH + nCbW – 1. When a sample p[x][y] marked as "not available for intra prediction", the value of 1 << (bitDepth − 1) is assigned to p[x][y].

All samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 are marked as "available for intra prediction".

Otherwise (when sps_eipd_flag is equal to 1), the following applies:

When sps_suco_flag is equal to 1, the following applies:

The values of the samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = 0 and x = nCbW, y = −1..nCbH + nCbW – 1 are modified as follows:

– If p[−1][−1] is marked as "not available for intra prediction", the value of 1 << (bitDepth − 1) is assigned to p[−1][−1].

– The following ordered steps are applied:

  1. Search sequentially starting from x = 0, y = –1 to x = nCbW + nCbH − 1, y = –1. When a sample p[x][y] marked as "not available for intra prediction", the value of p[x – 1][y] is assigned to p[x][y].

  2. Search sequentially starting from x = −1, y = 0 to x = −1, y = nCbH + nCbW – 1. When a sample p[x][y] marked as "not available for intra prediction", the value of p[x][y − 1] is assigned to p[x][y].

  3. Search sequentially starting from x = nCbW, y = 0 to x = nCbW, y = nCbH + nCbW – 1. When a sample p[x][y] marked as "not available for intra prediction", the value of p[x][y − 1] is assigned to p[x][y].

All samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 and x = nCbW, y = 0..nCbH + nCbW – 1 are marked as "available for intra prediction".

Otherwise, when sps_suco_flag is equal to 0, the following applies:

The values of the samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 are modified as follows:

– If p[−1][−1] is marked as "not available for intra prediction", the value of 1 << (bitDepth − 1) is is assigned to p[−1][−1].

– The following ordered steps are applied:

  1. Search sequentially starting from x = 0, y = –1 to x = nCbW + nCbH − 1, y = –1. When a sample p[x][y] marked as "not available for intra prediction", the value of p[x – 1][y] is assigned to p[x][y].

  2. Search sequentially starting from x = −1, y = 0 to x = −1, y = nCbH + nCbW – 1. When a sample p[x][y] marked as "not available for intra prediction", the value of p[x][y − 1] is assigned to p[x][y].

All samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbW + nCbH − 1, y = −1 are marked as "available for intra prediction".

8.4.4.3 Specification of intra prediction mode INTRA_DC

Inputs to this process are:

– the neighbouring samples p[x][y], with x = −1, y = 0..nCbH – 1 and x = 0..nCbW − 1, y = −1,

– the neighbouring samples p[x][y], with x = nCbW, y = 0..nCbH – 1 if sps_suco_flag is equal to 1,

– variables nCbW and nCbH specifying the width and height of the current coding block.

– the variable availLR specifying left and right neighbouring blocks' availability of luma coding block.

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

When sps_eipd_flag is equal to 0, the following applies:

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

$$\text{predSamples[x][y]} = \left(\sum_{x'=0}^{\text{nCbW}-1} \text{p}[x'][-1] + \sum_{y'=0}^{\text{nCbH}-1} \text{p}[-1][y'] + \text{nCbW}\right) \gg (\text{Log2}(\text{nCbW}) + 1) \quad (8\text{-}125)$$

Otherwise, if sps_eipd_flag is equal to 1 and sps_suco_flag is equal to 0, the following applies:

– The values of the prediction samples predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH − 1, aspRatioShift = (nCbW > nCbH) ? Log2(nCbH) : Log2(nCbW)),
log2AspRatio = (nCbW > nCbH) ? (Log2(nCbW) − Log2(nCbH)) : (Log2(nCbH) − Log2(nCbW)) are derived as follows:

$$\text{predSamples[x][y]} = \left(\left(\sum_{x'=0}^{\text{nCbW-1}} p[x'][-1] + \sum_{y'=0}^{\text{nCbH-1}} p[-1][y'] + ((\text{nCbW} + \text{nCbH}) \gg 1)\right) \times \text{divScaleMult}[\text{log2AspRatio}]\right) \gg (\text{divScaleShift} + \text{aspRatioShift}) \quad (8\text{-}126)$$

Otherwise, the following applies:

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2. The scaling coefficients divScaleMult [idx] with idx = 0..7 are specified in Table 8-5. The normalization factor divScaleShift = 12.

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

– If availLR is equal to LR_11, the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1, aspRatioShift = (nCbW > 2×nCbH) ? Log2(nCbH)+1 : Log2(nCbW)),
log2AspRatio = (nCbW > 2×nCbH) ? (Log2(nCbW) − Log2(nCbH) − 1) : (Log2(nCbH) − Log2(nCbW) + 1) are derived as follows:

$$\text{predSamples[x][y]} = \left(\left(\sum_{x'=0}^{\text{nCbW-1}} p[x'][-1] + \sum_{y'=0}^{\text{nCbH-1}} p[-1][y'] + \sum_{y'=0}^{\text{nCbH-1}} p[\text{nCbW}][y'] + ((\text{nCbW} + \text{nCbH} + \text{nCbH}) \gg 1)\right) \times \text{divScaleShift}[\text{log2AspRatio}]\right) \gg (\text{divScaleShift} + \text{aspRatioShift}) \quad (8\text{-}127)$$

– Otherwise, if availLR is equal to LR_01, the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1, aspRatioShift = (nCbW > nCbH) ? Log2(nCbH) : Log2(nCbW)),
log2AspRatio = (nCbW > nCbH) ? (Log2(nCbW) − Log2(nCbH)) : (Log2(nCbH) − Log2(nCbW)) are derived as follows:

$$\text{predSamples[x][y]} = \left(\left(\sum_{x'=0}^{\text{nCbW-1}} p[x'][-1] + \sum_{y'=0}^{\text{nCbH-1}} p[\text{nCbW}][y'] + ((\text{nCbW} + \text{nCbH}) \gg 1)\right) \times \text{divScaleMult}[\text{log2AspRatio}]\right) \gg (\text{divScaleShift} + \text{aspRatioShift}) \quad (8\text{-}128)$$

– Otherwise, if availLR is equal to LR_10, the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1, aspRatioShift = (nCbW > nCbH) ? Log2(nCbH) : Log2(nCbW)), log2AspRatio = (nCbW > nCbH) ? (Log2(nCbW) − Log2(nCbH)) : (Log2(nCbH) − Log2(nCbW)) are derived as follows:

$$\text{predSamples[x][y]} = \left(\left(\sum_{x'=0}^{\text{nCbW-1}} p[x'][-1] + \sum_{y'=0}^{\text{nCbH-1}} p[-1][y'] + ((\text{nCbW} + \text{nCbH}) \gg 1)\right) \times \text{divScaleMult}[\text{log2AspRatio}]\right) \gg (\text{divScaleShift} + \text{aspRatioShift}) \quad (8\text{-}129)$$

– Otherwise, (availLR is equal to LR_00), the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1 are derived as follows:

$$\text{predSamples[x][y]} = \left(\sum_{x'=0}^{\text{nCbW}-1} \text{p}[x'][-1] + (\text{nCbW} \gg 1)\right) \gg \text{Log2}(\text{nCbW}) \quad (8\text{-}130)$$

Table 8-5 – Specification of divScaleMult [idx] for various input values of idx

idx	0	1	2	3	4	5	6	7
divScaleMult [idx]	2048	1365	819	455	241	124	63	32

8.4.4.4 Specification of intra prediction mode INTRA_HOR

Inputs to this process are:

– the neighbouring samples p[x][y] with x = −1, y = 0..nCbH – 1,

– the neighbouring samples p[x][y] with x = nCbW, y = 0..nCbH – 1 if sps_suco_flag is equal to 1,

– variables nCbW and nCbH specifying the width and height of the current coding block.

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

When sps_suco_flag is equal to 1, the following applies:

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2. The scaling coefficients divScaleMult [idx] with idx = 0..7 are specified in Table 8-5. The normalization factor divScaleShift = 12.

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

– If availLR is equal to LR_11, the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1 are derived as follows:

predSamples[x][y] = predSamples[x][y] = ((p[−1][y] * (nCbW – y) + p[nCbW][y] * (y + 1) + (nCbW >> 1)) × divScaleMult [Log2(nCbW)]) >> divScaleShift (8-131)

– Otherwise, if availLR is equal to LR_01, the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1 are derived as follows:

predSamples[x][y] = p[nCbW][y] (8-132)

– Otherwise (if availLR is equal to LR_00 or LR_10), the values of the predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1 are derived as follows:

predSamples[x][y] = p[−1][y] (8-133)

Otherwise, when sps_suco_flag is equal to 0, the values of the prediction samples predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH − 1, are derived as follows:

predSamples[x][y] = p[−1][y] (8-134)

8.4.4.5 Specification of intra prediction mode INTRA_VER

Inputs to this process are:

– the neighbouring samples p[x][y] with x = 0..nCbW − 1, y = −1,

– variables nCbW and nCbH specifying the width and height of the current coding block.

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

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

predSamples[x][y] = p[x][−1] (8-135)

8.4.4.6 Specification of intra prediction mode INTRA_UL

Inputs to this process are:

– the neighbouring samples p[x][y], with x = −1, y = −1..nCbH − 1 and x = 0..nCbW − 1, y = −1,

– variables nCbW and nCbH specifying the width and height of the current coding block.

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

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

– if y is greater than x, the following applies:

predSamples[x][y] = p[−1][y − x − 1] (8-136)

– Otherwise, the following applies:

predSamples[x][y] = p[x – y – 1][−1] (8-137)

8.4.4.7 Specification of intra prediction mode INTRA_UR

Inputs to this process are:

– the neighbouring samples p[x][y], with x = −1, y = −1..nCbH + nCbW − 1 and x = 0.. nCbH + nCbW − 1, y = −1,

– variables nCbW and nCbH specifying the width and height of the current coding block.

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

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

predSamples[x][y] = predSamples[x + y + 1][−1] + predSamples[−1][x + y + 1] (8-138)

8.4.4.8 Specification of intra prediction mode INTRA_BI

Inputs to this process are:

– the neighbouring samples p[x][y], with x = −1, y = −1..nCbH − 1 and x = 0..nCbW − 1, y = −1,

– the neighbouring samples p[x][y] with x = nCbW, y = 0..nCbH if sps_suco_flag is equal to 1,

– variables nCbW and nCbH specifying the width and height of the current coding block,

– variables log2BlkWidth and log2BlkHeight specifying the width and height of the current coding block.

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

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to $BitDepth_Y$.

– Otherwise, bitDepth is set equal to $BitDepth_C$.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2. The scaling coefficients divScaleMult [idx] with idx = 0..7 are specified in Table 8-5. The normalization factor divScaleShift = 12.

Depending on the value of sps_suco_flag, the following applies:

– If sps_suco_flag is equal to 1, and availLR is equal to LR_11 or LR_01, the follows applies:

  – If availLR is equal to LR_11, the values of the prediction samples predSamples[x][y] are derived as follows:

$$\begin{aligned}predSamples[\,x\,][\,y\,] = Clip3(\,0, (\,1 << bitDepth\,) - 1,\\ (\,(\,(\,p[\,-1\,][\,y\,] \times (\,nCbW - x\,) + p[\,nCbW\,][\,y\,] \times (\,x + 1\,) + (nCbW >> 1\,)\,) \times\\ divScaleMult\,[\,Log2(nCbW)\,]\,) >> divScaleShift\,) +\\ (\,(\,p[\,x\,][\,-1\,] \times (\,nCbH - 1 - y\,) + (\,(\,p[\,-1\,][\,nCbH - 1\,] \times (\,nCbW - x\,) +\\ p[\,nCbW\,][\,nCbH - 1\,] \times (\,x + 1\,) + (nCbW >> 1\,)\,) \times divScaleMult\,[\,Log2(nCbW)\,]\,) >>\\ divScaleShift) \times (\,y + 1\,) + (\,nCbH >> 1\,)\,) >> Log2(nCbH)\,) + 1\,) >> 1\,)\end{aligned} \tag{8-139}$$

  – Otherwise, if availLR is equal to LR_01, the values of the prediction samples predSamples[x][y] are derived as follows:

    – The variables iA, iB, iC are specified as follows:

$$iA = p[\,-1\,][\,-1\,] \tag{8-140}$$

$$iB = p[\,nCbW\,][\,nCbH\,] \tag{8-141}$$

    – If nCbW is equal to nCbH, the variable iC is derived as follows:

$$iC = (\,iA + iB + 1\,) >> 1 \tag{8-142}$$

    – Otherwise, the variable iC is derived as follows:

$$iShift = Log2(\,Min(\,nCbW, nCbH\,)\,) \tag{8-143}$$

$$\begin{aligned}iC = (\,(\,(\,iA << log2BlkWidth\,) + (\,iB << log2BlkHeight\,)\,) \times 13 +\\ (\,1 << (\,iShift + 5\,)\,)\,) >> (\,iShift + 6\,)\end{aligned} \tag{8-144}$$

  – The values of the prediction samples predSamples[x][y] are derived as follows:

$$\begin{aligned}predSamples[\,x\,][\,y\,] = Clip3(\,0, (\,1 << bitDepth\,) - 1,\\ (\,(\,(\,(\,(\,iA - p[\,nCbW\,][\,y\,]\,) \times (\,x + 1\,)\,) << log2BlkHeight\,) + (\,(\,(\,iB - p[\,x + 1\,][\,-1\,]\,) \times\\ (\,y + 1\,)\,) << log2BlkWidth\,) + (\,(\,p[\,x\,][\,-1\,] + p[\,nCbW\,][\,y\,]\,) << (\,log2BlkWidth +\\ log2BlkHeight\,)\,) + (\,(\,iC << 1\,) - iA - iB\,) \times x \times y + (\,1 << (\,log2BlkWidth +\\ log2BlkHeight\,)\,)\,) >> (\,log2BlkWidth + log2BlkHeight + 1\,)\,)\end{aligned} \tag{8-145}$$

– Otherwise (sps_suco_flag is equal to 0 or availLR is equal to LR_10 or LR_00), the values of the prediction samples predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH − 1, are derived as follows:

  – The variables iA, iB, iC are specified as follows:

$$iA = p[\,nCbW\,][\,-1\,] \tag{8-146}$$

$$iB = p[\,-1\,][\,nCbH\,] \tag{8-147}$$

  – If nCbW is equal to nCbH, the variable iC is derived as follows:

$$iC = (\,iA + iB + 1\,) >> 1 \tag{8-148}$$

  – Otherwise, the variable iC is derived as follows:

$$iShift = Log2(\,Min(\,nCbW, nCbH\,)\,) \tag{8-149}$$

$$\begin{aligned}iC = (\,(\,(\,iA << log2BlkWidth\,) + (\,iB << log2BlkHeight\,)\,) \times 13 + (\,1 << (\,iShift + 5\,)\,)\,) >>\\ (\,iShift + 6\,)\end{aligned} \tag{8-150}$$

  – The values of the prediction samples predSamples[x][y] are derived as follows:

predSamples[x][y] = Clip3(0, (1 << bitDepth) – 1,
(((((iA – p[–1][y]) × (x + 1)) << log2BlkHeight) + (((iB – p[x + 1][−1]) × (y + 1))
<< log2BlkWidth) + ((p[x][–1] + p[–1][y]) << (log2BlkWidth + log2BlkHeight)) +
((iC << 1) – iA – iB) × x × y + (1 << (log2BlkWidth + log2BlkHeight))) >>
(log2BlkWidth + log2BlkHeight + 1)) (8-151)

8.4.4.9 Specification of intra prediction mode INTRA_PLN

Inputs to this process are:

– the neighbouring samples p[x][y] with x = −1, y = −1..nCbH − 1 and x = 0..nCbW, y = −1,

– the neighbouring samples p[x][y] with x = nCbW, y = 0..nCbH – 1 if sps_suco_flag is equal to 1,

– variables nCbW and nCbH specifying the width and height of the current coding block,

– variables log2BlkWidth and log2BlkHeight specifying the width and height of the current coding block.

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

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to $BitDepth_Y$.

– Otherwise, bitDepth is set equal to $BitDepth_C$.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

Depending on the value of sps_suco_flag, the following applies:

– If sps_suco_flag is equal to 1, and availLR is equal to LR_11 or LR_01, the values of the prediction samples predSamples[x][y] are derived as follows:

$$iH = \left(\sum_{x'=0}^{(nCbW \gg 1)-1}(x'+1) \times \left(p[(nCbW \gg 1)+x'+1][-1] - p[(nCbW \gg 1)-x'-1][-1]\right)\right) \quad (8\text{-}152)$$

$$iV = \left(\sum_{y'=0}^{(nCbH \gg 1)-1}(y'+1) \times \left(p[nCbW][(nCbH \gg 1)+y'] - p[nCbW][(nCbH \gg 1)-y'-2]\right)\right) \quad (8\text{-}153)$$

iA = (p[0][–1] + p[nCbW][nCbH – 1]) << 4 (8-154)

iB = ((iH << 5) × mult[log2BlkWidth – 2] + (1 << (shift[log2BlkWidth – 2] – 1))) >>
shift[log2BlkWidth – 2] (8-155)

iC = ((iV << 5) × mult[log2BlkHeight – 2] + (1 << (shift[log2BlkHeight – 2] – 1))) >>
shift[log2BlkHeight – 2] (8-156)

predSamples[x][y] =
Clip(((iA + (x – ((nCbW >> 1) – 1)) × iB + (y – ((nCbH >> 1) – 1)) × iC + 16) >> 5) (8-157)

– Otherwise (sps_suco_flag is equal to 0 or availLR is equal to LR_10 or LR_00), the values of the prediction samples predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH − 1, are derived as follows:

$$iH = \left(\sum_{x'=0}^{(nCbW \gg 1)-1}\left(x'+1\right) \times \left(p\left[(nCbW \gg 1)+x'\right][-1] - p\left[(nCbW \gg 1)-x'-2\right][-1]\right)\right) \quad (8\text{-}158)$$

$$iV = \left(\sum_{y'=0}^{(nCbH \gg 1)-1}\left(y'+1\right) \times \left(p[-1]\left[(nCbH \gg 1)+y'\right] - p[-1]\left[(nCbH \gg 1)-y'-2\right]\right)\right) \quad (8\text{-}159)$$

iA = (p[nCbW – 1][–1] + p[–1][nCbH – 1]) << 4 (8-160)

iB = ((iH << 5) × mult[log2BlkWidth – 2] + (1 << (shift[log2BlkWidth – 2] – 1))) >>

shift[log2BlkWidth – 2] (8-161)

iC = ((iV << 5) × mult[log2BlkHeight – 2] + (1 << (shift[log2BlkHeight – 2] – 1))) >>

shift[log2BlkHeight – 2] (8-162)

predSamples[x][y] =

Clip(((iA + (x – ((nCbW >> 1) – 1)) × iB + (y – ((nCbH >> 1) – 1)) × iC + 16) >> 5) (8-163)

Table 8-6 – Specification of mult[i] and shift[i]

i	2	3	4	5	6
mult[i]	13	17	5	11	23
shift[i]	7	10	11	15	19

8.4.4.10 Specification of directional intra prediction modes

Inputs to this process are:

– the intra prediction mode predModeIntra,

– the neighbouring samples p[x][y] with x = −1, y = −1..nCbH + nCbW − 1 and x = 0..nCbH + nCbW − 1, y = −1,

– the neighbouring samples p[x][y] with x = nCbW, y = 0..nCbH + nCbW – 1 if sps_suco_flag is equal to 1,

– variables nCbW and nCbH specifying the width and height of the current coding block.

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

The value of the prediction sample predSamples[x][y], with x = 0..nCbW − 1, y = 0..nCbH – 1 are derived by the following ordered steps:

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

1. The values iOffset, iX, iY and refPosition are specified as follows:

– Depending on the values sps_suco_flag and availLR, the following applies:

  – If sps_suco_flag is equal to 1, and availLR is equal to LR_01 or LR_11, Tthe following applies:

    iTanY = (((y + 1) × divDxy) >> 10) (8-164)

    iTanX = (((x + 1) × divDyx) >> 10) (8-165)

  – If predModeIntra is less than INTRA_VER, the following applies:

    – If x is less than nCbW – iTanY (refer to upper pixel), the following applies:

      iOffset = (((y + 1) × divDxy) >> 5) – (iTanY << 5) (8-166)

      iX = x + iTanY (8-167)

      refPosition = refUp (8-168)

    – Otherwise, (refer to right pixel), the following applies:

      iOffset = (((nCbW – x) × divDyx) >> 5) – ((((nCbW – x) × divDyx) >> 10) << 5) (8-169)

      iX = x + (((nCbW – x) × divDyx) >> 10) (8-170)

      refPosition = refRight (8-171)

  – Otherwise, if predModeIntra is greater than INTRA_HOR, the following applies:

    iTanX = (((nCbW – x) × divDyx) >> 10) (8-172)

    iTanY = (((nCbW – x) × divDxy) >> 10) (8-173)

– If y is greater than iTanX (refer to right pixel), the following applies:

iOffset = (((nCbW – x) × divDxy) >> 5) – (iTanX << 5) (8-174)

iY = y – iTanX (8-175)

refPosition = refRight (8-176)

– Otherwise, (refer to left pixel), the following applies:

iOffset = (((nCbW – x) × divDxy) >> 5) – (iTanY << 5) (8-177)

iX = x + iTanY (8-178)

refPosition = refLeft (8-179)

– Otherwise, the following applies:

– If y is less than iTanX or avail_LR is equal to LR_00 or LR_01 (refer to upper pixel), the following applies:

iOffset = (((y + 1) × divDxy) >> 5) – (iTanY << 5) (8-180)

iX = x – iTanY (8-181)

refPosition = refUp (8-182)

– Otherwise if avail_LR is equal to LR_01 (refer to right pixel), the following applies:

iOffset = (((nCbW – x) × divDxy) >> 5) – ((((nCbW – x) × divDyx) >> 10) << 5) (8-183)

iY = y + (((nCbW – x) × divDyx) >> 10) (8-184)

refPosition = refRight (8-185)

– Otherwise (refer to left pixel), the following applies:

iOffset = (((x + 1) × divDyx) >> 5) – (iTanX << 5) (8-186)

iY = y – iTanX (8-187)

refPosition = refLeft (8-188)

– Otherwise (sps_suco_flag is equal to 0 or availLR is equal to LR_10 or LR_00, Tthe following applies:

iTanY = (((y + 1) × divDxy) >> 10) (8-189)

iTanX = (((x + 1) × divDyx) >> 10) (8-190)

– If predModeIntra is less than INTRA_VER (refer to upper pixel), the following applies:

iOffset = (((y + 1) × divDxy) >> 5) – (iTanY << 5) (8-191)

iX = x + iTanY (8-192)

refPosition = refUp (8-193)

– Otherwise, if predModeIntra is greater than INTRA_HOR (refer to left pixel), the following applies:

iOffset = (((x + 1) × divDyx) >> 5) – (iTanX << 5) (8-194)

iY = y + iTanX (8-195)

refPosition = refLeft (8-196)

- Otherwise, the following applies:
  - If y is less than iTanX (refer to upper pixel), the following applies:

    iOffset = (((y + 1) × divDxy) >> 5) – (iTanY << 5) (8-197)

    iX = x – iTanY (8-198)

    refPosition = refUp (8-199)

  - Otherwise (refer to left pixel), the following applies:

    iOffset = (((x + 1) × divDyx) >> 5) – (iTanX << 5) (8-200)

    iY = y – iTanX (8-201)

    refPosition = refLeft (8-202)

2. The value of the prediction samples predSamples[x][y] are derived as follows:
   - If refPosition is equal to refUp (refer to upper pixel), then the following applies:
     - A variable clipMax is set to (nCbW + nCbH – 1) .
     - A variable clipMin is set to –1.
     - If dirXYSign is equal to –1, then the following applies:

       iXn = Clip3(clipMin, clipMax, iX + 1),

       iXnP2 = Clip3(clipMin, clipMax, iX + 2),

       iXnN1 = Clip3(clipMin, clipMax, iX – 1) (8-203)

     - Otherwise, the following applies:

       iXn = Clip3(clipMin, clipMax, iX – 1),

       iXnP2 = Clip3(clipMin, clipMax, iX – 2),

       iXnN1 = Clip3(clipMin, clipMax, iX + 1) (8-204)

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

       predSamples[x][y] = (p[iXnN1][–1] × (32 – iOffset) + p[iX][–1] × (64 – iOffset) + p[iXn][–1] × (32 + iOffset) + p[iXnP2][–1] × iOffset + 64) >> 7 (8-205)

   - Otherwise, if refPosition is equal to refLeft (refer to left pixel), then the following applies:
     - A variable clipMax is set to (nCbW + nCbH – 1) .
     - A variable clipMin is set to –1.
     - If dirXYSign is equal to –1, then the following applies:

       iYn = Clip3(clipMin, clipMax, iY + 1),

       iYnP2 = Clip3(clipMin, clipMax, iY + 2),

       iYnN1 = Clip3(clipMin, clipMax, iY + 2) (8-206)

– Otherwise, the following applies:

iYn = Clip3(clipMin, clipMax, iY – 1),

iYnP2 = Clip3(clipMin, clipMax, iY – 2),

iYnN1 = Clip3(clipMin, clipMax, iY + 1) (8-207)

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

predSamples[x][y] = (p[–1][iYnN1] × (32 – iOffset) + p[–1][iY] × (64 – iOffset) +
p[–1][iYn] × (32 + iOffset) + p[–1][iYnP2] × iOffset + 64) >> 7 (8-208)

− Otherwise (refPosition is equal to refRight), then the following applies:

– A variable clipMax is set to (nCbW + nCbH – 1) .

– A variable clipMin is set to –1.

– If dirXYSign is equal to –1, then the following applies:

iYn = Clip3(clipMin, clipMax, iY – 1),

iYnP2 = Clip3(clipMin, clipMax, iY – 2),

iYnN1 = Clip3(clipMin, clipMax, iY + 1) (8-209)

– Otherwise, the following applies:

iYn = Clip3(clipMin, clipMax, iY + 1),

iYnP2 = Clip3(clipMin, clipMax, iY + 2),

iYnN1 = Clip3(clipMin, clipMax, iY – 1) (8-210)

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

predSamples[x][y] = (p[nCbW][iYnN1] × (32 – iOffset) + p[nCbW][iY] × (64 – iOffset)
+ p[nCbW][iYn] × (32 + iOffset) + p[nCbW][iYnP2] × iOffset + 64) >> 7 (8-211)

Table 8-7 – Specification of dirDx, dirDy, dirXYSign, divDxy and divDyx

predModeIntra	dirDx	dirDy	dirXYSign	divDxy	divDyx
3	11	−4	−1	2816	372
4	2	−1	−1	2048	512
5	11	−8	−1	1408	744
6	1	−1	−1	1024	1024
7	8	−11	−1	744	1408
8	1	−2	−1	512	2048
9	4	−11	−1	372	2816
10	1	−4	−1	256	4096
11	1	−8	−1	128	8192
12	-	-	-	-	-
13	1	8	1	128	8192
14	1	4	1	256	4096
15	4	11	1	372	2816
16	1	2	1	512	2048

17	8	11	1	744	1408
18	1	1	1	1024	1024
19	11	8	1	1408	744
20	2	1	1	2048	512
21	11	4	1	2816	372
22	4	1	1	4096	256
23	8	1	1	8192	128
24	-	-	-	-	-
25	8	−1	−1	8192	128
26	4	−1	−1	4096	256
27	11	−4	−1	2816	372
28	2	−1	−1	2048	512
29	11	−8	−1	1408	744
30	1	−1	−1	1024	1024
31	8	−11	−1	744	1408
32	1	−2	−1	512	2048

8.5 Decoding process for coding units coded in inter prediction mode

8.5.1 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 luma coding block relative to the top-left luma sample of the current picture,

– two variables log2CbWidth and log2CbHeight specifying the width and the height of the current luma coding block.

Output of this process is a modified reconstructed block before post-reconstruction filtering.

The derivation process for quantization parameters as specified in clause 8.7.1 is invoked with the luma location (xCb, yCb) as input.

The variables nCbW and nCbH are set equal to 1 << log2CbWidth and 1 << log2CbHeight, respectively. The variables $nCbW_L$ and $nCbH_L$ are set equal to 1 << log2CbWidth and 1 << log2CbHeight, repectively. When ChromaArrayType is not equal to 0, the variable $nCbW_C$ is set equal to (1 << log2CbWidth) / SubWidthC and the variable $nCbH_C$ is set equal to (1 << log2CbHeight) / SubHeightC.

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

1. If affine_flag[xCb][yCb] is equal to 1, The derivation process for affine motion vector components and reference indices as specified in clause 8.5.3 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 numSbXLX and in vertical direction numSbYLX, the prediction list utilization flags predFlagLX, the luma motion vector array mvLX[xSbIdx][ySbIdx], and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXLX − 1, and ySbIdx = 0..numSbYLX − 1, with X being 0 or 1 and the number of control point motion vectors numCpMv and the control point motion vectors cpMvL0[cpIdx] with cpIdx = 0..numCpMv – 1, cpMvL1[cpIdx] with cpIdx = 0..numCpMv – 1 as outputs
2. Otherwise, the derivation process for motion vector components and reference indices as specified in clause 8.5.2 is invoked with the luma coding block location (xCb, yCb), the luma coding block width nCbW, and the luma coding block height nCbH as inputs, and the luma motion vectors mvL0 and mvL1, and the DMVR utilization flag dmvr_applied_flag, when ChromaArrayType is not equal to 0, the chroma motion vector mvCL0 and mvCL1,

the reference indices refIdxL0 and refIdxL1 and the prediction list utilization flags predFlagL0 and predFlagL1 as outputs.

3. If dmvr_applied_flag is equal to 1,

   The reference picture consisting of an ordered two-dimensional array $refPicLX_L$ of luma samples and two ordered two-dimensional arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$ of chroma samples is derived by invoking the process specified in clause 8.5.4.1 with X and refIdxLX as inputs.

   – If either CbWidth is greater than 16 or CbHeight is greater than 16, the coded block is partitioned into sub-blocks with the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, and the sub-block width sbWidth and the sub-block height sbHeight are computed as follows:

   – 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:

   $$numSbX = (cbWidth > 16) \ ? \ (cbWidth >> 4) : 1$$

   $$numSbY = (cbHeight > 16) \ ? \ (cbHeight >> 4) : 1$$

   $$sbWidth = (cbWidth > 16) \ ? \ 16 : cbWidth$$

   $$sbHeight = (cbHeight > 16) \ ? \ 16 : cbHeight$$

   – 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:

     $$mvLX[xSbIdx][ySbIdx] = mvLX[0][0]$$

     $$predFlagLX[xSbIdx][ySbIdx] = predFlagLX[0][0]$$

     $$xSb[xSbIdx][ySbIdx] = xCb + xSbIdx * sbWidth$$

     $$ySb[xSbIdx][ySbIdx] = yCb + ySbIdx * sbHeight$$

     – The decoder side motion vector refimenent process specified in clause 8.5.5 is invoked with xSb[xSbIdx][ySbIdx], ySb[xSbIdx][ySbIdx], sbWidth, sbHeight, the motion vectors mvLX[xSbIdx][ySbIdx] and the reference picture array $refPicLX_L$ as inputs and delta motion vector dMvL0[xSbIdx][ySbIdx] as output.

     – When ChromaArrayType is not equal to 0, the derivation process for chroma motion vectors in clause 8.5.2.14 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 ChromaArrayType is not equal to 0 and predFlagLX[0][0], with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.14 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.

4. 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 dmvr_applied_flag is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.14 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] + dMvL0[xSbIdx][ySbIdx] if X is equal to 0 and refMvLX[xSbIdx][ySbIdx] = mvLX[xSbIdx][ySbIdx] - dMvL0[xSbIdx][ySbIdx] if X is equal to 1.

refMvLX[xSbIdx][ySbIdx][0] = Clip3(-2^{17}, $2^{17} - 1$, refMvLX[xSbIdx][ySbIdx][0])

refMvLX[xSbIdx][ySbIdx][1] = Clip3(-2^{17}, $2^{17} - 1$, refMvLX[xSbIdx][ySbIdx][1])

– Otherwise (dmvr_applied_flag is equal to 0), the following applies:

refMvLX[xSbIdx][ySbIdx] = mvLX[xSbIdx][ySbIdx]

refMvCLX [xSbIdx][ySbIdx] = mvCLX[xSbIdx][ySbIdx]

NOTE – The array refMvLX is used in the derivation process for collocated motion vectors in clause 8.5.2.12. The array of non-refined luma motion vectors MvLX is used in deblocking boundary strength derivation processes. refMvLX is used in the derivation process for spatial motion vectors if the neighbor coding block is in the coding tree block that is above or above left of the current coding tree block. mvLX is used in the derivation process for spatial motion vectors if the neighbor block is in the coding tree block that is left or above right of the current coding tree block or if the current coding block and neighbor coding block are in the same coding tree block.

5. If affine_flag[xCb][yCb] is equal to 0 and dmvr_applied_flag is equal to 0, the following applies:

numSbXL0 = 1 (8-212)

numSbYL0 = 1 (8-213)

numSbXL1 = 1 (8-214)

numSbYL1 = 1 (8-215)

mvL0[0][0] = mvL0 << 2 (8-216)

mvL1[0][0] = mvL1 << 2 (8-217)

mvCL0[0][0] = mvL0 << 2 (8-218)

mvCL1[0][0] = mvL1 << 2 (8-219)

numCpMv = 2 (8-220)

cpMvL0[0] = mvL0 (8-221)

cpMvL0[1] = mvL0 (8-222)

cpMvL1[0] = mvL1 (8-223)

cpMvL1[1] = mvL1 (8-224)

6. The decoding process for inter sample prediction as specified in clause 8.5.4 is invoked with the luma coding block location (xCb, yCb), the luma coding block width nCbW, the luma coding block height nCbH, the number of luma coding subblocks in horizontal direction numSbXL0 and numSbXL1, and in vertical direction numSbYL0 and numSbYL1, the luma motion vectors mvL0[xSbIdx][ySbIdx] , and the refined luma motion vectors refMvL0[xSbIdx][ySbIdx] and refMvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXL0 − 1, and ySbIdx = 0..numSbYL0 − 1 and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXL1 − 1, and ySbIdx = 0..numSbYL1 − 1, when ChromaArrayType is not equal to 0, the chroma motion vectors mvCL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXL0 − 1, and ySbIdx = 0..numSbYL0 − 1 and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXL1 − 1, and ySbIdx = 0..numSbYL1 − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1, the number of control point motion vectors numCpMv and the control point motion vectors cpMvL0[cpIdx] with cpIdx = 0..numCpMv – 1, cpMvL1[cpIdx] with cpIdx = 0..numCpMv – 1 as inputs, and the inter prediction samples (predSamples) that are an ($nCbW_L$)x($nCbH_L$) array $predSamples_L$ of prediction luma samples and, when ChromaArrayType is not equal to 0, two ($nCbW_C$)x($nCbH_C$) arrays $predSamples_{Cr}$ and $predSamples_{Cr}$ of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

7. The decoding process for the residual signal of coding units coded in inter prediction mode specified in clause 8.5.6 is invoked with the luma location (xCb, yCb) and the luma coding block width log2CbWidth and the luma

coding block height log2CbHeight as inputs, and the outputs are three arrays $resSamples_L$, $resSamples_{Cb}$, and $resSamples_{Cr}$.

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

   – The picture reconstruction process prior to post-reconstruction and in-loop filtering for a colour component as specified in clause 8.7.5 is invoked with the luma coding block location (xCb, yCb), the variable nCurrW set equal to $nCbW_L$, the variable nCurrH set equal to $nCbH_L$, the variable cIdx set equal to 0, the $(nCbW_L)x(nCbH_L)$ array predSamples set equal to $predSamples_L$, and the $(nCbW_L)x(nCbH_L)$ array resSamples set equal to $resSamples_L$ as inputs.

   – When value of cbf_luma is equalt to 1and sps_htdf_flag is equal to 1, the post-reconstruction filtering process prior to in-loop filtering for a luma component as specified in clause 8.7.6 is invoked with the luma coding block location (xCb, yCb), the variable nCurrW set equal to $nCbW_L$, the variable nCurrH set equal to $nCbH_L$, and the $(nCbW_L)x(nCbH_L)$ array recSamples set equal to output of picture reconstruction process $recSamples_L$ as inputs.

   – When ChromaArrayType is not equal to 0, the picture reconstruction process prior to in-loop filtering for a colour component as specified in clause 8.7.5 is invoked with the chroma coding block location (xCb / SubWidthC, yCb / SubHeightC), the variable nCurrW set equal to $nCbW_C$, the variable nCurrH set equal to $nCbH_C$, the variable cIdx set equal to 1, the $(nCbW_C)x(nCbH_C)$ array predSamples set equal to $predSamples_{Cb}$, and the $(nCbW_C)x(nCbH_C)$ array resSamples set equal to $resSamples_{Cb}$ as inputs.

   – When ChromaArrayType is not equal to 0, the picture reconstruction process prior to in-loop filtering for a colour component as specified in clause 8.7.5 is invoked with the chroma coding block location (xCb / SubWidthC, yCb / SubHeightC), the variable nCurrW set equal to $nCbW_C$, the variable nCurrH set equal to $nCbH_C$, the variable cIdx set equal to 2, the $(nCbW_C)x(nCbH_C)$ array predSamples set equal to $predSamples_{Cr}$, and the $(nCbW_C)x(nCbH_C)$ array resSamples set equal to $resSamples_{Cr}$ as inputs.

If affine_flag[xCb][yCb] is equal to 0, for use in derivation processes of variables invoked later in the decoding process, the following assignments are made for x = 0..nCbW − 1 and y = 0..nCbH − 1:

MvL0[xCb + x][yCb + y] = mvL0 (8-225)

MvL1[xCb + x][yCb + y] = mvL1 (8-226)

RefIdxL0[xCb + x][yCb + y] = refIdxL0 (8-227)

RefIdxL1[xCb + x][yCb + y] = refIdxL1 (8-228)

PredFlagL0[xCb + x][yCb + y] = predFlagL0 (8-229)

PredFlagL1[xCb + x][yCb + y] = predFlagL1 (8-230)

8.5.2 Derivation process for motion vector components and reference indices

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 nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are:

– the luma motion vectors mvL0 and mvL1,

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

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the DMVR utilization flag dmvrAppliedFlag.

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

The function LongTermRefPic(aPic, aPb, refIdx, LX), with X being 0 or 1, is defined as follows:

– If the picture with index refIdx from reference picture list LX of the slice containing coding block aPb in the picture aPic was marked as "used for long-term reference" at the time when aPic was the current picture, LongTermRefPic(aPic, aPb, refIdx, LX) is equal to 1.

– Otherwise, LongTermRefPic(aPic, aPb, refIdx, LX) is equal to 0.

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

– If sps_admvp_flag is equal to 0 and cu_skip_flag[xCb][yCb] is equal to 1, the derivation process for luma motion vectors for skip mode as specified in clause 8.5.2.1 is invoked with the luma location (xCb, yCb), the variables nCbW and nCbH as inputs, and the output being the luma motion vectors mvL0, mvL1, the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0,predFlagL1, the variable dmvrAppliedFlag is set equal to 0.

– Otherwise, if sps_admvp_flag is equal to 1 and cu_skip_flag[xCb][yCb] is equal to 1 or merge_mode_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 nCbW and nCbH as inputs, and the output being the luma motion vectors mvL0, mvL1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0 and predFlagL1. If affine_flag[xCb][yCb] is equal to 0, the DMVR utilization flag dmvrAppliedFlag is set 1, otherwise dmvrAppliedFlag is set 0.

– Otherwise, if sps_admvp_flag is equal to 0 or if direct_mode_flag[xCb][yCb] is equal to 1, the derivation process for luma motion vectors for direct mode as specified in clause 8.5.2.10 is invoked with the luma location (xCb, yCb), the variables nCbW and nCbH as inputs, and the output being the luma motion vectors mvL0, mvL1, the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0 and predFlagL1, the variable dmvrAppliedFlag is set equal to 1.

– Otherwise, for X being replaced by either 0 or 1 in the variables predFlagLX, mvLX and refIdxLX, in PRED_LX, and in the syntax elements ref_idx_lX and MvdLX, the variable dmvrAppliedFlag is set equal to 0 and the following ordered steps apply:

1. The variables refIdxLX and predFlagLX are derived as follows:
   - If sps_amis_flag is equal to 0, the following applies:
     - If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

       refIdxLX = ref_idx_lX[xCb][yCb] (8-231)

       predFlagLX = 1 (8-232)

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

       refIdxLX = −1 (8-233)

       predFlagLX = 0 (8-234)

   - If sps_amis_flag is equal to 1, the following applies:
     - If bi_pred_idx is equal to 0, the following applies:
       – If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

         refIdxLX = ref_idx_lX[xCb][yCb] (8-235)

         predFlagLX = 1 (8-236)

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

         refIdxLX = −1 (8-237)

         predFlagLX = 0 (8-238)

- Otherwise, the following applies:
  - If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

$$predFlagLX = 1 \quad (8\text{-}239)$$

    - The derivation process for luma reference index in clause 8.5.2.15 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH as inputs, and the output being refIdxLX.
  - Otherwise, the variables refIdxLX and predFlagLX are specified by:

$$refIdxLX = -1 \quad (8\text{-}240)$$

$$predFlagLX = 0 \quad (8\text{-}241)$$

2. The variable mvdLX is derived as follows:

$$mvdLX[\ 0\ ] = MvdLX[\ xCb\ ][\ yCb\ ][\ 0\ ] \quad (8\text{-}242)$$

$$mvdLX[\ 1\ ] = MvdLX[\ xCb\ ][\ yCb\ ][\ 1\ ] \quad (8\text{-}243)$$

3. When predFlagLX is equal to 1, the derivation process for luma motion vector prediction in clause 8.5.2.13 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH and refIdxLX as inputs, and the output being mvpLX.
4. When predFlagLX is equal to 1, the luma motion vector mvLX is derived as follows:

$$uLX[\ 0\ ] = (\ mvpLX[\ 0\ ] + mvdLX[\ 0\ ] + 2^{16}\ )\ \%\ 2^{16} \quad (8\text{-}244)$$

$$mvLX[\ 0\ ] = (\ uLX[\ 0\ ] >= 2^{15}\ )\ ?\ (\ uLX[\ 0\ ] - 2^{16}\ ) : uLX[\ 0\ ] \quad (8\text{-}245)$$

$$uLX[\ 1\ ] = (\ mvpLX[\ 1\ ] + mvdLX[\ 1\ ] + 2^{16}\ )\ \%\ 2^{16} \quad (8\text{-}246)$$

$$mvLX[\ 1\ ] = (\ uLX[\ 1\ ] >= 2^{15}\ )\ ?\ (\ uLX[\ 1\ ] - 2^{16}\ ) : uLX[\ 1\ ] \quad (8\text{-}247)$$

NOTE 1– The resulting values of mvLX[0] and mvLX[1] as specified above will always be in the range of -2^{15} to $2^{15} - 1$, inclusive.

When ChromaArrayType is not equal to 0 and predFlagLX, with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.14 is invoked with mvLX as input, and the output being mvCLX.

The updating process for the history-based motion vector predictor list as specified in clause 8.5.2.8. is invoked with luma motion vectors mvL0 and mvL1, reference indices refIdxL0 and refIdxL1 and HmvpCandList as input.

8.5.2.1 Derivation process for luma motion vectors for skip mode

This process is only invoked when cu_skip_flag[xCb][yCb] is equal to 1 and sps_amis_flag is equal to 0, and sps_admvp_flag is equal to 0 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,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are:

– the luma motion vectors mvL0 and mvL1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

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

1. When slice_type is equal to P, the variables mvL1, refIdxL1 and predFlagL1 are set as follows:

$$mvL1[\ 0\ ] = 0 \tag{8-248}$$

$$mvL1[\ 1\ ] = 0 \tag{8-249}$$

$$refIdxL1 = -1 \tag{8-250}$$

$$predFlagL0 = 0 \tag{8-251}$$

2. The derivation process for motion vector predictor from neighbouring coding unit partitions in clause 8.5.2.9 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the motion vector prediction index mvpIdx equal to mvp_idx_l0[xCb][yCb] and the reference list identifier listX set equal to 0 as inputs, and the output being the motion vector predictor mvPred. The variables mvL0, refIdxL0 and predFlagL0 are set as follows:

$$mvL0 = mvPred \tag{8-252}$$

$$refIdxL0 = 0 \tag{8-253}$$

$$predFlagL0 = 1 \tag{8-254}$$

3. When slice_type is equal to B, the derivation process for motion vector predictor from neighbouring coding unit partitions in clause 8.5.2.9 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the motion vector prediction index mvpIdx equal to mvp_idx_l1[xCb][yCb] and the reference list identifier listX set equal to 1 as inputs, and the output being the motion vector predictor mvPred. The variables mvL1, refIdxL1 and predFlagL1 are set as follows:

$$mvL1 = mvPred \tag{8-255}$$

$$refIdxL1 = 0 \tag{8-256}$$

$$predFlagL1 = 1 \tag{8-257}$$

8.5.2.2 Derivation process for luma motion vectors for merge mode

This process is only invoked when cu_skip_flag[xCb][yCb] is equal to 1 or merge_mode_flag[xCb][yCb] is equal to 1 when and when sps_admvp_flag 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,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are:

– the luma motion vectors mvL0 and mvL1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

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

1. The derivation process for spatial merge candidates in clause 8.5.2.3 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW and the coding block height nCbH as inputs, and the output being the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$ and $availableFlagB_2$, the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$ and $refIdxLXB_2$, the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$ and $predFlagLXB_2$, and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$ and $mvLXB_2$, with X being 0 or 1.

2. The reference indices for the temporal merging candidate, refIdxLXCol, with X being 0 or 1, are set equal to 0.

3. The derivation process for temporal merge candidates in clause 8.5.2.4 is invoked with the luma location (xCb, yCb), the luma coding block width nCbW and the luma coding block height nCbH as inputs, and the output being the availability flags availableFlagLXCol, , the reference indices refIdxLXCol, the prediction list utilization flags predFlagLXCol, predFlagLXColand the motion vectors mvLXCol, with X being equal to 0 or 1.

4. When slice_type is not equal to B, the variables availableFlagCol, predFlagL0Col and predFlagL1Col are derived as follows:

$$availableFlagCol = availableFlagL0Col \tag{8-258}$$

$$predFlagL0Col = availableFlagL0Col \tag{8-259}$$

$$predFlagL1Col = 0 \tag{8-260}$$

5. When slice_type is equal to B and, input variables nCbW and nCbH are both larger than 4, the variables availableFlagCol, predFlagL0Col and predFlagL1Col are derived as follows:

$$availableFlagCol = availableFlagL0Col \ || \ availableFlagL1Col \tag{8-261}$$

$$predFlagL0Col = availableFlagL0Col \tag{8-262}$$

$$predFlagL1Col = availableFlagL1Col \tag{8-263}$$

6. If one or more of the following conditions are true, availableFlagCol is set equal to 0, both components of mvLXCol are set equal to 0, refIdxLXCol is set equal to −1 and predFlagLXCol is set equal to 0, with X being 0 or 1:

   – availableFlagCol is equal to FALSE.

   – availableA1 is equal to TRUE and coding units covering the luma locations A1 and Col have the same motion vectors and the same reference indices.

   – availableB1 is equal to TRUE and the coding units covering the luma locations B1 and Col have the same motion vectors and the same reference indices.

   – availableB0 is equal to TRUE and the coding units covering the luma locations B0 and Col have the same motion vectors and the same reference indices.

   – availableA0 is equal to TRUE and the coding units covering the luma locations A0 and Col have the same motion vectors and the same reference indices.

   – availableB2 is equal to TRUE and the coding units covering the luma locations B2 and Col have the same motion vectors and the same reference indices.

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

```
Initialize variable mLSize = ( nCbW * nCbH <=  32 ) ? 4 : 6.
numCurrMergeCand = 0
if( availableFlagA1 )
        mergeCandList[ numCurrMergeCand++ ] = A1
if( availableFlagB1 )
        mergeCandList[ numCurrMergeCand++ ] = B1
if( availableFlagB0 )
        mergeCandList[ numCurrMergeCand++ ] = B0                    (8-264)
if( availableFlagA0  && numCurrMergeCand < mLSize – 1 )
        mergeCandList[ numCurrMergeCand++ ] = A0
if( availableFlagB2 && numCurrMergeCand < mLSize – 1 )
        mergeCandList[ numCurrMergeCand++ ] = B2
if( availableFlagCol && numCurrMergeCand < mLSize )
        mergeCandList[ numCurrMergeCand++ ] = Col
```

8. When numCurrMergeCand is less than mLSize and NumHmvpCand (number of entries in HmvpCandList) is greater than 2, the derivation process of history-based merging candidates as specified in clause 8.5.2.6 is invoked

with mergeCandList, numCurrMergeCand and maximal size of merge candidate list mLSize as inputs, and modified mergeCandList and numCurrMergeCand as outputs.

9. When slice_type is equal to B, input variables nCbW and nCbH are both larger than 4, and numCurrMergeCand is less than mLSize the derivation process for combined bi-predictive merging candidates specified in clause 8.5.2.11 is invoked with mergeCandList, numCurrMergeCand and maximal size of merge candidate list mLSize as inputs, and modified mergeCandList and numCurrMergeCand as output.
10. When numCurrMergeCand is less then mLSize, the derivation process for zero motion vector merging candidates specified in clause 8.5.2.12 is invoked with the mergeCandList, numCurrMergeCand and maximal size of merge candidate list mLSize as inputs, and modified mergeCandList and numCurrMergeCand as output.
11. The following assignments are made with N being the candidate at position mvp_idx[xOrigP][yOrigP] in the merging candidate list mergeCandList (N = mergeCandList[mvp_idx[xOrigP][yOrigP]]) and X being replaced by 0 or 1:

$$refIdxLX = refIdxLXN \tag{8-265}$$

$$mvLX[\ 0\ ] = mvLXN[\ 0\ ] \tag{8-266}$$

$$mvLX[\ 1\ ] = mvLXN[\ 1\ ] \tag{8-267}$$

refIdxLX if a valid refernece picture:

$$predFlagLX = 1 \tag{8-268}$$

12. When nCbW and nCbH are both less than 8 , for any entry of the mergeCandList[i] the following applies:

$$refIdxL1 = -1 \tag{8-269}$$

$$predFlagL1 = 0 \tag{8-270}$$

13. When mmvd_flag[xCb][yCb] is equal to 1, the following applies:
    - The derivation process for MMVD motion vector as specified in 8.5.2.18 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 MMVD motion vectors mmvdMvL0 and mmvdMvL1, the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0] as outputs.
    - The luma motion vector mvLX is replaced by the MMVD motion vector mmvdMvLX for X being 0 and 1 as follows:

$$mvLX[\ 0\ ][\ 0\ ][\ 0\ ] = mmvdMvLX[\ 0\ ] \tag{8-271}$$

$$mvLX[\ 0\ ][\ 0\ ][\ 1\ ] = mmvdMvLX[\ 1\ ] \tag{8-272}$$

8.5.2.3 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,
- two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are as follows, with X being 0 or 1:

- the availability flags availableFlagA_0, availableFlagA_1, availableFlagB_0, availableFlagB_1 and availableFlagB_2 of the neighbouring coding units,
- the reference indices refIdxLXA_0, refIdxLXA_1, refIdxLXB_0, refIdxLXB_1 and refIdxLXB_2 of the neighbouring coding units,

– the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$ and $predFlagLXB_2$ of the neighbouring coding units,

– the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$ and $mvLXB_2$ of the neighbouring coding units.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

When availLR is equal to LR_11, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW , yCb).

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb − 1).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1 , yCb − 1).

Otherwise, when availLR is equal to LR_01, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb + nCbH − 1).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb + nCbH).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).

Otherwise, when availLR is equal to LR_10 or LR_00, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW − 1, yCb − 1).

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

For the derivation of $availableFlagA_1$, $refIdxLXA_1$, $predFlagLXA_1$ and $mvLXA_1$ the following applies:

– The availability derivation process for a coding block as specified below in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the coding block availability flag $availableA_1$.

– The variables $availableFlagA_1$, $refIdxLXA_1$, $predFlagLXA_1$ and $mvLXA_1$ are derived as follows:

  – If $availableA_1$ is equal to FALSE, $availableFlagA_1$ is set equal to 0, both components of $mvLXA_1$ are set equal to 0, $refIdxLXA_1$ is set equal to −1 and $predFlagLXA_1$ is set equal to 0, with X being 0 or 1.

  – Otherwise, $availableFlagA_1$ is set equal to 1 and the following assignments are made:

$$mvLXA_1 = MvLX[\ xNbA_1\ ][\ yNbA_1\ ] \quad (8\text{-}273)$$

$$refIdxLXA_1 = RefIdxLX[\ xNbA_1\ ][\ yNbA_1\ ] \quad (8\text{-}274)$$

$$predFlagLXA_1 = PredFlagLX[\ xNbA_1\ ][\ yNbA_1\ ] \quad (8\text{-}275)$$

For the derivation of $availableFlagB_1$, $refIdxLXB_1$, $predFlagLXB_1$ and $mvLXB_1$ the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current coding block height nCbH and the luma location ($xNbB_1$, $yNbB_1$) as inputs, and the output is assigned to the coding block availability flag $availableB_1$.

– The variables $availableFlagB_1$, $refIdxLXB_1$, $predFlagLXB_1$ and $mvLXB_1$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagB_1$ is set equal to 0, both components of $mvLXB_1$ are set equal to 0, $refIdxLXB_1$ is set equal to −1 and $predFlagLXB_1$ is set equal to 0, with X being 0 or 1:

    – $availableB_1$ is equal to FALSE.

    – $availableA_1$ is equal to TRUE and the coding units covering the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbB_1$, $yNbB_1$) have the same motion vectors and the same reference indices.

  – Otherwise, $availableFlagB_1$ is set equal to 1 and the following assignments are made:

$$mvLXB_1 = MvLX[\ xNbB_1\ ][\ yNbB_1\ ] \quad (8\text{-}276)$$

$$refIdxLXB_1 = RefIdxLX[\ xNbB_1\ ][\ yNbB_1\ ] \quad (8\text{-}277)$$

$$predFlagLXB_1 = PredFlagLX[\ xNbB_1\ ][\ yNbB_1\ ] \quad (8\text{-}278)$$

For the derivation of $availableFlagB_0$, $refIdxLXB_0$, $predFlagLXB_0$ and $mvLXB_0$ the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbB_0$, $yNbB_0$) as inputs, and the output is assigned to the coding block availability flag $availableB_0$.

– The variables $availableFlagB_0$, $refIdxLXB_0$, $predFlagLXB_0$ and $mvLXB_0$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagB_0$ is set equal to 0, both components of $mvLXB_0$ are set equal to 0, $refIdxLXB_0$ is set equal to −1 and $predFlagLXB_0$ is set equal to 0, with X being 0 or 1:

    – $availableB_0$ is equal to FALSE.

    – $availableA_1$ is equal to TRUE and the coding units covering the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbB_0$, $yNbB_0$) have the same motion vectors and the same reference indices.

    – $availableB_1$ is equal to TRUE and the coding units covering the luma locations ($xNbB_1$, $yNbB_1$) and ($xNbB_0$, $yNbB_0$) have the same motion vectors and the same reference indices.

  – Otherwise, $availableFlagB_0$ is set equal to 1 and the following assignments are made:

$$mvLXB_0 = MvLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (8\text{-}279)$$

$$refIdxLXB_0 = RefIdxLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (8\text{-}280)$$

$$predFlagLXB_0 = PredFlagLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (8\text{-}281)$$

For the derivation of $availableFlagA_0$, $refIdxLXA_0$, $predFlagLXA_0$ and $mvLXA_0$ the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_0$, $yNbA_0$) as inputs, and the output is assigned to the coding block availability flag $availableA_0$.

– The variables $availableFlagA_0$, $refIdxLXA_0$, $predFlagLXA_0$ and $mvLXA_0$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagA_0$ is set equal to 0, both components of $mvLXA_0$ are set equal to 0, $refIdxLXA_0$ is set equal to −1 and $predFlagLXA_0$ is set equal to 0, with X being 0 or 1:

    – $availableA_0$ is equal to FALSE.

    – $availableA_1$ is equal to TRUE and the coding units covering the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbB_0$, $yNbB_0$) have the same motion vectors and the same reference indices.

- $availableB_1$ is equal to TRUE and the coding units covering the luma locations ($xNbB_1$, $yNbB_1$) and ($xNbB_0$, $yNbB_0$) have the same motion vectors and the same reference indices.
- $availableB_0$ is equal to TRUE and the coding units covering the luma locations ($xNbB_0$, $yNbB_0$) and ($xNbB_0$, $yNbB_0$) have the same motion vectors and the same reference indices.

– Otherwise, $availableFlagA_0$ is set equal to 1 and the following assignments are made:

$$mvLXA_0 = MvLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}282)$$

$$refIdxLXA_0 = RefIdxLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}283)$$

$$predFlagLXA_0 = PredFlagLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}284)$$

For the derivation of $availableFlagB_2$, $refIdxLXB_2$, $predFlagLXB_2$ and $mvLXB_2$ the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block nCbH and the luma location ($xNbB_2$, $yNbB_2$) and the partition index partIdx as inputs, and the output is assigned to the coding block availability flag $availableB_2$.

– The variables $availableFlagB_2$, $refIdxLXB_2$, $predFlagLXB_2$ and $mvLXB_2$ are derived as follows:

  - If one or more of the following conditions are true, $availableFlagB_2$ is set equal to 0, both components of $mvLXB_2$ are set equal to 0, $refIdxLXB_2$ is set equal to −1 and $predFlagLXB_2$ is set equal to 0, with X being 0 or 1:
    - $availableB_2$ is equal to FALSE.
    - $availableA_1$ is equal to TRUE and coding units covering the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbB_2$, $yNbB_2$) have the same motion vectors and the same reference indices.
    - $availableB_1$ is equal to TRUE and the coding units covering the luma locations ($xNbB_1$, $yNbB_1$) and ($xNbB_2$, $yNbB_2$) have the same motion vectors and the same reference indices.
    - $availableB_0$ is equal to TRUE and the coding units covering the luma locations ($xNbB_0$, $yNbB_0$) and ($xNbB_2$, $yNbB_2$) have the same motion vectors and the same reference indices.
    - $availableA_0$ is equal to TRUE and the coding units covering the luma locations ($xNbA_0$, $yNbA_0$) and ($xNbB_2$, $yNbB_2$) have the same motion vectors and the same reference indices.
  - Otherwise, $availableFlagB_2$ is set equal to 1 and the following assignments are made:

$$mvLXB_2 = MvLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}285)$$

$$refIdxLXB_2 = RefIdxLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}286)$$

$$predFlagLXB_2 = PredFlagLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}287)$$

8.5.2.4 Derivation process for temporal merge 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 nCbW and nCbH specifying the width and the height of the luma coding block,

– a reference index refIdxLX, with X being 0 or 1.

Outputs of this process are:

– the availability flags availableFlagLXCol,

– the motion vectors mvLXCol.

The variable currCb specifies the current luma coding block at luma location (xCb, yCb).

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

The variables mvLXCol and availableFlagLXCol are derived as follows:

1. The central collocated motion vector is derived as follows:

$$xColCtr = xCb + (cbWidth >> 1) \quad (8\text{-}288)$$

$$yColCtr = yCb + (cbHeight >> 1) \quad (8\text{-}289)$$

   – The variable colCb specifies the luma coding block covering the modified location given by ((xColCtr >> MIN_CU_LOG2) << MIN_CU_LOG2, (yColCtr >> MIN_CU_LOG2) << MIN_CU_LOG2) 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 8.5.2.5 is invoked with currCb, colCb, (xColCb, yColCb) and refIdxLX as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

2. When availableFlagLXCol is equal to 0, the bottom collocated motion vector is derived as follows:

   When availLR is equal to LR_01, the following applies:

$$xColBot = xCb \quad (8\text{-}290)$$

$$yColBot = yCb + cbHeight \quad (8\text{-}291)$$

   Otherwise, the following applies:

$$xColBot = xCb + cbWidth - 1 \quad (8\text{-}292)$$

$$yColBot = yCb + cbHeight \quad (8\text{-}293)$$

   – If yCb >> CtbLog2SizeY is equal to yColBot >> CtbLog2SizeY, yColBot is less than pic_height_in_luma_samples, the following applies:

      – The variable colCb specifies the luma coding block covering the modified location given by ((xColBot >> MIN_CU_LOG2) << MIN_CU_LOG2, (yColBot >> MIN_CU_LOG2) << MIN_CU_LOG2) 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 8.5.2.5 is invoked with currCb, colCb, (xColCb, yColCb) and refIdxLX 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.

3. When availableFlagLXCol is equal to 0, the bottom right collocated motion vector is derived as follows:

   When availLR is equal to LR_01, the following applies:

$$xColBr = xCb - 1 \quad (8\text{-}294)$$

$$yColBr = yCb + cbHeight - 1 \quad (8\text{-}295)$$

   Otherwise, the following applies:

$$xColBr = xCb + cbWidth \quad (8\text{-}296)$$

$$yColBr = yCb + cbHeight - 1 \quad (8\text{-}297)$$

   – If xCb >> CtbLog2SizeX is equal to yColBr >> CtbLog2SizeX, 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 >> MIN_CU_LOG2) << MIN_CU_LOG2, (yColBr >> MIN_CU_LOG2) << MIN _CU_LOG2) 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 8.5.2.5 is invoked with currCb, colCb, (xColCb, yColCb) and refIdxLX 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.

8.5.2.5 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.

Outputs of this process are:

– the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy, with X being equal 0 or 1

– the availability flag availableFlagCol (jointly for lists 0 and 1).

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], refMvL0[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], refMvL1[x][y] and RefIdxL1[x][y], respectively, of the collocated picture specified by ColPic.

The variables mvLXCol and availableFlagLXCol, for X being equal to 0 or 1, are derived as follows:

If colCb is coded in an intra prediction mode, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

Otherwise, the motion vector mvCol, the reference index refIdxCol and the reference list identifier listCol are derived as follows:

– If temporal_mvp_asigned_flag is equal to 0, the following is applied:
  - If predFlagL0Col[x][y] is equal to 0, and predFlagL1Col[x][y] is equal to 1, mvCol, refIdxCol and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] and LX with X being equal to 1, respectively.
  - If predFlagL0Col[x][y] is equal to 1, and predFlagL1Col[x][y] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] and LX with X being equal to 0, respectively.
  - Otherwise (predFlagL0Col[x][y] is equal to 1, and predFlagL1Col[x][y] is equal to 1), mvCol, refIdxCol and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] and LX with X being 0 or 1 specifying reference list 0 and reference list 1, respectively.
– If temporal_mvp_asigned_flag is equal to 1, the following is applied:
  - If predFlagLXCol[x][y] with X equal to collocated_mvp_source_list_idx is equal to 1, mvCol, refIdxCol and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] and LX with X being equal to col_source_mvp_list_idx, respectively.

and mvLXCol and availableFlagLXCol are derived as follows:

– the variable refPicListCol[refIdxCol] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the slice containing prediction block colPb in the collocated picture specified by ColPic, and the following applies:

– The POC distance (denoted as currPocDiffX) between current picture and current picutre's reference picture list RefPicListX[refIdxLX] with refIdxLX being equal to 0, is computed as following:

currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) (8-298)

– The POC distance (denoted as colPocDiffLX) between the collocated picture ColPic and the list X reference picture of the collocated picture refPicListCol[refIdxLXCol] is computed as following:

colPocDiffLX = DiffPicOrderCnt(ColPic, refPicListCol[refIdxLXCol]) (8-299)

– For valid refIdxCol and colPocDiffLX is not equal to 0, the variable availableFlagLXCol is set equal to 1, the mvLXCol is derived as a scaled version of the motion vector mvColLX as follows:

distScaleFactorLX = currPocDiffLX << 5 / colPocDiffLX (8-300)

mvLXCol = Clip3(−32768, 32767, Sign(distScaleFactorLX * mvCol) *
((Abs(distScaleFactorLX * mvCol) + 16) >> 5)) (8-301)

– Otherwise (invalid refIdxCol or colPocDiffLX is equal to 0) the variable availableFlagLXCol is set equal to 0 and

mvLXCol = 0 (8-302)

with X being 0 or 1 specifying reference list 0 and reference list 1, respectively.

The picture boundary based clipping process for variables for collocated motion vectors mvLXCol is invoked as specified in clause 8.5.2.21 with mvLXCol, (xCb, yCb), pic_width_in_luma_samples and pic_height_in_luma_samples as input parameters.

The availableFlagCol is derived as follows:

– If availableFlagLXCol for X being 0 and 1 are both valid, availableFlagCol is set to 3,
– Otherwise if availableFlagLXCol for X being equal to 0 is valid and availableFlagLXCol for X being 1 is invalid, availableFlagCol is set to 1,
– Otherwise if availableFlagLXCol for X being equal to 1 is valid and availableFlagLXCol for X being 0 is invalid, availableFlagCol is set to 2,
– Otherwise, availableFlagCol is set to 0.

8.5.2.6 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,

– history based motion information table HmvpCandList,

– the maximal number of elements mLSize within mergeCandList.

Outputs to this process are:

– the modified merging candidate list mergeCandList,

– the modified number of merging candidates in the list numCurrMergeCand.

The variable numOrigMergeCand is set equal to numCurrMergeCand. The variable hmvpStop is set to FALSE. The variable NumHmvpCand is set to the number of motion entries in HmvpCandList.

The variable maxNumCheckedHistory is set to be equal to (((NumHmvpCand+1) >> 2) << 2) – 1.

For each candidate in HmvpCandList with hMvpIdx = 3, 7, …, min(maxNumCheckedHistory, (mLSize = 4) ? 15 : 23), the following ordered steps are repeated until hmvpStop is equal to TRUE

1. The variable sameMotion is derived as follows:

- sameMotion is set to FALSE

- if HMVPCandList[NumHmvpCand− hMvpIdx] have the same motion vectors and the same reference indices with any mergeCandList[i] with i being 0… numOrigMergeCand – 1, sameMotion is set to TRUE

2. When sameMotion is equal to false, the candidate HmvpCandList[NumHmvpCand− hMvpIdx] is added to the merging candidate list as follows:

mergeCandList[numCurrMergeCand++] = HmvpCandList[NumHmvpCand− hMvpIdx] (8-303)

3. if numCurrMergeCand is equal to mLSize, hmvpStop is set to TRUE.

8.5.2.7 Derivation process for history-based motion vector prediction candidates

Inputs to this process are:

– history based motion information table HmvpCandList,

– the current reference picture index curRefIdxLX, with X being 0 or 1.

Outputs to this process are:

– the DefaultMvLX, with X being 0 or 1,

– the DefaultRefIdxLX, with X being 0 or 1.

Variable HmvpMVList is defined as the history based motion vector table part derived from HmvpCandList. HmvpRefList is defined as the reference indices table part derived from HmvpCandList. NumHmvpCand is set to the number of motion entries in HmvpCandList.

For a given reference list X, with X being 0 or 1, for each index hMvpIdx with hMvpIdx = 1, …, min(4, NumHmvpCand) in HmvpMVList[NumHmvpCand - hMvpIdx] and HmvpRefList[NumHmvpCand - hMvpIdx], the following steps are executed until variable HMVPDerived is set to TRUE:

1. hMvpIdx is set to 1, HMVPDerived is set to FALSE
2. if curRefIdxLX = HmvpRefList[NumHmvpCand − hMvpIdx][X], with X being 0 or 1, DefaultRefIdxLX is set to be HmvpRefList[NumHmvpCand − hMvpIdx][X] and DefaultMvLX is set to NumHmvpCand [HMVPCandNum – hMvpIdx][X] and HMVPDerived is set to TRUE, otherwise hMvpIdx ++
3. When HMVPDerived is equal to FALSE and hMvpIdx < min(4, NumHmvpCand), repeat step 2.

If HMVPDerived is equal to FALSE, variable DefaultRefIdxLX and DefaultMvLX are derived as follows:

1. hMvpIdx is set to 1
2. if curRefIdxLX = HmvpRefList[NumHmvpCand − hMvpIdx][X] is valid, DefaultRefIdxLX is set to be HmvpRefList[NumHmvpCand − hMvpIdx][X] and DefaultMvLX is set to HmvpMVList[NumHmvpCand − hMvpIdx][X] and HMVPDerived is set to TRUE, otherwise hMvpIdx++.
3. When HMVPDerived is equal to FALSE and hMvpIdx < min(4, NumHmvpCand), repeat step 2.

If HMVPDerived is equal to FALSE, DefaultMvLX[x], and DefaultMvLX[y] and DefaultRefIdxLX are set equal to 0.

8.5.2.8 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,

– history based motion information table HmvpCandList.

Output to this process is the modified history based motion information table HmvpCandList.

The MVP candidate hMvpCand consists of the luma motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1. NumHmvpCand is set to the number of motion entries in HmvpCandList.

If slie_type is equal to P and refIdxL0 is valid or if slice_type is equal to B and either refIdxL0 or refIdxL1 is valid, the candidate list HmvpCandList is modified using the candidate mvCand by the following ordered steps:

1. The variable curIdx is set equal to NumHmvpCand.
2. If NumHmvpCand is equal to 23, for each index hMvpIdx = 1 … NumHmvpCand − 1, copy HMVPCandList[hMvpIdx] to HMVPCandList[hMvpIdx − 1].
3. Copy hMvpCand to HMVPCandList[hMvpIdx].

4. If NumHmvpCand is smaller than 23, NumHmvpCand is increased by 1.

8.5.2.9 Derivation process for motion vector predictor from neighbouring coding unit partitions

This process is only invoked when sps_admvp_flag is equal to 0.

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 nCbW and nCbH specifying the width and the height of the current luma coding block,

– a motion vector prediction index mvpIdx,

– a reference list identifier listX.

Output of this process is the motion vector predictor mvPred.

The motion vector predictors list, mvpList, is derived with X being replaced by listX as follows:

For the derivation of mvpList[0] the following applies:

– The luma location (xNb0, yNb0) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

– The availability derivation process for a coding block as specified in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH, the luma location (xNb0, yNb0) and the partition index partIdx as inputs, and the output is assigned to the coding block availability flag availableNb0.

– If availableNb0 is equal to TRUE, mvpList[0] is derived as follows:

$$mvpList[0] = MvLX[xNb0][yNb0] \quad (8\text{-}304)$$

– Otherwise, mvpList[0] is derived as follows:

$$mvpList[0][0] = 1 \quad (8\text{-}305)$$

$$mvpList[0][1] = 1 \quad (8\text{-}306)$$

For the derivation of mvpList[1] the following applies:

– The luma location (xNb1, yNb1) inside the neighbouring luma coding block is set equal to (xCb, yCb− 1).

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH, the luma location (xNb1, yNb1) and the partition index partIdx as inputs, and the output is assigned to the coding block availability flag availableNb1.

– If availableNb1 is equal to TRUE, mvpList[1] is derived as follows:

  – if yNb1 >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than xNb1 >> MaxCbLog2Size11Ratio

$$mvpList[1] = refMvLX[xNb1][yNb1] \quad (8\text{-}307)$$

  – Otherwise, the following applies:

$$mvpList[1] = MvLX[xNb1][yNb1] \quad (8\text{-}308)$$

– Otherwise, mvpList[1] is derived as follows:

$$mvpList[1][0] = 1 \quad (8\text{-}309)$$

$$mvpList[1][1] = 1 \quad (8\text{-}310)$$

For the derivation of mvpList[2] the following applies:

– The luma location (xNb2, yNb2) inside the neighbouring luma coding block is set equal to (xCb +xCbW− 1, yCb− 1).

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH, the luma location (xNb2, yNb2) and the partition index partIdx as inputs, and the output is assigned to the coding block availability flag availableNb2.

– If availableNb2 is equal to TRUE, mvpList[2] is derived as follows:

  – If yNb2 >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than xNb2 >> MaxCbLog2Size11Ratio

    mvpList[2] = refMvLX[xNb2][yNb2] (8-311)

  – Otherwise, the following applies:

    mvpList[2] = MvLX[xNb2][yNb2] (8-312)

– Otherwise, mvpList[2] is derived as follows:

    mvpList[2][0] = 1 (8-313)

    mvpList[2][1] = 1 (8-314)

For the derivation of mvpList[3] the following applies:

– The variable TempPic is set equal to RefPicListX[0].

– The luma location (xNb3, yNb3) inside the neighbouring luma coding block is set equal to (xCb , yCb).

– The variable mvTemp specifies the refined motion vector of list 0 (refMvL0) covering the luma location (xNb3, yNb3) inside the temporal picture specified by TempPic.

– mvpList[3] is derived as follows:

    mvpList[3] = mvTemp (8-315)

The variable mvPred is set to mvpList[mvpIdx].

8.5.2.10 Derivation process for luma motion vectors for direct mode

This process is only invoked when direct_mode_flag[xCb][yCb] is equal to 1, sps_amis_flag is equal to 0 and sps_admvp_flag is equal to 0, 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,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are:

– the luma motion vectors mvL0 and mvL1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

The variable currPic specifies the current picture.

The following steps apply:

– The variables mvL0, mvL1, refIdxL0, refIdxL1, predFlagL0 and predFlagL1 are set as follows:

    mvL0[0] = 0 (8-316)

    mvL0[1] = 0 (8-317)

mvL1[0] = 0 (8-318)

mvL1[1] = 0 (8-319)

refIdxL0 = 0 (8-320)

refIdxL1 = 0 (8-321)

predFlagL0 = 1 (8-322)

predFlagL1 = 0 (8-323)

– The variable TempPic is set equal to RefPicList1[0]
– The motion vector mvTemp is set equal to the refined motion vector of list 0 (refMvL0) covering the luma location (xCb, yCb) inside the temporal picture specified by TempPic.
– The variable refPicListTemp[0] is set to be the picture with reference index 0 in the reference picture list of the coding block covering the luma location (xCb, yCb) in the temporal picture specified by TempPic.
– The variables diffPocNorm and diffPocDeNormL0 are derived as follows:

diffPocDeNorm = DiffPicOrderCnt(TempPic, refPicListTemp[0]) (8-324)

diffPocNormL0 = DiffPicOrderCnt(currPic, RefPicList0[0]) (8-325)

diffPocNormL1 = DiffPicOrderCnt(RefPicList1[0], currPic) (8-326)

– If diffPocDeNorm is equal to 0, the following applies:

mvL0[0] = 0 (8-327)

mvL0[1] = 0 (8-328)

mvL1[0] = 0 (8-329)

mvL1[1] = 0 (8-330)

– Otherwise, the following applies:

mvL0[0] = diffPocNormL0 * mvTemp[0] / diffPocDeNorm (8-331)

mvL0[1] = diffPocNormL0 * mvTemp[1] / diffPocDeNorm (8-332)

mvL1[0] = –diffPocNormL1 * mvTemp[0] / diffPocDeNorm (8-333)

mvL1[1] = –diffPocNormL1 * mvTemp[1] / diffPocDeNorm (8-334)

8.5.2.11 Derivation process for combined bi-predictive merging candidates

Inputs to this process are:

– a merging candidate list mergeCandList,
– the number of elements numCurrMergeCand within mergeCandList,
– the maximal number of elements mLSize within mergeCandList.

Outputs of this process are:

– the merging candidate list mergeCandList,
– the number of elements numCurrMergeCand within mergeCandList.

The arrays refIdxL0 and refIdxL1, each size of numCurrMergeCand, are composed by the reference indices of the list 0 and reference indices of the list 1, respectively, of every candidate in the mergeCandList. The arrays of motion vectors mvL0 and mvL1, each size of numCurrMergeCand, are composed by motion vectors in list 0 and list 1, respectively, of every candidate in mergeCandList.

When numCurrMergeCand is greater than 1 and less than mLSize, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable combIdx is set equal to 0, the variable combStop is set equal to FALSE and the following ordered steps are repeated until combStop is equal to TRUE:

1. The variables l0CandIdx and l1CandIdx are derived using combIdx as specified in Table 8-8.
2. The following assignments are made:

$$refIdxL0l0Cand = refIdxL0[\ l0CandIdx\ ] \quad (8\text{-}335)$$

$$refIdxL1l1Cand = refIdxL1[\ l1CandIdx\ ] \quad (8\text{-}336)$$

$$mvL0l0Cand = mvL0[\ l0CandIdx\ ] \quad (8\text{-}337)$$

$$mvL1l1Cand = mvL1[\ l1CandIdx\ ] \quad (8\text{-}338)$$

3. When both refIdxL0l0Cand and refIdxL1l1Cand are valid, the candidate $combCand_k$ with k equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $combCand_k$, the motion vectors of $combCand_k$ and numCurrMergeCand are derived as follows:

$$refIdxL0combCand_k = refIdxL0l0Cand \quad (8\text{-}339)$$

$$refIdxL1combCand_k = refIdxL1l1Cand \quad (8\text{-}340)$$

$$mvL0combCand_k[\ 0\ ] = mvL0l0Cand[\ 0\ ] \quad (8\text{-}341)$$

$$mvL0combCand_k[\ 1\ ] = mvL0l0Cand[\ 1\ ] \quad (8\text{-}342)$$

$$mvL1combCand_k[\ 0\ ] = mvL1l1Cand[\ 0\ ] \quad (8\text{-}343)$$

$$mvL1combCand_k[\ 1\ ] = mvL1l1Cand[\ 1\ ] \quad (8\text{-}344)$$

$$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}345)$$

4. The variable combIdx is incremented by 1.
5. When combIdx is equal to (numInputMergeCand* (numInputMergeCand− 1)) or numCurrMergeCand is equal to mLSize, combStop is set equal to TRUE.

Table 8-8 – Specification of l0CandIdx and l1CandIdx

combIdx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
l0CandIdx	0	1	0	2	1	2	0	3	1	3	2	3
l1CandIdx	1	0	2	0	2	1	3	0	3	1	3	2
combIdx	12	13	14	15	16	17	18	19				
l0CandIdx	0	4	1	4	2	4	3	4				
l1CandIdx	4	0	4	1	4	2	4	3				

8.5.2.12 Derivation process for zero motion vector merging candidates

Inputs to this process are:

– a merging candidate list mergeCandList,

– the number of elements numCurrMergeCand within mergeCandList,

– the maximal number of elements mLSize within mergeCandList.

Outputs of this process are:

– the merging candidate list mergeCandList,

– the number of elements numCurrMergeCand within mergeCandList.

The variable zeroIdx is set equal to 1.

Until numCurrMergeCand is less than mLSize, the candidate $zeroCand_m$ with m equal to zeroIdx is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $zeroCand_m$ as follows:

– If slice_type is equal to P, the reference indices and the motion vectors of $zeroCand_m$ are derived as follows:

$$refIdxL0zeroCand_m = 0 \quad (8\text{-}346)$$

$$refIdxL1zeroCand_m = -1 \quad (8\text{-}347)$$

$$mvL0zeroCand_m[\ 0\ ] = 0 \quad (8\text{-}348)$$

$$mvL0zeroCand_m[\ 1\ ] = 0 \quad (8\text{-}349)$$

$$mvL1zeroCand_m[\ 0\ ] = 0 \quad (8\text{-}350)$$

$$mvL1zeroCand_m[\ 1\ ] = 0 \quad (8\text{-}351)$$

$$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}352)$$

$$zeroIdx = zeroIdx + 1 \quad (8\text{-}353)$$

– Otherwise (slice_type is equal to B), the reference indices and the motion vectors of $zeroCand_m$ are derived as follows and numCurrMergeCand is incremented by 1:

$$refIdxL0zeroCand_m = 0 \quad (8\text{-}354)$$

$$refIdxL1zeroCand_m = 0 \quad (8\text{-}355)$$

$$mvL0zeroCand_m[\ 0\ ] = 0 \quad (8\text{-}356)$$

$$mvL0zeroCand_m[\ 1\ ] = 0 \quad (8\text{-}357)$$

$$mvL1zeroCand_m[\ 0\ ] = 0 \quad (8\text{-}358)$$

$$mvL1zeroCand_m[\ 1\ ] = 0 \quad (8\text{-}359)$$

$$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}360)$$

$$zeroIdx = zeroIdx + 1 \quad (8\text{-}361)$$

8.5.2.13 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,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the prediction mvpLX of the motion vector mvLX, with X being 0 or 1.

When sps_amis_flag is equal to 0, the following applies:

The motion vector predictor mvpL0 and mvpL1 are derived in the following ordered steps:

1. The variables mvpL0 and mvpL1 are set as follows:

$$mvpL0[\ 0\ ] = 0 \tag{8-362}$$

$$mvpL0[\ 1\ ] = 0 \tag{8-363}$$

$$mvpL1[\ 0\ ] = 0 \tag{8-364}$$

$$mvpL1[\ 1\ ] = 0 \tag{8-365}$$

2. If refIdxL0 is not equal to –1, the derivation process for motion vector predictor from neighbouring coding unit partitions in clause 8.5.2.9 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the motion vector prediction index mvpIdx equal to mvp_idx_l0[xCb][yCb] and the reference list identifier listX set equal to 0 as inputs, and the output being the motion vector predictor mvPred. The variable mvpL0 is set equal to mvPred.
3. If refIdxL1 is not equal to –1, the derivation process for motion vector predictor from neighbouring coding unit partitions in clause 8.5.2.9 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the motion vector prediction index mvpIdx equal to mvp_idx_l1[xCb][yCb] and the reference list identifier listX set equal to 1 as inputs, and the output being the motion vector predictor mvPred. The variable mvL1 is set equal to mvPred

Otherwise, when sps_amis_flag is equal to 1, the following applies:

A variable mvpAvailFlag is set to equal to 0.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

When availLR is equal to LR_11, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).
– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW , yCb).
– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).
– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb − 1).
– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1 , yCb − 1).

Otherwise, when availLR is equal to LR_01, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb + nCbH − 1).
– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).
– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).
– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb + nCbH).
– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).

Otherwise, when availLR is equal to LR_10 or LR_00, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).
– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW − 1, yCb − 1).
– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

If amvr_idx[x0][y0] is equal to 0, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the coding block availability flag $availableA_1$.

– If $availableA_1$ is equal to TRUE, the following applies:

  – mvpAvailFlag is set to equal to 1.

  – if $yNbA_1$ >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than $xNbA_1$ >> MaxCbLog2Size11Ratio

    mvpLX is set equal to refMvLX[$xNbA_1$][$yNbA_1$].

  – otherwise

    mvpLX is set equal to MvLX[$xNbA_1$][$yNbA_1$].

  – If refIdxLX is not equal to RefIdxLX[$xNbA_1$][$yNbA_1$] and , DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbA_1$][$yNbA_1$]], RefPicListX[refIdxLX]) is not equal to 0, mvpLX is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}366)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}367)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactor * mvpLX) * ((Abs(distScaleFactor * mvpLX) + 127) >> 8)) \quad (8\text{-}368)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[xNbA_1][yNbA_1]])) \quad (8\text{-}369)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}370)$$

Otherwise, if amvr_idx[x0][y0] is equal to 1, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current coding block height nCbH and the luma location ($xNbB_1$, $yNbB_1$) as inputs, and the output is assigned to the coding block availability flag $availableB_1$.

– If $availableB_1$ is equal to TRUE, the following applies:.

  – mvpAvailFlag is set to equal to 1.

  – If $yNbB_1$ >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than $xNbB_1$ >> MaxCbLog2Size11Ratio

    mvpLX is set equal to refMvLX[$xNbB_1$][$yNbB_1$].

  – Otherwise, the following applies:

    mvpLX is set equal to MvLX[$xNbB_1$][$yNbB_1$].

  – If refIdxLX is not equal to RefIdxLX[$xNbB_1$][$yNbB_1$] and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbB_1$][$yNbB_1$]], RefPicListX[refIdxLX]) is not equal to 0, mvpLX is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}371)$$

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-372)

mvpLX = Clip3(−32768, 32767, Sign(distScaleFactor * mvpLX) *
((Abs(distScaleFactor * mvpLX) + 127) >> 8)) (8-373)

where td and tb are derived as follows:

td = Clip3(−128, 127, DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[$xNbB_1$][$yNbB_1$]])) (8-374)

tb = Clip3(−128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) (8-375)

Otherwise, if amvr_idx[x0][y0] is equal to 2, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbB_0$, $yNbB_0$) as inputs, and the output is assigned to the coding block availability flag $availableB_0$.

– If $availableB_0$ is equal to TRUE, the following applies:.

  – mvpAvailFlag is set to equal to 1.

  – If $yNbB_0$ >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than $xNbB_0$ >> MaxCbLog2Size11Ratio

    mvpLX is set equal to refMvLX[$xNbB_0$][$yNbB_0$].

  – Otherwise, the following applies:

    mvpLX is set equal to MvLX[$xNbB_0$][$yNbB_0$].

  – If refIdxLX is not equal to RefIdxLX[$xNbB_0$][$yNbB_0$] and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbB_0$][$yNbB_0$]], RefPicListX[refIdxLX]) is not equal to 0, mvpLX is derived as follows:

    tx = (16384 + (Abs(td) >> 1)) / td (8-376)

    distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-377)

    mvpLX = Clip3(−32768, 32767, Sign(distScaleFactor * mvpLX) *
    ((Abs(distScaleFactor * mvpLX) + 127) >> 8)) (8-378)

    where td and tb are derived as follows:

    td = Clip3(−128, 127, DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[$xNbB_0$][$yNbB_0$]])) (8-379)

    tb = Clip3(−128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) (8-380)

Otherwise, if amvr_idx[x0][y0] is equal to 3, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_0$, $yNbA_0$) as inputs, and the output is assigned to the coding block availability flag $availableA_0$.

– If $availableA_0$ is equal to TRUE, the following applies:

  – mvpAvailFlag is set to equal to 1.

  – If $yNbA_0$ >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than $xNbA_0$ >> MaxCbLog2Size11Ratio

    mvpLX is set equal to refMvLX[$xNbA_0$][$yNbA_0$].

  – Otherwise, the following applies:

    mvpLX is set equal to MvLX[$xNbA_0$][$yNbA_0$].

– If refIdxLX is equal to RefIdxLX[$xNbA_0$][$yNbA_0$] and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbA_0$][$yNbA_0$]], RefPicListX[refIdxLX]) is not equal to 0, mvpLX is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}381)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}382)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactor * mvpLX) * ((Abs(distScaleFactor * mvpLX) + 127) >> 8)) \quad (8\text{-}383)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[xNbA_0][yNbA_0]])) \quad (8\text{-}384)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}385)$$

Otherwise, if amvr_idx[x0][y0] is equal to 4, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block nCbH and the luma location ($xNbB_2$, $yNbB_2$) and the partition index partIdx as inputs, and the output is assigned to the coding block availability flag $availableB_2$.

– If $availableB_2$ is equal to TRUE, the following applies:

  – mvpAvailFlag is set to equal to 1.

  – If $yNbB_2$ >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than $xNbB_2$ >> MaxCbLog2Size11Ratio

    mvpLX is set equal to refMvLX[$xNbB_2$][$yNbB_2$].

  – Otherwise, the following applies:

    mvpLX is set equal to MvLX[$xNbB_2$][$yNbB_2$].

  – If refIdxLX is equal to RefIdxLX[$xNbB_2$][$yNbB_2$] and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbB_2$][$yNbB_2$]], RefPicListX[refIdxLX]) is not equal to 0, mvpLX is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}386)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}387)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactor * mvpLX) * ((Abs(distScaleFactor * mvpLX) + 127) >> 8)) \quad (8\text{-}388)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[xNbB_2][yNbB_2]])) \quad (8\text{-}389)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}390)$$

When mvpAvailFlag is equal to 0, the following applies:

– The derivation process for default reference index as specified in clause 8.5.2.19 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the reference index of the current coding unit partition refIdxLX as inputs, and the output is assigned to the default reference index DefaultRefIdxLX, with X being 0 or 1.

– The derivation process for default motion vector as specified in clause 8.5.2.20 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the reference index of the current coding unit partition refIdxLX as inputs, and the output is assigned to the default motion vector predictor DefaultMvLX, with X being 0 or 1..

– If refIdxLX is not equal to DefaultRefIdxLX and DiffPicOrderCnt(RefPicListX[DefaultRefIdxLX], RefPicListX[refIdxLX]) is not equal to 0, mvpLX is dervied as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}391)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}392)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactor * DefaultMvLX) * ((Abs(distScaleFactor * DefaultMvLX) + 127) >> 8)) \quad (8\text{-}393)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[DefaultRefIdxLX])) \quad (8\text{-}394)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}395)$$

When sps_amvr_flag is equal to 1 and amvr_idx[x0][y0] is not equal to 0, the following applies:

$$mvpLX[0] = mvpLX[0] >= 0 ? ((mvpLX[0] + (1 << (amvr\_idx[x0][y0] - 1))) >> amvr\_idx[x0][y0]) << amvr\_idx[x0][y0] : -(((-mvpLX[0] + (1 << (amvr\_idx[x0][y0] - 1))) >> amvr\_idx[x0][y0]) << amvr\_idx[x0][y0]) \quad (8\text{-}396)$$

$$mvpLX[1] = mvpLX[1] >= 0 ? ((mvpLX[1] + (1 << (amvr\_idx[x0][y0] - 1))) >> amvr\_idx[x0][y0]) << amvr\_idx[x0][y0] : -(((-mvpLX[1] + (1 << (amvr\_idx[x0][y0] - 1))) >> amvr\_idx[x0][y0]) << amvr\_idx[x0][y0]) \quad (8\text{-}397)$$

8.5.2.14 Derivation process for chroma motion vector

Input to this process is a luma motion vector mvLX.

Output of this process is a chroma motion vector mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvCLX, the following applies:

$$mvCLX[0] = mvLX[0] \quad (8\text{-}398)$$

$$mvCLX[1] = mvLX[1] \quad (8\text{-}399)$$

8.5.2.15 Derivation process for the luma reference index

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 nCbW and nCbH specifying the width and the height of the current luma coding block.

Output of this process is the reference index refIdxLX.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

When availLR is equal to LR_11, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW , yCb).

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb − 1).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1 , yCb − 1).

Otherwise, when availLR is equal to LR_01, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb + nCbH − 1).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb + nCbH).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).

Otherwise, when availLR is equal to LR_10 or LR_00, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW − 1, yCb − 1).

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH).

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

The derivation process for default reference index as specified in clause 8.5.2.19 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the reference index of the current coding unit partition set to 0 as inputs, and the output is assigned to the default reference index DefaultRefIdxLX, with X being 0 or 1.

If amvr_idx[x0][y0] is equal to 0, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the coding block availability flag $availableA_1$.

– If $availableA_1$ is equal to FALSE, refIdxLX is set equal to DefaultRefIdxLX.

– Otherwise, the following applies:

  – If RefIdxLX[$xNbA_1$][$yNbA_1$] is not equal to –1, refIdxLX is set equal to RefIdxLX[$xNbA_1$][$yNbA_1$].

  – Otherwise, refIdxLX is set equal to DefaultRefIdxLX.

Otherwise, if amvr_idx[x0][y0] is equal to 1, the following applies:

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW − 1, yCb − 1).

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current coding block height nCbH and the luma location ($xNbB_1$, $yNbB_1$) as inputs, and the output is assigned to the coding block availability flag $availableB_1$.

– If $availableB_1$ is equal to FALSE, refIdxLX is set equal to DefaultRefIdxLX.

– Otherwise, the following applies:

  – If RefIdxLX[$xNbB_1$][$yNbB_1$] is not equal to –1, refIdxLX is set equal to RefIdxLX[$xNbB_1$][$yNbB_1$].

– Otherwise, refIdxLX is set equal to DefaultRefIdxLX.

Otherwise, if amvr_idx[x0][y0] is equal to 2, the following applies:

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb − 1).
– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbB_0$, $yNbB_0$) as inputs, and the output is assigned to the coding block availability flag $availableB_0$.
– If $availableB_0$ is equal to FALSE, refIdxLX is set equal to DefaultRefIdxLX.
– Otherwise, the following applies:
  – If $RefIdxLX[xNbB_0][yNbB_0]$ is not equal to –1, refIdxLX is set equal to $RefIdxLX[xNbB_0][yNbB_0]$.
  – Otherwise, refIdxLX is set equal to DefaultRefIdxLX.

Otherwise, if amvr_idx[x0][y0] is equal to 3, the following applies:

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH).
– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_0$, $yNbA_0$) as inputs, and the output is assigned to the coding block availability flag $availableA_0$.
– If $availableA_0$ is equal to FALSE, refIdxLX is set equal to DefaultRefIdxLX.
– Otherwise, the following applies:
  – If $RefIdxLX[xNbA_0][yNbA_0]$ is not equal to –1, refIdxLX is set equal to $RefIdxLX[xNbA_0][yNbA_0]$.
  – Otherwise, refIdxLX is set equal to DefaultRefIdxLX.

Otherwise, if amvr_idx[x0][y0] is equal to 4, the following applies:

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).
– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block nCbH and the luma location ($xNbB_2$, $yNbB_2$) and the partition index partIdx as inputs, and the output is assigned to the coding block availability flag $availableB_2$.
– If $availableB_2$ is equal to FALSE, refIdxLX is set equal to DefaultRefIdxLX.
– Otherwise, the following applies:
  – If $RefIdxLX[xNbB_2][yNbB_2]$ is not equal to –1, refIdxLX is set equal to $RefIdxLX[xNbB_2][yNbB_2]$.
  – Otherwise, refIdxLX is set equal to DefaultRefIdxLX.

8.5.2.16 Conversion luma motion vectors from 1/16 fractional-sample accuracy to ¼ fractional-sample accuracy

Inputs to this process are:

– the motion vectors mvIL0 and mvIL1 in 1/16 pel accuracy,

Outputs of the process are:

– the motion vectors mvL0 and mvL1 in ¼ pel accuracy.

For the conversion of the luma motion vector mvILX, the following applies:

$$mvL0[0] = mvIL0[0] >> 2 \quad (8\text{-}400)$$

$$mvL0[1] = mvIL0[1] >> 2 \quad (8\text{-}401)$$

8.5.2.17 Conversion luma motion vectors from ¼ fractional-sample accuracy to 1/16 fractional-sample accuracy

Inputs to this process are:

– the motion vectors mvIL0 and mvIL1 in 1/16 pel accuracy.

Outputs of the process are:

– the motion vectors mvL0 and mvL1 in ¼ pel accuracy.

For the conversion of the luma motion vector mvILX, the following applies:

$$mvL0[0] = mvIL0[0] << 2 \quad (8\text{-}402)$$

$$mvL0[1] = mvIL0[1] << 2 \quad (8\text{-}403)$$

8.5.2.18 Derivation process for MMVD motion vector

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,

– luma merge motion vectors mvL0 and mvL1,

– reference indices refIdxL0 and refIdxL1,

– prediction list utilization flags predFlagL0 and predFlagL1,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are the luma merge motion vector mMvL0, mMvL1, refIdxL0, refIdxL1, predFlagL0 and predFlagL1.

The variable currPic specifies the current picture.

The MMVD motion vectors mMvL0 and mMvL1 are derived, and refIdxL0, refIdxL1, predFlagL0 and predFlagL1 are updated as follows:

– The MMVD motion vectors are set as follows:

$$mMvL0 = mvL0 \quad (8\text{-}404)$$

$$mMvL1 = mvL1 \quad (8\text{-}405)$$

– If the value of mmvd_group_idx is equal to 1, the following applies:

  – If both predFlagL0 and predFlagL1 are equal to 1, the following applies:

$$refIdxL1 = -1 \quad (8\text{-}406)$$

$$predFlagL1 = 0 \quad (8\text{-}407)$$

$$mMvL1[\ 0\ ] = 0 \quad (8\text{-}408)$$

$$mMvL1[\ 1\ ] = 0 \quad (8\text{-}409)$$

  – Otherwise, if predFlagL0 is equal to 1, the following applies:

$$predFlagL1 = 1 \quad (8\text{-}410)$$

    – If PicOrderCnt(RefPicList1[1]) is equal to (2 * PicOrderCnt(currPic) – PicOrderCnt(RefPicList0[refIdxL0])), the following applies:

$$refIdxL1 = 1 \quad (8\text{-}411)$$

    – Otherwise, the following applies:

$$refIdxL1= 0 \quad (8\text{-}412)$$

    – mMvL1 is set as follows:

$$currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) \quad (8\text{-}413)$$

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (8-414)

td = Clip3(−128, 127, currPocDiffL1) (8-415)

tb = Clip3(−128, 127, currPocDiffL0) (8-416)

tx = (16384 + (Abs(td) >> 1)) / td (8-417)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-418)

mMvL1[0] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL0[0]) * ((Abs(distScaleFactor * mMvL0[0]) + 127) >> 8)) (8-419)

mMvL1[1] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL0[1]) * ((Abs(distScaleFactor * mMvL0[1]) + 127) >> 8)) (8-420)

– Otherwise, if predFlagL1 are equal to 1, the following applies:

predFlagL0 = 1 (8-421)

– If PicOrderCnt(RefPicList0[1]) is equal to (2 * PicOrderCnt(currPic) – PicOrderCnt(RefPicList1[refIdxL1])), the following applies:

refIdxL0 = 1 (8-422)

– Otherwise, the following applies:

refIdxL0 = 0 (8-423)

– mMvL0 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (8-424)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (8-425)

td = Clip3(−128, 127, currPocDiffL0) (8-426)

tb = Clip3(−128, 127, currPocDiffL1) (8-427)

tx = (16384 + (Abs(td) >> 1)) / td (8-428)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-429)

mMvL0[0] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL1[0]) * ((Abs(distScaleFactor * mMvL1[0]) + 127) >> 8)) (8-430)

mMvL0[1] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL1[1]) * ((Abs(distScaleFactor * mMvL1[1]) + 127) >> 8)) (8-431)

– Otherwise, if the value of mmvd_group_idx is equal to 2, the following applies:

– If both predFlagL0 and predFlagL1 are equal to 1, the following applies:

refIdxL0 = −1 (8-432)

predFlagL0 = 0 (8-433)

mMvL0[0] = 0 (8-434)

mMvL0[1] = 0 (8-435)

– Otherwise, if predFlagL0 is equal to 1, the following applies:

predFlagL1 = 1 (8-436)

– If PicOrderCnt(RefPicList1[1]) is equal to (2 * PicOrderCnt(currPic) – PicOrderCnt(RefPicList0[refIdxL0])), the following applies:

refIdxL1 = 1 (8-437)

– Otherwise, the following applies:

refIdxL1 = 0 (8-438)

– mMvL1 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (8-439)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (8-440)

td = Clip3(−128, 127, currPocDiffL1) (8-441)

tb = Clip3(−128, 127, currPocDiffL0) (8-442)

tx = (16384 + (Abs(td) >> 1)) / td (8-443)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-444)

mMvL1[0] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL0[0]) * ((Abs(distScaleFactor * mMvL0[0]) + 127) >> 8)) (8-445)

mMvL1[1] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL0[1]) * ((Abs(distScaleFactor * mMvL0[1]) + 127) >> 8)) (8-446)

– refIdxL0 is set equal to –1, and predFlagL0, mMvL0[0] and mMvL0[1] are set equal to 0.

– Otherwise, if predFlagL1 is equal to 1, the following applies:

predFlagL0 = 1 (8-447)

– If PicOrderCnt(RefPicList0[1]) is equal to (2 * PicOrderCnt(currPic) – PicOrderCnt(RefPicList1[refIdxL1])), the following applies:

refIdxL0 = 1 (8-448)

– Otherwise, the following applies:

refIdxL0 = 0 (8-449)

– mMvL0 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (8-450)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (8-451)

td = Clip3(−128, 127, currPocDiffL0) (8-452)

tb = Clip3(−128, 127, currPocDiffL1) (8-453)

tx = (16384 + (Abs(td) >> 1)) / td (8-454)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-455)

mMvL0[0] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL1[0]) * ((Abs(distScaleFactor * mMvL1[0]) + 127) >> 8)) (8-456)

mMvL0[1] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL1[1]) * ((Abs(distScaleFactor * mMvL1[1]) + 127) >> 8)) (8-457)

refIdxL1 is set equal to –1, and predFlagL1, mMvL1[0] and mMvL1[1] are set equal to 0.

– Otherwise, if the value of mmvd_group_idx is equal to 0 and nCbW*nCbH <=32 is equal to 1, the following applies:
  – If both predFlagL0 and predFlagL1 are equal to 1, the following applies:

refIdxL1 = −1 (8-458)

predFlagL1 = 0 (8-459)

mMvL1[0] = 0 (8-460)

mMvL1[1] = 0 (8-461)

  – Otherwise, if predFlagL1 is equal to 1, the following applies:

predFlagL0 = 1 (8-462)

    – If PicOrderCnt(RefPicList0[1]) is equal to (2 * PicOrderCnt(currPic) – PicOrderCnt(RefPicList1[refIdxL1])), the following applies:

refIdxL0 = 1 (8-463)

    – Otherwise, the following applies:

refIdxL0 = 0 (8-464)

    – mMvL0 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (8-465)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (8-466)

td = Clip3(−128, 127, currPocDiffL0) (8-467)

tb = Clip3(−128, 127, currPocDiffL1) (8-468)

tx = (16384 + (Abs(td) >> 1)) / td (8-469)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-470)

mMvL0[0] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL1[0]) * ((Abs(distScaleFactor * mMvL1[0]) + 127) >> 8)) (8-471)

mMvL0[1] = Clip3(-2^{15}, $2^{15} - 1$, Sign(distScaleFactor * mMvL1[1]) * ((Abs(distScaleFactor * mMvL1[1]) + 127) >> 8)) (8-472)

    – refIdxL1 is set equal to –1, and predFlagL1, mMvL1[0] and mMvL1[1] are set equal to 0.
    – refIdxL1 = −1 (8-473)
    – predFlagL1 = 0 (8-474)
    – mMvL1[0] = 0 (8-475)
    – mMvL1[1] = 0 (8-476)

– If both predFlagL0 and predFlagL1 are equal to 1, the following applies:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (8-477)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (8-478)

mMvdL0[0] = MmvdOffset[0] (8-479)

mMvdL0[1] = MmvdOffset[1] (8-480)

mMvdL1[0] = MmvdOffset[0] (8-481)

mMvdL1[1] = MmvdOffset[1] (8-482)

– If Abs(currPocDiffL0) is greater than Abs(currPocDiffL1), the following applies:

td = Clip3(−128, 127, Abs(currPocDiffL0)) (8-483)

tb = Clip3(−128, 127, Abs(currPocDiffL1)) (8-484)

tx = (16384 + (Abs(td) >> 1)) / td (8-485)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-486)

mMvdL0[0] = Clip3(-2^{15}, $2^{15}-1$, Sign(distScaleFactor * mMvdL0[0]) * ((Abs(distScaleFactor * mMvdL0[0]) + 127) >> 8)) (8-487)

mMvdL0[1] = Clip3(-2^{15}, $2^{15}-1$, Sign(distScaleFactor * mMvdL0[1]) * ((Abs(distScaleFactor * mMvdL0[1]) + 127) >> 8)) (8-488)

– Otherwise if Abs(currPocDiffL0) is less than Abs(currPocDiffL1), the following applies:

td = Clip3(−128, 127, Abs(currPocDiffL1)) (8-489)

tb = Clip3(−128, 127, Abs(currPocDiffL0)) (8-490)

tx = (16384 + (Abs(td) >> 1)) / td (8-491)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-492)

mMvdL1[0] = Clip3(-2^{15}, $2^{15}-1$, Sign(distScaleFactor * mMvdL1[0]) * ((Abs(distScaleFactor * mMvdL1[0]) + 127) >> 8)) (8-493)

mMvdL1[1] = Clip3(-2^{15}, $2^{15}-1$, Sign(distScaleFactor * mMvdL1[1]) * ((Abs(distScaleFactor * mMvdL1[1]) + 127) >> 8)) (8-494)

– If currPocDiffL0 * currPocDiffL1 is smaller than 0, the following applies:

mMvdL0[0] = mMvdL0[0] (8-495)

mMvdL0[1] = mMvdL0[1] (8-496)

mMvdL1[0] = −mMvdL1[0] (8-497)

mMvdL1[1] = −mMvdL1[1] (8-498)

– Otherwise (predFlagL0 or predFlagL1 are equal to 1), the following applies for X being 0 and 1:

mMvdLX[0] = (predFlagLX = = 1) ? MmvdOffset[0] : 0 (8-499)

mMvdLX[1] = (predFlagLX = = 1) ? MmvdOffset[1] : 0 (8-500)

– The MMVD motion vectors are updated as follows:

mMvL0[0] += mMvdL0[0] (8-501)

mMvL0[1] += mMvdL0[1] (8-502)

mMvL1[0] += mMvdL1[0] (8-503)

mMvL1[1] += mMvdL1[1] (8-504)

8.5.2.19 Derivation process for default reference index

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 nCbW and nCbH specifying the width and the height of the current luma coding block,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the DefaultRefIdxLX, with X being 0 or 1.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

When availLR is equal to LR_11, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW , yCb).

Otherwise, when availLR is equal to LR_01, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb + nCbH − 1).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

Otherwise, when availLR is equal to LR_10 or LR_00, the following applies:

– The luma location ($xNbA_1$, $yNbA_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb + nCbW − 1, yCb − 1).

For the derivation of DefaultRefIdxLX, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the coding block availability flag $availableA_1$.

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current coding block height nCbH and the luma location ($xNbB_1$, $yNbB_1$) as inputs, and the output is assigned to the coding block availability flag $availableB_1$.

– For the derivation of DefaultRefIdxL0, the following applies:

  – DefaultRefIdxL0 is set equal to 0.

  – If $availableA_1$ is equal to TRUE and refIdxL0 is equal to $RefIdxL0[xNbA_1][yNbA_1]$, DefaultRefIdxL0 is set to refIdxL0.

  – Otherwise if $availableB_1$ is equal to TRUE and refIdxL0 is equal to $RefIdxL0[xNbB_1][yNbB_1]$, DefaultRefIdxL0 is set to refIdxL0.

  – Otherwise if $availableA_1$ is equal to TRUE and $RefIdxL0[xNbA_1][yNbA_1]$ is not equal to −1, DefaultRefIdxL0 is set to $RefIdxL0[xNbA_1][yNbA_1]$.

  – Otherwise if $availableB_1$ is equal to TRUE and $RefIdxL0[xNbB_1][yNbB_1]$ is not equal to −1, DefaultRefIdxL0 is set to $RefIdxL0[xNbB_1][yNbB_1]$.

  – Otherwise, when sps_admvp is set equal to 1, the DefaultRefIdxL1 is set equal to the refIdxL1 of the history motion vector candidates derived as specified in clause 8.5.2.7.

– For the derivation of DefaultRefIdxL1, the following applies

  – DefaultRefIdxL1 is set equal to 0.

  – If availableA_1 is equal to TRUE and refIdxL1 is equal to RefIdxL1[xNbA_1][yNbA_1], DefaultRefIdxL1 is set to refIdxL1.

  – Otherwise if availableB_1 is equal to TRUE and refIdxL1 is equal to RefIdxL1[xNbB_1][yNbB_1], DefaultRefIdxL1 is set to refIdxL1.

  – Otherwise if availableA_1 is equal to TRUE and RefIdxL1[xNbA_1][yNbA_1] is not equal to −1, DefaultRefIdxL1 is set to RefIdxL1[xNbA_1][yNbA_1].

  – Otherwise if availableB_1 is equal to TRUE and RefIdxL1[xNbB_1][yNbB_1] is not equal to −1, DefaultRefIdxL1 is set to RefIdxL1[xNbB_1][yNbB_1].

  – Otherwise, when sps_admvp is set equal to 1, the DefaultRefIdxL1 is set equal to the refIdxL1 of the history motion vector candidates derived as specified in clause 8.5.2.7.

8.5.2.20 Derivation process for default 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,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the DefaultMvLX, with X being 0 or 1.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

When availLR is equal to LR_11, the following applies:

– The luma location (xNbA_1, yNbA_1) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

– The luma location (xNbB_1, yNbB_1) inside the neighbouring luma coding block is set equal to (xCb+ nCbW , yCb).

Otherwise, when availLR is equal to LR_01, the following applies:

– The luma location (xNbA_1, yNbA_1) inside the neighbouring luma coding block is set equal to (xCb + nCbW , yCb + nCbH − 1).

– The luma location (xNbB_1, yNbB_1) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

Otherwise, when availLR is equal to LR_10 or LR_00, the following applies:

– The luma location (xNbA_1, yNbA_1) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).

– The luma location (xNbB_1, yNbB_1) inside the neighbouring luma coding block is set equal to (xCb + nCbW − 1, yCb − 1).

For the derivation of DefaultMvLX, the following applies:

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbA_1, yNbA_1) as inputs, and the output is assigned to the coding block availability flag availableA_1.

– The availability derivation process for a coding block as specified in clause 6.4.3 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current coding block height nCbH and the luma location (xNbB_1, yNbB_1) as inputs, and the output is assigned to the coding block availability flag availableB_1.

– For each X=0,1, the motion vectors refinedMvLX MvLX[xNbA_1][yNbA_1] and refinedMvLX[xNbB_1][yNbB_1] are set equal to MvLX[xNbA_1][yNbA_1] and MvLX[xNbB_1][yNbB_1], respectively.

– For each N equal to A_1, and B_1

– if yNbN >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than xNbN >> MaxCbLog2Size11Ratio

refinedMvLX[N][N] is set equal to refMvLX[N][N].

– For the derivation of DefaultMvL0, the following applies:
  – Both components of DefaultMvL0 is set equal to 0.
  – If availableA_1 is equal to TRUE and refIdxL0 is equal to RefIdxL0[xNbA_1][yNbA_1], DefaultMvL0 is set equal to MvL0[xNbA_1][yNbA_1].
  – Otherwise if availableB_1 is equal to TRUE and refIdxL0 is equal to RefIdxL0[xNbB_1][yNbB_1], DefaultMvL0 is set equal to refinedMvL0[xNbB_1][yNbB_1].
  – Otherwise if availableA_1 is equal to TRUE and RefIdxL0[xNbA_1][yNbA_1] is not equal to −1, DefaultMvL0 is set equal to refinedMvL0[xNbA_1][yNbA_1].
  – Otherwise if availableB_1 is equal to TRUE and RefIdxL0[xNbB_1][yNbB_1] is not equal to −1, DefaultMvL0 is set equal to refinedMvL0[xNbB_1][yNbB_1].
  – Otherwise, when sps_admvp is set equal to 1, the DefaultMvL0 is set equal to the history motion vector candidates derived as specified in clause 8.5.2.7.
– For the derivation of DefaultMvL1, the following applies:
  – Both components of DefaultMvL1 is set equal to 0.
  – If availableA_1 is equal to TRUE and refIdxL1 is equal to RefIdxL1[xNbA_1][yNbA_1], DefaultMvL1 is set equal to refinedMvL1[xNbA_1][yNbA_1].
  – Otherwise if availableB_1 is equal to TRUE and refIdxL1 is equal to RefIdxL1[xNbB_1][yNbB_1], DefaultMvL1 is set equal to refinedMvL1[xNbB_1][yNbB_1].
  – Otherwise if availableA_1 is equal to TRUE and RefIdxL1[xNbA_1][yNbA_1] is not equal to −1, DefaultMvL1 is set equal to refinedMvL1[xNbA_1][yNbA_1].
  – Otherwise if availableB_1 is equal to TRUE and RefIdxL1[xNbB_1][yNbB_1] is not equal to −1, DefaultMvL1 is set equal to refinedMvL1[xNbB_1][yNbB_1].
  – Otherwise, when sps_admvp is set equal to 1, the DefaultMvL1 is set equal to the history motion vector candidates derived as specified in clause 8.5.2.7.

8.5.2.21 Derivation process for constrained scaled motion

Inputs to this process are:

– the motion vector prediction mvLXCol,
– a luma location (xCb, yCb) specifying the top-left sample of the luma coding block,
– pic_width_in_luma_samples and pic_height_in_luma_samples specifying the width and length of coded pictures.

Outputs of this process is the motion vector prediction mvLXCol.

The variable mvLXCol for x being set 0 or 1 derived as follows:

– The variables paddedWidth and paddedHeights are derived as follows:

paddedWidth = pic_width_in_luma_samples + picPaddingSize,

paddedHeight = pic_height_in_luma_samples + picPaddingSize.

mvLXCol[MV_X] = (xCb + mvLXCol[MV_X]) < picPaddingSize ? -(xCb + picPaddingSize) : mvLXCol [MV_X]
mvLXCol[MV_Y] = (yCb + mvLXCol [MV_Y]) < picPaddingSize ? -(yCb + picPaddingSize) : mvLXCol [MV_Y]
mvLXCol[MV_X] = (xCb + mvLXCol [MV_X]) > paddedWidth ? (paddedWidth - xCb) : mvLXCol [MV_X] (8-505)
mvLXCol[MV_Y] = (yCb + mvLXCol [MV_Y]) > paddedHeight ? (paddedHeight - yCb) : mvLXCol [MV_Y];

8.5.3 Derivation process for affine motion vector components and reference indices

8.5.3.1 General

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the luma subblock motion vector array in 1/16 fractional-sample accuracy mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1,

– the chroma subblock motion vector array in 1/32 fractional-sample accuracy mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0 .. numSbX − 1, ySbIdx = 0 .. numSbY − 1, X being 0 and 1,

– the number of control point motion vectors numCpMv,

– the control point motion vectors cpMvLX[cpIdx] with cpIdx = 0..numCpMv – 1, X being 0 and 1.

For the derivation of the variables mvL0[xSbIdx][ySbIdx], mvL1[xSbIdx][ySbIdx], mvCL0[xSbIdx][ySbIdx] and mvCL1[xSbIdx][ySbIdx], refIdxL0, refIdxL1, numSbXL0, numSbXL1, numSbYL0, numSbYL1, predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx], the following applies:

– For the derivation of the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, refIdxL0, refIdxL1, predFlagL0 and predFlagL1, the following applies:

  – If merge_flag[xCb][yCb] is equal to 1, the derivation process for affine control point motion vectors and reference indices in affine merge mode as specified in 8.5.3.2is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight as input, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx], cpMvL1[cpIdx], with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as output.

  – Otherwise (merge_flag[xCb][yCb] is equal to 0 and affine_flag is equal to 1), for X being replaced by either 0 or 1 in the variables predFlagLX, cpMvLX, MvdCpLX, and refIdxLX, in PRED_LX, and in the syntax element ref_idx_lX, the following ordered steps apply:

    1. The number of control point motion vectors numCpMv is set equal to MotionModelIdc[xCb][yCb] + 1.

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

       − If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

         refIdxLX = ref_idx_lX[xCb][yCb] (8-506)

         predFlagLX[0][0] = 1 (8-507)

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

         refIdxLX = −1 (8-508)

         predFlagLX[0][0] = 0 (8-509)

    3. The variable mvdCpLX[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, is derived as follows:

       mvdCpLX[cpIdx][0] = MvdCpLX[xCb][yCb][cpIdx][0] (8-510)

       mvdCpLX[cpIdx][1] = MvdCpLX[xCb][yCb][cpIdx][1] (8-511)

4. When predFlagLX[0][0] is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in clause 8.5.3.5 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.

5. 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:

$$uLX[cpIdx][0] = (mvpCpLX[cpIdx][0] + mvdCpLX[cpIdx][0] + 2^{16}) \% 2^{16} \quad (8\text{-}512)$$

$$cpMvLX[cpIdx][0] = (uLX[cpIdx][0] >= 2^{15}) ? (uLX[cpIdx][0] - 2^{16}) : uLX[cpIdx][0] \quad (8\text{-}513)$$

$$uLX[cpIdx][1] = (mvpCpLX[cpIdx][1] + mvdCpLX[cpIdx][1] + 2^{16}) \% 2^{16} \quad (8\text{-}514)$$

$$cpMvLX[cpIdx][1] = (uLX[cpIdx][1] >= 2^{15}) ? (uLX[cpIdx][1] - 2^{16}) : uLX[cpIdx][1] \quad (8\text{-}515)$$

– The derivation process for subblock motion vector arrays from affine control point motion vectors as specified in subclause 8.5.3.7 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL1[cpIdx], cpMvL0[cpIdx] with cpIdx being 0..2, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, the number of luma coding subblocks in horizontal direction numSbXLX and in vertical direction numSbYLX, the size of luma coding subblocks in horizontal direction sizeSbXLX and in vertical direction sizeSbYLX, the luma motion vector array mvLX[xSbIdx][ySbIdx] and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0.. numSbXLX − 1, ySbIdx = 0 .. numSbYLX − 1, X being 0 and 1 as output.

– For xSbIdx = 0.. numSbXLX − 1 and ySbIdx = 0.. numSbYLX − 1, the motion vectors MvLX with X being 0 and 1 are derived as follows:

  – The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

$$(xSb, ySb) = (xCb + xSbIdx * sizeSbXLX, yCb + ySbIdx * sizeSbYLX) \quad (8\text{-}516)$$

  – If xSbIdx and ySbIdx are both equal to 0 (top-left subblock), the following applies for x = 0.. sizeSbXLX − 1 and y = 0..sizeSbYLX − 1:

$$MvL0[xSb + x][ySb + y] = cpMvL0[0] \quad (8\text{-}517)$$

$$MvL1[xSb + x][ySb + y] = cpMvL1[0] \quad (8\text{-}518)$$

  – Otherwise, if xSbIdx is equal to numSbXLX − 1 and ySbIdx is equal to 0 (top-right subblock), the following applies for x = 0.. sizeSbXLX – 1 and y = 0.. sizeSbYLX − 1:

$$MvL0[xSb + x][ySb + y] = cpMvL0[1] \quad (8\text{-}519)$$

$$MvL1[xSb + x][ySb + y] = cpMvL1[1] \quad (8\text{-}520)$$

  – Otherwise, if numCpMv is equal to 3 and xSbIdx is equal to 0 and ySbIdx is equal to numSbYLX − 1 (belowleft subblock), the following applies for x = 0..sizeSbXLX − 1 and y = 0..sizeSbYLX − 1:

$$MvL0[xSb + x][ySb + y] = cpMvL0[2] \quad (8\text{-}521)$$

$$MvL1[xSb + x][ySb + y] = cpMvL1[2] \quad (8\text{-}522)$$

  – Otherwise, the following applies for x = 0..sizeSbXLX − 1 and y = 0..sizeSbYLX − 1:

$$MvL0[xSb + x][ySb + y] = mvL0[xSbIdx][ySbIdx] >> 2 \quad (8\text{-}523)$$

$$MvL1[xSb + x][ySb + y] = mvL1[xSbIdx][ySbIdx] >> 2 \quad (8\text{-}524)$$

– For x = 0..cbWidth − 1 and y = 0..cbHeight − 1, the reference indices refIdxLX and the prediction list utilization flags PredFlagLX with X being 0 and 1 are derived as follows:

$$RefIdxL0[xCb + x][yCb + y] = refIdxL0 \quad (8\text{-}525)$$

RefIdxL1[xCb + x][yCb + y] = refIdxL1 (8-526)

PredFlagL0[xCb + x][yCb + y] = predFlagL0 (8-527)

PredFlagL1[xCb + x][yCb + y] = predFlagL1 (8-528)

8.5.3.2 Derivation process for motion vectors and reference indices in affine merge mode

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block,

– The variable availLR specifying left and right neighbouring blocks' availability of luma coding block.

Outputs of this process are:

– the number of control point motion vectors numCpMv,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the luma affine control point motion vector cpMvLX[cpIdx] with X being 0 or 1, and cpIdx = 0 .. numCpMv − 1.

The affine merging candidate list, affineMergeCandList are derived by the following ordered steps:

1. The sample locations ($xNbA_0$, $yNbA_0$), ($xNbA_1$, $yNbA_1$), ($xNbA_2$, $yNbA_2$), ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$), ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$) are derived as follows:

   (xA_0, yA_0) = (xCb − 1, yCb + cbHeight) (8-529)

   (xA_1, yA_1) = (xCb − 1, yCb + cbHeight − 1) (8-530)

   (xA_2, yA_2) = (xCb − 1, yCb) (8-531)

   (xB_0, yB_0) = (xCb + cbWidth , yCb − 1) (8-532)

   (xB_1, yB_1) = (xCb + cbWidth − 1, yCb − 1) (8-533)

   (xB_2, yB_2) = (xCb − 1, yCb − 1) (8-534)

   (xB_3, yB_3) = (xCb, yCb − 1) (8-535)

   (xC_0, yC_0) = (xCb + cbWidth, yCb + cbHeight) (8-536)

   (xC_1, yC_1) = (xCb + cbWidth, yCb + cbHeight − 1) (8-537)

   (xC_2, yC_2) = (xCb + cbWidth, yCb) (8-538)

2. The following applies for (xNbBLK, yNbBLK) with BLK being replaced by A_0, A_1, A_2, B_0, B_1 , B_2, B_3, C_0, C_1, and C_2:

   – The availability derivation process for a block is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbBLK, yNbBLK) as inputs, and the output is assigned to the block availability flag availableBLK.

3. The following applies for (xNbBLK, yNbBLK) with BLK being replaced by C_1, B_3, B_2, C_0, and B_0 if availLR is equal to LR_01, otherwise replaced by A_1, B_1, B_0, A_0, and B_2:

   – When availableBLK is equal to TRUE and MotionModelIdc[xNbBLK][yNbBLK] is greater than 0, availableFlagBLK is set equal to 1.

4. The following applies to update availability flags:

   – If availLR is equal to LR_01,

- If availableFlagB_3 and availableFlagB_2 are both equal to 1 and CbPosX[xNbB$_3$][yNbB$_3$] is equal to CbPosX[xNbB$_2$][yNbB$_2$], and CbPosY[xNbB$_3$][yNbB$_3$] is equal to CbPosY[xNbB$_2$][yNbB$_2$], availableFlagB_2 is set equal to 0.
- If availableFlagC_1 and availableFlagC_0 are both equal to 1 and CbPosX[xNbC$_1$][yNbC$_1$] is equal to CbPosX[xNbC$_0$][yNbC$_0$], and CbPosY[xNbC$_1$][yNbC$_1$] is equal to CbPosY[xNbC$_0$][yNbC$_0$], availableFlagC_0 is set equal to 0.
- If availableFlagB_3 and availableFlagB_0 are both equal to 1 and CbPosX[xNbB$_3$][yNbB$_3$] is equal to CbPosX[xNbB$_0$][yNbB$_0$], and CbPosY[xNbB$_3$][yNbB$_3$] is equal to CbPosY[xNbB$_0$][yNbB$_0$], availableFlagB_0 is set equal to 0.
- If availableFlagC_1 and availableFlagB_0 are both equal to 1 and CbPosX[xNbC$_1$][yNbC$_1$] is equal to CbPosX[xNbB$_0$][yNbB$_0$], and CbPosY[xNbC$_1$][yNbC$_1$] is equal to CbPosY[xNbB$_0$][yNbB$_0$], availableFlagB_0 is set equal to 0.

– Otherwise, the following applies:

- If availableFlagB_1 and availableFlagB_0 are both equal to 1 and CbPosX[xNbB$_1$][yNbB$_1$] is equal to CbPosX[xNbB$_0$][yNbB$_0$], and CbPosY[xNbB$_1$][yNbB$_1$] is equal to CbPosY[xNbB$_0$][yNbB$_0$], availableFlagB_0 is set equal to 0.
- If availableFlagA_1 and availableFlagA_0 are both equal to 1 and CbPosX[xNbA$_1$][yNbA$_1$] is equal to CbPosX[xNbA$_0$][yNbA$_0$], and CbPosY[xNbA$_1$][yNbA$_1$] is equal to CbPosY[xNbA$_0$][yNbA$_0$], availableFlagA_0 is set equal to 0.
- If availableFlagB_1 and availableFlagB_2 are both equal to 1 and CbPosX[xNbB$_1$][yNbB$_1$] is equal to CbPosX[xNbB$_2$][yNbB$_2$], and CbPosY[xNbB$_1$][yNbB$_1$] is equal to CbPosY[xNbB$_2$][yNbB$_2$], availableFlagB_2 is set equal to 0.
- If availableFlagA_1 and availableFlagB_2 are both equal to 1 and CbPosX[xNbA$_1$][yNbA$_1$] is equal to CbPosX[xNbB$_2$][yNbB$_2$], and CbPosY[xNbA$_1$][yNbA$_1$] is equal to CbPosY[xNbB$_2$][yNbB$_2$], availableFlagB_2 is set equal to 0.

5. The following applies for (xNbBLK, yNbBLK) with BLK being replaced by C_1, B_3, B_2, C_0, and B_0 if availLR is equal to LR_01, otherwise replaced by A_1, B_1, B_0, A_0, and B_2:
   - When availableFlagBLK is equal to 1, the following applies:
     - The variable motionModelIdcBLK is set equal to MotionModelIdc[xNbBLK][yNbBLK], (xNb, yNb) is set equal to (CbPosX[xNbBLK][yNbBLK], CbPosY[xNbBLK][yNbBLK]), nbW is set equal to CbWidth[xNbBLK][yNbBLK], nbH is set equal to CbHeight[xNbBLK][yNbBLK], numCpMv is set equal to MotionModelIdc[xNbBLK][yNbBLK] + 1.
     - For X being replaced by either 0 or 1, the following applies:
       - When PredFlagLX[xNbBLK][yNbABLK] is equal to 1, the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (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:

$$predFlagLXBLK = PredFlagLX[xNbBLK][yNbBLK] \quad (8\text{-}539)$$

$$refIdxLXBLK = RefIdxLX[xNbBLK][yNbBLK] \quad (8\text{-}540)$$

6. The derivation process for constructed affine control point motion vector merging candidates as specified in clause 8.5.3.4 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the availability flags availableA_0, availableA_1, availableA_2, availableB_0, availableB_1, availableB_2, availableB_3, availableC_0, availableC_1, availableC_2 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.
7. The initial affine merging candidate list, affineMergeCandList, is constructed as follows:

– i = 0

If availLR is equal to LR_01,

if(availableFlagC_1 && i < 5)

affineMergeCandList[i++] = C_1

if(availableFlagB_3 && i < 5)

affineMergeCandList[i++] = B_3

if(availableFlagB_2 && i < 5)

affineMergeCandList[i++] = B_2

if(availableFlagC_0 && i < 5)

affineMergeCandList[i++] = C_0

if(availableFlagB_0 && i < 5)

affineMergeCandList[i++] = B_0

– Otherwise

if(availableFlagA_1 && i < 5)
affineMergeCandList[i++] = A_1
if(availableFlagB_1 && i < 5)
affineMergeCandList[i++] = B_1
if(availableFlagB_0 && i < 5)
affineMergeCandList[i++] = B_0
if(availableFlagA_0 && i < 5)
affineMergeCandList[i++] = A_0
if(availableFlagB_2 && i < 5)
affineMergeCandList[i++] = B_2

Affine control point motion vector merging candidates follows model-based affine merge candidates:
if(availableFlagConst1 && i < 5)
affineMergeCandList[i++] = Const1
if(availableFlagConst2 && i < 5)
affineMergeCandList[i++] = Const2
if(availableFlagConst3 && i < 5)
affineMergeCandList[i++] = Const3
if(availableFlagConst4 && i < 5)
affineMergeCandList[i++] = Const4
if(availableFlagConst5 && i < 5)
affineMergeCandList[i++] = Const5
if(availableFlagConst6 && i < 5)
affineMergeCandList[i++] = Const6

8. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the affineMergeCandList.

9. When numCurrMergeCand is less than 5, the following is repeated until numCurrMrgeCand is equal to 5, with mvZero[0] and mvZero[1] both being equal to 0:

– The reference indices, the prediction list utilization flags and the motion vectors of $zeroCand_m$ with m equal to (numCurrMergeCand − numOrigMergeCand) are derived as follows:

$refIdxL0ZeroCand_m = 0$ (8-541)

$predFlagL0ZeroCand_m = 1$ (8-542)

$cpMvL0ZeroCand_m[0] = mvZero$ (8-543)

$cpMvL0ZeroCand_m[1] = mvZero$ (8-544)

$cpMvL0ZeroCand_m[2] = mvZero$ (8-545)

$refIdxL1ZeroCand_m$ = (slice_type = = B) ? 0 : −1 (8-546)

$predFlagL1ZeroCand_m$ = (slice_type = = B) ? 1 : 0 (8-547)

$cpMvL1ZeroCand_m$[0] = mvZero (8-548)

$cpMvL1ZeroCand_m$[1] = mvZero (8-549)

$cpMvL1ZeroCand_m$[2] = mvZero (8-550)

$motionModelIdcZeroCand_m$ = 1 (8-551)

– The candidate $zeroCand_m$ with m equal to (numCurrMergeCand − numOrigMergeCand) is added at the end of affineMergeCandList and numCurrMergeCand is incremented by 1 as follows:

affineMergeCandList[numCurrMergeCand++] = $zeroCand_m$ (8-552)

The variables refIdxL0, refIdxL1, predFlagL0, predFlagL1, cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx = 0 .. 2 are derived as follows:

– The following assignments are made with N being the candidate at position affine_merge_idx[xCb][yCb] in the affine merging candidate list affineMergeCandList (N = affineMergeCandList[affine_merge_idx[xCb][yCb]]):

refIdxLX = refIdxLXN (8-553)

predFlagLX = predFlagLXN (8-554)

cpMvLX[0] = cpMvLXN[0] (8-555)

cpMvLX[1] = cpMvLXN[1] (8-556)

cpMvLX[2] = cpMvLXN[2] (8-557)

numCpMv = motionModelIdxN + 1 (8-558)

– The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[x][y] = numCpMv − 1 (8-559)

8.5.3.3 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) (8-560)

log2NbH = Log2(nNbH) (8-561)

– For each X=0,1;

  – a motion vector refinedMvLX[xNb][yNb] is set equal to MvLX[xNb][yNb]

– if yNb >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than xNb >> MaxCbLog2Size11Ratio

refinedMvLX [xNb][yNb] is set equal to refMvLX[xNb][yNb].

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

– If is CTUboundary is equal to TRUE, the following applies:

mvScaleHor = refinedMvLX[xNb][yNb + nNbH − 1][0] << 7 (8-562)

mvScaleVer = refinedMvLX[xNb][yNb + nNbH − 1][1] << 7 (8-563)

dHorX = (refinedMvLX[xNb + nNbW − 1][yNb + nNbH − 1][0] − refinedMvLX[xNb][yNb + nNbH − 1][0])
<< (7 − log2NbW) (8-564)

dVerX = (refinedMvLX[xNb + nNbW − 1][yNb + nNbH − 1][1] − refinedMvLX[xNb][yNb + nNbH − 1][1])
<< (7 − log2NbW) (8-565)

– Otherwise (isCTUboundary is equal to FALSE), the following applies:

mvScaleHor = refinedMvLX[xNb][yNb][0][0] << 7 (8-566)

mvScaleVer = refinedMvLX[xNb][yNb][0][1] << 7 (8-567)

dHorX = (refinedMvLX[xNb + nNbW − 1][yNb][1][0] − refinedMvLX[xNb][yNb][0][0])
<< (7 − log2NbW) (8-568)

dVerX = (refinedMvLX[xNb + nNbW − 1][yNb][1][1] − refinedMvLX[xNb][yNb][0][1])
<< (7 − log2NbW) (8-569)

The variables dHorY and dVerY are derived as follows:

– If CTUboundary is equal to FALSE and MotionModelIdc[xNb][yNb] is equal to 2, the following applies:

dHorY = (refinedMvLX[xNb][yNb + nNbH − 1][2][0] − refinedMvLX[xNb][yNb][2][0])
<< (7 − log2NbH) (8-570)

dVerY = (refinedMvLX[xNb][yNb + nNbH − 1][2][1] − refinedMvLX[xNb][yNb][2][1])
<< (7 − log2NbH) (8-571)

– Otherwise (MotionModelIdc[xNb][yNb] is equal to 1), the following applies,

dHorY = − dVerX (8-572)

dVerY = dHorX (8-573)

The luma affine control point motion vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 are derived as follows:

– The first two control point motion vectors cpMvLX[0] and cpMvLX[1] are derived as follows:

cpMvLX[0][0] = (mvScaleHor + dHorX * (xCb − xNb) + dHorY * (yCb − yNb)) (8-574)

cpMvLX[0][1] = (mvScaleVer + dVerX * (xCb − xNb) + dVerY * (yCb − yNb)) (8-575)

cpMvLX[1][0] = (mvScaleHor + dHorX * (xCb + cbWidth − xNb) + dHorY * (yCb − yNb)) (8-576)

cpMvLX[1][1] = (mvScaleVer + dVerX * (xCb + cbWidth − xNb) + dVerY * (yCb − yNb)) (8-577)

– 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)) (8-578)

cpMvLX[2][1] = (mvScaleVer + dVerX * (xCb − xNb) + dVerY * (yCb + cbHeight − yNb)) (8-579)

– The rounding process for motion vectors as specified in clause 8.5.3.10 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(-2^{15}, $2^{15} - 1$, cpMvLX[cpIdx][0]) (8-580)

cpMvLX[cpIdx][1] = Clip3(-2^{15}, $2^{15} - 1$, cpMvLX[cpIdx][1]) (8-581)

8.5.3.4 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 availableA$_0$, availableA$_1$, availableA$_2$, availableB$_0$, availableB$_1$, availableB$_2$, availableB$_3$, availableC$_0$, availableC$_1$, availableC$_2$

– the sample locations (xNbA$_0$, yNbA$_0$), (xNbA$_1$, yNbA$_1$), (xNbA$_2$, yNbA$_2$), (xNbB$_0$, yNbB$_0$), (xNbB$_1$, yNbB$_1$), (xNbB$_2$, yNbB$_2$), (xNbB$_3$, yNbB$_3$), (xNbC$_0$, yNbC$_0$), (xNbC$_1$, yNbC$_1$), and (xNbC$_2$, yNbC$_2$).

Output of this process are:

– the availability flag of the constructed affine control point motion vector merging candidates 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,

– For each X=0,1 and Y being A$_1$, A$_0$, B$_1$, B$_0$, B$_2$, B$_3$, A$_2$, C$_1$, C$_0$, and C$_2$

  – a motion vector refinedMvLX[xNbY][yNbY] is set equal to MvLX[xNbY][yNbY]

  – if yNbY >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than xNbY >> MaxCbLog2Size11Ratio

    refinedMvLX [xNbY][yNbY] is set equal to refMvLX[xNbY][yNbY].

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 B$_2$, B$_3$, and A$_2$:

  – 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] (8-582)

    predFlagLXCorner[0] = PredFlagLX[xNbTL][yNbTL] (8-583)

    cpMvLXCorner[0] = refinedMvLX[xNbTL][yNbTL] (8-584)

    availableFlagCorner[0] = TRUE (8-585)

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, B0, and C2:

  – 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] (8-586)

    predFlagLXCorner[1] = PredFlagLX[xNbTR][yNbTR] (8-587)

cpMvLXCorner[1] = refinedMvLX[xNbTR][yNbTR] (8-588)

availableFlagCorner[1] = TRUE (8-589)

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 A_1 and A_0:

  – 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] (8-590)

    predFlagLXCorner[2] = PredFlagLX[xNbBL][yNbBL] (8-591)

    cpMvLXCorner[2] = refinedMvLX[xNbBL][yNbBL] (8-592)

    availableFlagCorner[2] = TRUE (8-593)

The fourth (bottom-right or 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 availability flag availableFlagCorner[3] is set equal to FALSE.

– If availLR is equal to LR_01 or LR_11, the following applies for (xNbBR, yNbBR) with BL being replaced by C_1 and C_0:

  – When availableBR is equal to TRUE and availableFlagCorner[3] is equal to FALSE, the following applies with X being 0 and 1:

    refIdxLXCorner[3] = RefIdxLX[xNbBR][yNbBR] (8-594)

    predFlagLXCorner[3] = PredFlagLX[xNbBR][yNbBR] (8-595)

    cpMvLXCorner[3] = refinedMvLX[xNbBR][yNbBR] (8-596)

    availableFlagCorner[3] = TRUE

– Otherwise, 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:

    – The following applies:

      xColBr = xCb + cbWidth (8-597)

      yColBr = yCb + cbHeight (8-598)

    – 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 8.5.2.5 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.

– The variables availableFlagCorner[3], predFlagL0Corner[3], cpMvL0Corner[3] and predFlagL1Corner[3] are derived as follows:

availableFlagCorner[3] = availableFlagL0Col (8-599)

predFlagL0Corner[3] = availableFlagL0Col (8-600)

cpMvL0Corner[3] = mvL0Col (8-601)
predFlagL1Corner[3] = 0 (8-602)

– When slice_type is equal to B, the variables availableFlagCorner[3], predFlagL1Corner[3] and cpMvL1Corner[3] are derived as follows:

availableFlagCorner[3] = availableFlagL0Col || availableFlagL1Col (8-603)

predFlagL1Corner[3] = availableFlagL1Col (8-604)

cpMvL1Corner[3] = mvL1Col (8-605)

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 (8-606)

refIdxLXConst1 = refIdxLXCorner[0] (8-607)

cpMvLXConst1[0] = cpMvLXCorner[0] (8-608)

cpMvLXConst1[1] = cpMvLXCorner[1] (8-609)

cpMvLXConst1[2] = cpMvLXCorner[2] (8-610)

   – 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.
2. 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:

$$predFlagLXConst2 = 1 \quad (8\text{-}611)$$

$$refIdxLXConst2 = refIdxLXCorner[\ 0\ ] \quad (8\text{-}612)$$

$$cpMvLXConst2[\ 0\ ] = cpMvLXCorner[\ 0\ ] \quad (8\text{-}613)$$

$$cpMvLXConst2[\ 1\ ] = cpMvLXCorner[\ 1\ ] \quad (8\text{-}614)$$

$$cpMvLXConst2[\ 2\ ] = cpMvLXCorner[\ 3\ ] + cpMvLXCorner[\ 0\ ] - cpMvLXCorner[\ 1\ ] \quad (8\text{-}615)$$

$$cpMvLXConst2[\ 2\ ][\ 0\ ] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst2[\ 2\ ][\ 0\ ]) \quad (8\text{-}616)$$

$$cpMvLXConst2[\ 2\ ][\ 1\ ] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst2[\ 2\ ][\ 1\ ]) \quad (8\text{-}617)$$

– 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.

3. 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:

$$predFlagLXConst3 = 1 \quad (8\text{-}618)$$

$$refIdxLXConst3 = refIdxLXCorner[\ 0\ ] \quad (8\text{-}619)$$

$$cpMvLXConst3[\ 0\ ] = cpMvLXCorner[\ 0\ ] \quad (8\text{-}620)$$

$$cpMvLXConst3[\ 1\ ] = cpMvLXCorner[\ 3\ ] + cpMvLXCorner[\ 0\ ] - cpMvLXCorner[\ 2\ ] \quad (8\text{-}621)$$

$$cpMvLXConst3[\ 1\ ][\ 0\ ] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst3[\ 1\ ][\ 0\ ]) \quad (8\text{-}622)$$

$$cpMvLXConst3[\ 1\ ][\ 1\ ] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst3[\ 1\ ][\ 1\ ]) \quad (8\text{-}623)$$

$$cpMvLXConst3[\ 2\ ] = cpMvLXCorner[\ 2\ ] \quad (8\text{-}624)$$

– 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.

4. 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 the 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:

$$predFlagLXConst4 = 1 \tag{8-625}$$

$$refIdxLXConst4 = refIdxLXCorner[\ 1\ ] \tag{8-626}$$

$$cpMvLXConst4[\ 0\ ] = cpMvLXCorner[\ 1\ ] + cpMvLXCorner[\ 2\ ] - cpMvLXCorner[\ 3\ ] \tag{8-627}$$

$$cpMvLXConst4[\ 0\ ][\ 0\ ] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst4[\ 0\ ][\ 0\ ]) \tag{8-628}$$

$$cpMvLXConst4[\ 0\ ][\ 1\ ] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst4[\ 0\ ][\ 1\ ]) \tag{8-629}$$

$$cpMvLXConst4[\ 1\ ] = cpMvLXCorner[\ 1\ ] \tag{8-630}$$

$$cpMvLXConst4[\ 2\ ] = cpMvLXCorner[\ 2\ ] \tag{8-631}$$

   - 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:

5. 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 the 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:

$$predFlagLXConst5 = 1 \tag{8-632}$$

$$refIdxLXConst5 = refIdxLXCorner[\ 0\ ] \tag{8-633}$$

$$cpMvLXConst5[\ 0\ ] = cpMvLXCorner[\ 0\ ] \tag{8-634}$$

$$cpMvLXConst5[\ 1\ ] = cpMvLXCorner[\ 1\ ] \tag{8-635}$$

   - 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.

6. 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 the 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:

$$cpMvLXCorner[\ 1\ ][\ 0\ ] = (cpMvLXCorner[\ 0\ ][\ 0\ ] << 7\ ) + ((\ cpMvLXCorner[\ 2\ ][\ 1\ ] - cpMvLXCorner[\ 0\ ][\ 1\ ]\ ) << (\ 7 + Log2(\ cbHeight / cbWidth\ )\ )\ ) \quad (8\text{-}636)$$

$$cpMvLXCorner[\ 1\ ][\ 1\ ] = (cpMvLXCorner[\ 0\ ][\ 1\ ] << 7\ ) + ((\ cpMvLXCorner[\ 2\ ][\ 0\ ] - cpMvLXCorner[\ 0\ ][\ 0\ ]\ ) << (\ 7 + Log2(\ cbHeight / cbWidth\ )\ )\ ) \quad (8\text{-}637)$$

       - The rounding process for motion vectors as specified in clause 8.5.3.10 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:

$$predFlagLXConst6 = 1 \quad (8\text{-}638)$$

$$refIdxLXConst6 = refIdxLXCorner[\ 0\ ] \quad (8\text{-}639)$$

$$cpMvLXConst6[\ 0\ ] = cpMvLXCorner[\ 0\ ] \quad (8\text{-}640)$$

$$cpMvLXConst6[\ 1\ ] = cpMvLXCorner[\ 1\ ] \quad (8\text{-}641)$$

$$cpMvLXConst6[\ 1\ ][\ 0\ ] = Clip3(\ -2^{15},\ 2^{15} - 1,\ cpMvLXConst6[\ 1\ ][\ 0\ ]\ ) \quad (8\text{-}642)$$

$$cpMvLXConst6[\ 1\ ][\ 1\ ] = Clip3(\ -2^{15},\ 2^{15} - 1,\ cpMvLXConst6[\ 1\ ][\ 1\ ]\ ) \quad (8\text{-}643)$$

   - 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.

8.5.3.5 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,
- the variable availLR specifying left and right neighbouring blocks' availability of luma coding block.

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, availableFlagB, and availableFlagC are both set equal to FALSE.
3. The sample locations ($xNbA_0$, $yNbA_0$), ($xNbA_1$, $yNbA_1$), ($xNbA_2$, $yNbA_2$), ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$), ($xNbB_2$, $yNbB_2$), ($xNbC_0$, $yNbC_0$), ($xNbC_1$, $yNbC_1$), and ($xNbC_2$, $yNbC_2$) are derived as follows:

$$(xA_0, yA_0) = (xCb - 1, yCb + cbHeight) \quad (8\text{-}644)$$

$$(xA_1, yA_1) = (xCb - 1, yCb + cbHeight - 1) \quad (8\text{-}645)$$

$$(xB_0, yB_0) = (xCb + cbWidth , yCb - 1) \quad (8\text{-}646)$$

$$(xB_1, yB_1) = (xCb + cbWidth - 1, yCb - 1) \quad (8\text{-}647)$$

$$(xB_2, yB_2) = (xCb - 1, yCb - 1) \quad (8\text{-}648)$$

$$(xC_0, yC_0) = (xCb + cbWidth, yCb + cbHeight) \quad (8\text{-}649)$$

$$(xC_1, yC_1) = (xCb + cbWidth, yCb + cbHeight - 1) \quad (8\text{-}650)$$

$$(xC_2, yC_2) = (xCb + cbWidth, yCb) \quad (8\text{-}651)$$

4. The following applies for ($xNbA_k$, $yNbA_k$) from ($xNbA_0$, $yNbA_0$) to ($xNbA_1$, $yNbA_1$):
   - The availability derivation process for a block is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbA_k$, $yNbA_k$) as inputs, and the output is assigned to the block availability flag $availableA_k$.
     - When $availableA_k$ is equal to TRUE and MotionModelIdc[$xNbA_k$][$yNbA_k$] is greater than 0 and availableFlagA is equal to FALSE, the following applies:
       - The variable (xNb, yNb) is set equal to (CbPosX[$xNbA_k$][$yNbA_k$], CbPosY[$xNbA_k$][$yNbA_k$]), nbW is set equal to CbWidth[$xNbA_k$][$yNbA_k$],and nbH is set equal to CbHeight[$xNbA_k$][$yNbA_k$].
       - If PredFlagLX[$xNbA_k$][$yNbA_k$] is equal to 1 and RefIdxLX[$xNbA_k$][$yNbA_k$] is equal to refIdxLX, 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 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (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 following assignments are made:

$$cpMvpListLX[numCpMvpCandLX][0] = cpMvpLX[0] \quad (8\text{-}652)$$

$$cpMvpListLX[numCpMvpCandLX][1] = cpMvpLX[1] \quad (8\text{-}653)$$

$$cpMvpListLX[numCpMvpCandLX][2] = cpMvpLX[2] \quad (8\text{-}654)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (8\text{-}655)$$

5. The following applies for ($xNbB_k$, $yNbB_k$) from ($xNbB_0$, $yNbB_0$) to ($xNbB_2$, $yNbB_2$):
   - The availability derivation process for a block is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_k$, $yNbB_k$) as inputs, and the output is assigned to the block availability flag $availableB_k$.
     - When $availableB_k$ is equal to TRUE and MotionModelIdc[$xNbB_k$][$yNbB_k$] is greater than 0 and availableFlagB is equal to FALSE, the following applies:
       - The variable (xNb, yNb) is set equal to (CbPosX[$xNbB_k$][$yNbB_k$], CbPosY[$xNbB_k$][$yNbB_k$]), nbW is set equal to CbWidth[$xNbB_k$][$yNbB_k$],and nbH is set equal to CbHeight[$xNbB_k$][$yNbB_k$].

- If PredFlagLX[$xNbB_k$][$yNbB_k$] is equal to 1 and RefIdxLX[$xNbB_k$][$yNbB_k$] is equal to refIdxLX, 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 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (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 following assignments are made:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (8\text{-}656)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 1\ ] = cpMvpLX[\ 1\ ] \quad (8\text{-}657)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 2\ ] = cpMvpLX[\ 2\ ] \quad (8\text{-}658)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (8\text{-}659)$$

6. When numCpMvpCandLX is less than 2, the following applies for ($xNbC_k$, $yNbC_k$) from ($xNbC_0$, $yNbC_0$) to ($xNbC_1$, $yNbC_1$):
   - The availability derivation process for a block is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbC_k$, $yNbC_k$) as inputs, and the output is assigned to the block availability flag $availableC_k$.
     - When $availableC_k$ is equal to TRUE and MotionModelIdc[$xNbC_k$][$yNbC_k$] is greater than 0 and availableFlagC is equal to FALSE, the following applies:
       - The variable (xNb, yNb) is set equal to (CbPosX[$xNbC_k$][$yNbC_k$], CbPosY[$xNbC_k$][$yNbC_k$]), nbW is set equal to CbWidth[$xNbC_k$][$yNbC_k$],and nbH is set equal to CbHeight[$xNbC_k$][$yNbC_k$].
       - If PredFlagLX[$xNbC_k$][$yNbC_k$] is equal to 1 and RefIdxLX[$xNbC_k$][$yNbC_k$]] is equal to refIdxLX, the following applies:
         - The variable availableFlagC is set equal to TRUE
         - The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (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 following assignments are made:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (8\text{-}660)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 1\ ] = cpMvpLX[\ 1\ ] \quad (8\text{-}661)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 2\ ] = cpMvpLX[\ 2\ ] \quad (8\text{-}662)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (8\text{-}663)$$

7. 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 8.5.3.6 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX 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:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (8\text{-}664)$$

cpMvpListLX[numCpMvpCandLX][1] = cpMvpLX[1] (8-665)

cpMvpListLX[numCpMvpCandLX][2] = cpMvpLX[2] (8-666)

numCpMvpCandLX = numCpMvpCandLX + 1 (8-667)

8. The following applies for cpIdx = 2 . . 0:

   – When numCpMvpCandLX is less than 2 and availableFlagLX[cpIdx] is equal to 1, the following assignments are made:

     cpMvpListLX[numCpMvpCandLX][0] = cpMvpLX[cpIdx] (8-668)

     cpMvpListLX[numCpMvpCandLX][1] = cpMvpLX[cpIdx] (8-669)

     cpMvpListLX[numCpMvpCandLX][2] = cpMvpLX[cpIdx] (8-670)

     numCpMvpCandLX = numCpMvpCandLX + 1 (8-671)

9. 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 (8-672)

   cpMvpListLX[numCpMvpCandLX][1] = mvZero (8-673)

   cpMvpListLX[numCpMvpCandLX][2] = mvZero (8-674)

   numCpMvpCandLX = numCpMvpCandLX + 1 (8-675)

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]] (8-676)

8.5.3.6 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 coding 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 candidates 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 candidates cpMvLX[cpIdx] with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

– The sample locations ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$) and ($xNbA_2$, $yNbA_2$) are set equal to (xCb − 1, yCb − 1), (xCb , yCb − 1) and (xCb − 1, yCb), respectively.

– The sample locations ($xNbB_1$, $yNbB_1$) and ($xNbB_0$, $yNbB_0$) are set equal to (xCb + cbWidth − 1, yCb − 1) and (xCb + cbWidth, yCb − 1), respectively.

– The sample locations ($xNbA_1$, $yNbA_1$) and ($xNbA_0$, $yNbA_0$) are set equal to (xCb − 1, yCb + cbHeight − 1) and (xCb − 1, yCb + cbHeight), respectively.

– For each X=0,1 and Y being A_1, A_0, B_1, B_0, B_2, B_3, and A_2

  – a motion vector refinedMvLX[xNbY][yNbY] is set equal to MvLX[xNbY][yNbY]

  – if yNbY >> MaxCbLog2Size11Ratio is not equal to yCb >> MaxCbLog2Size11Ratio and xCb >> MaxCbLog2Size11Ratio is not smaller than xNbY >> MaxCbLog2Size11Ratio

    refinedMvLX [xNbY][yNbY] is set equal to refMvLX[xNbY][yNbY].

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 availability flag availableFlagLX[0] is set equal to 0 and both components of cpMvLX[0] are set equal to 0.
2. The following applies for (xNbTL, yNbTL) with TL being replaced by B_2, B_3, and A_2:
   - The availability derivation process for a coding block 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 RefIdxLX[xNbTL][yNbTL] is equal to refIdxLX, availableFlagLX[0] is set equal to 1 and the following assignments are made:

       cpMvLX[0] = refinedMvLX[xNbTL][yNbTL] (8-677)

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 availability flag availableFlagLX[1] is set equal to 0 and both components of cpMvLX[1] are set equal to 0.
2. The following applies for (xNbTR, yNbTR) withTR being replaced by B_1, B_0 and C_2:
   - The availability derivation process for a coding block 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 RefIdxLX[xNbTR][yNbTR] is equal to refIdxLX, availableFlagLX[1] is set equal to 1 and the following assignments are made:

       cpMvLX[1] = refinedMvLX[xNbTR][yNbTR] (8-678)

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 availability flag availableFlagLX[2] is set equal to 0 and both components of cpMvLX[2] are set equal to 0.
2. The following applies for (xNbBL, yNbBL) with BL being replaced by A_1 and A_0:
   - The availability derivation process for a coding block 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 RefIdxLX[xNbBL][yNbBL] is equal to refIdxLX, availableFlagLX[2] is set equal to 1 and the following assignments are made:

       cpMvLX[2] = refinedMvLX[xNbBL][yNbBL] (8-679)

The fourth (bottom-right) control point motion vector cpMvLX[3] and the availability flag availableFlagLX[3] are derived in the following ordered steps:

3. The sample locations ($xNbC_1$, $yNbC_1$) and ($xNbC_0$, $yNbC_0$) are set equal to (xCb + cbWidth, yCb + cbHeight − 1) and (xCb + cbWidth, yCb + cbHeight), respectively.
4. The availability flag availableFlagLX[3] is set equal to 0 and both components of cpMvLX[3] are set equal to 0.
5. The following applies for (xNbBR, yNbBR) with BR being replaced by C_1 and C_0:
   - The availability derivation process for a coding block 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 (xNbBR, yNbBR) as inputs, and the output is assigned to the coding block availability flag availableBR.

– When availableBR is equal to TRUE and availableFlagLX[2] is equal to 0, the following applies:

  – If PredFlagLX[xNbBR][yNbBR] is equal to 1, andRefIdxLX[xNbBR][yNbBR] is equal to refIdxLX, availableFlagLX[3] is set equal to 1 and the following assignments are made:

$$cpMvLX[3] = MvLX[xNbBR][yNbBR] \quad (8\text{-}680)$$

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, availableFlagLX[2] is equal to 0 and availableFlagLX[3] is equal to 1, availableConsFlagLX is set equal to 1, and following applies:

  – $cpMvLX[2] = Clip3(-2^{15}, 2^{15} - 1, cpMvLX[3] + cpMvLX[0] - cpMvLX[1])$ (8-681)

– 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.

8.5.3.7 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.

Outputs of this process are:

– the number of luma coding subblocks in horizontal direction numSbXLX and in vertical direction numSbYLX with X being 0 or 1,

– the size of luma coding subblocks in horizontal direction sizeSbXLX and in vertical direction sizeSbYLX with X being 0 or 1,

– the luma subblock motion vector array mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXLX − 1, ySbIdx = 0 .. numSbYLX − 1 and X being 0 or 1,

– the chroma subblock motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXLX − 1, ySbIdx = 0 .. numSbYLX − 1 and X being 0 or 1.

The variables sizeSbXLX, sizeSbYLX, numSbXLX, numSbYLX, are derived according to 8.5.3.8.

When predFlagLX is equal to 1, the following applies for X being 0 and 1:

– Horizontal change of motion vector dX, vertical change of motion vector dY and base motion vector mvBaseScaled are derived by invoking the process specified in clause 8.5.3.9 with the luma coding block width cbWidth, the luma coding block height cbHeight, number of control point motion vectors numCpMv and the control point motion vectors cpMvLX[cpIdx] with cpIdx = 0..numCpMv − 1 as inputs.

– For ySbIdx = 0..numSbYLX – 1:

  – For xSbIdx = 0.. numSbXLX – 1:

    – The luma motion vector mvLX[xSbIdx][ySbIdx] is derived as follows:

$$xPosSb = sizeSbXLX * xSbIdx + (sizeSbXLX >> 1) \quad (8\text{-}682)$$

$$yPosSb = sizeSbYLX * ySbIdx + (sizeSbYLX >> 1) \quad (8\text{-}683)$$

$$mvLX[xSbIdx][ySbIdx][0] = (mvBaseScaled[0] + dX[0] * xPosSb + dY[0] * yPosSb) \quad (8\text{-}684)$$

mvLX[xSbIdx][ySbIdx][1] = (mvBaseScaled[1] + dX[1] * xPosSb + dY[1] * yPosSb) (8-685)

– The rounding process for motion vectors as specified in clause 8.5.3.10 is invoked the with mvX set equal to mvLX[xSbIdx][ySbIdx], rightShift set equal to 5, 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(-2^{17}, $2^{17} - 1$, mvLX[xSbIdx][ySbIdx][0]) (8-686)

mvLX[xSbIdx][ySbIdx][1] = Clip3(-2^{17}, $2^{17} - 1$, mvLX[xSbIdx][ySbIdx][1]) (8-687)

– The derivation process for chroma motion vectors in clause 8.5.2.14 is invoked with mvLX[xSbIdx][ySbIdx] as input, and the chroma motion vector mvCLX[xSbIdx][ySbIdx] as output.

8.5.3.8 Derivation process for affine subblock size

Inputs to this process are:

– 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.

Outputs of this process are:

– the size of luma coding subblocks in horizontal direction sizeSbX and in vertical direction sizeSbY,

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY.

The variable sizeSbX is derrived as follows:

sizeSbX = max(((cbWidth >> 2) / mvWx), 1)

```
while (cbWidth % sizeSbX)
{
    sizeSbX – –
}
```

sizeSbX = max(4, sizeSbX)

Where the variable mvWx is derived as follows:

mvWx = max(abs(cpMvLX[1][0] – cpMvLX[0][0]), abs(cpMvLX[1][1] – ac_mv[0][1]))

The varible sizeSbY is derrived as follows:

– If thr number of control point motion vectors numCpMv is equeal to 2:

sizeSbY = min(sizeSbX, cbHeight)

– Otherwise the following applyes:

sizeSbY = max(((cbHeight >> 2) / mvWy), 1)

```
while (cbHeight % sizeSbY)
{
    sizeSbY – –
}
```

sizeSbY = max(4, sizeSbY)

Where the variable mvWy is derived as follows:

mvWy = max(abs(cpMvLX[2][0] – cpMvLX[0][0]), abs(cpMvLX[2][1] – ac_mv[0][1]))

– The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

numSbX = cbWidth / sizeSbX

numSbY = cbHeight / sizeSbY

8.5.3.9 Derrivation process for affine motion model parameters from control point motion vectors

Inputs to this process are:

– 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.

Outputs of this process are:

– horizontal change of motion vector dX,

– vertical change of motion vector dY,

– motion vector mvBaseScaled corresponding to the top left corner of the luma coding block.

The variables log2CbW and log2CbH are derived as follows:

log2CbW = Log2(cbWidth) (8-688)

log2CbH = Log2(cbHeight) (8-689)

Horizontal change of motion vector dX is derived as follows:

dX[0] = (cpMvLX[1][0] − cpMvLX[0][0]) << (7 − log2CbW) (8-690)

dX[1] = (cpMvLX[1][1] − cpMvLX[0][1]) << (7 − log2CbW) (8-691)

Vertical change of motion vector dY is derived as follows:

– If numCpMv is equal to 3, dY is derived as follow:

dY[0] = (cpMvLX[2][0] − cpMvLX[0][0]) << (7 − log2CbH) (8-692)

dY[1] = (cpMvLX[2][1] − cpMvLX[0][1]) << (7 − log2CbH) (8-693)

– Otherwise (numCpMv is equal to 2), dY is derived as follows:

dY[0] = −dX[1] (8-694)

dY[1] = dX[0] (8-695)

Motion vector mvBaseScaled corresponding to the top left corner of the luma coding block is derived as follows:

mvBaseScaled[0] = cpMvLX[0][0] << 7 (8-696)

mvBaseScaled[1] = cpMvLX[0][1] << 7 (8-697)

8.5.3.10 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)) (8-698)

mvX[0] = ((mvX[0] + offset – (mvX[0] >= 0)) >> rightShift) << leftShift (8-699)

mvX[1] = ((mvX[0] + offset – (mvX[1] >= 0)) >> rightShift) << leftShift (8-700)

8.5.4 Decoding process for inter prediction samples

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 nCbW and nCbH specifying the width and the height of the current luma coding block,

– variables numSbXL0, numSbXL1 and numSbYL0, numSbYL1 specifying the number of luma coding subblocks in horizontal and vertical direction,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] with xSbIdx = 0 .. numSbXL0 − 1, and ySbIdx = 0 .. numSbYL0 − 1, and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0 .. numSbXL1 − 1, and ySbIdx = 0 .. numSbYL1 − 1

– the refined motion vectors refMvL0[xSbIdx][ySbIdx] and refMvL1[xSbIdx][ySbIdx] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY – 1,

– the chroma motion vectors in 1/32 fractional-sample accuracy mvCL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXL0 − 1, ySbIdx = 0..numSbYL0 – 1, and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbXL1 − 1, ySbIdx = 0..numSbYL1 − 1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags, predFlagL0, and predFlagL1,

– a variable dmvrAppliedFlag indicating the use of DMVR,

– the number of control point motion vectors numCpMv,

– the control point motion vectors cpMvL0[cpIdx] with cpIdx = 0..numCpMv − 1 and cpMvL1[cpIdx] with cpIdx = 0..numCpMv – 1.

Outputs of this process are:

– an ($nCbW_L$)x($nCbH_L$) array $predSamples_L$ of luma prediction samples, where $nCbW_L$ and $nCbH_L$ are derived as specified below,

– when ChromaArrayType is not equal to 0, an ($nCbW_C$)x($nCbH_C$) array $preSamples_{Cb}$ of chroma prediction samples for the component Cb, where $nCbW_C$ and $nCbH_C$ are derived as specified below,

– when ChromaArrayType is not equal to 0, an ($nCbW_C$)x($nCbH_C$) array $predSamples_{Cr}$ of chroma residual samples for the component Cr, where $nCbW_C$ and $nCbH_C$ are derived as specified below.

The variables $nCbW_L$ and $nCbH_L$ are set equal to nCbW and nCbH, respectively, and the variables $nCbW_C$ and $nCbH_C$ are set equal to nCbW / SubWidthC and nCbH / SubHeightC, respectively.

Let $predSamplesL0_L$ and $predSamplesL1_L$ be (nCbW)x(nCbH) arrays of predicted luma sample values and, when ChromaArrayType is not equal to 0, $predSampleL0_{Cb}$, $predSampleL1_{Cb}$, $predSampleL0_{Cr}$, and $predSampleL1_{Cr}$ be (nCbW / SubWidthC)x(nCbH / SubHeightC) arrays of predicted chroma sample values.

For X being each of 0 and 1, when predFlagLX is equal to 1, the following applies:

– The reference picture consisting of an ordered two-dimensional array $refPicLX_L$ of luma samples and, when ChromaArrayType is not equal to 0, two ordered two-dimensional arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$ of chroma samples is derived by invoking the process specified in clause 8.5.4.1 with refIdxLX as input.

– The width and the height of the current luma coding sublock sbWidth, sbHeight are derived as follows:

$$sbWidth = nCbW / numSbXLX \quad (8\text{-}701)$$

$$sbHeight = nCbH / numSbYLX \quad (8\text{-}702)$$

– If affine_flag is equal to 1 and one of the variables sbWidth, sbHeight is less than 8, the following applies:

  – Horizontal change of motion vector dX, vertical change of motion vector dY and base motion vector mvBaseScaled are derived by invoking the process specified in clause 8.5.3.9 with the luma coding block width nCbW, the luma coding block height nCbH, number of control point motion vectors numCpMv and the control point motion vectors cpMvLX[cpIdx] with cpIdx = 0..numCpMv − 1 as inputs.

  – The array $predSamplesLX_L$ is derived by invoking interpolation process for enhanced interpolation filter specified in clause 8.5.4.3 with the luma locations (xSb, ySb), the luma coding block width nCbW, the luma coding block height nCbH, horizontal change of motion vector dX, vertical change of motion vector dY, base

motion vector mvBaseScaled, the reference array refPicLX$_L$, sample bitDepth bitDepth$_Y$, picture width pic_width_in_luma_samples and height pic_height_in_luma_samples as inputs.

- If ChromaArrayType is not equal to 0, the following applies:
  - mvBaseScaled [0] = mvBaseScaled [0] / SubWidthC (8-703)
  - mvBaseScaled [1] = mvBaseScaled [1] / SubHeightC (8-704)
  - The arrays predSamplesLX$_{Cb}$, is derived by invoking iterpolation process for enhanced interpolation filter specified in clause 8.5.4.3 with the chroma locations (xSb / SubWidthC, ySb / SubHeightC), the chroma coding block width nCbW / subWidthC, the luma coding block height nCbH / SubHeightC, horizontal change of motion vector dX, vertical change of motion vector dY, base motion vector mvBaseScaled, the reference array refPicLX$_{Cb}$, sample bitDepth bitDepth$_C$, picture width pic_width_in_luma_samples / SubWidthC and height pic_height_in_luma_samples / SubHeightC as inputs.
  - The arrays predSamplesLX$_{Cb}$, is derived by invoking iterpolation process for enhanced interpolation filter specified in clause 8.5.4.3 with the chroma locations (xSb / SubWidthC, ySb / SubHeightC), the chroma coding block width nCbW / subWidthC, the chroma coding block height nCbH / SubHeightC, horizontal change of motion vector dX, vertical change of motion vector dY, base motion vector mvBaseScaled, the reference array refPicLX$_{Cb}$, sample bitDepth bitDepth$_C$, picture width pic_width_in_luma_samples / SubWidthC and height pic_height_in_luma_samples / SubHeightC as inputs.

- Otherwise, for each coding subblock at subblock index (xSbIdx, ySbIdx) with xSbIdx = 0 .. numSbXLX − 1, and ySbIdx = 0 .. numSbYLX − 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) (8-705)

  - 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
  - The arrays predSamplesLX$_L$ and, when ChromaArrayType is not equal to 0, predSamplesLX$_{Cb}$, and predSamplesLX$_{Cr}$ are derived by invoking the fractional sample interpolation process specified in clause 8.5.4.2 with the luma locations (xSb, ySb), the luma coding block width sbWidth, the luma coding block height sbHeight, the luma motion vector offset mvOffset, the refined motion vectors refMvLX [xSb][ySb] and, when ChromaArrayType is not equal to 0, mvCLX[xSb][ySb], and the reference arrays refPicLX$_L$, refPicLX$_{Cb}$, and refPicLX$_{Cr}$ as inputs.

The prediction samples inside the current luma coding block, predSamples$_L$[x$_L$][y$_L$] with x$_L$ = 0..nCbW − 1 and y$_L$ = 0..nCbH − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.4.4 with the coding block width nCbW, the coding block height nCbH and the sample arrays predSamplesL0$_L$ and predSamplesL1$_L$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx equal to 0 as inputs.

When ChromaArrayType is not equal to 0, the prediction samples inside the current chroma component Cb coding block, predSamples$_{Cb}$[x$_C$][y$_C$] with x$_C$ = 0..nCbW / SubWidthC − 1 and y$_C$ = 0..nCbH / SubHeightC − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.4.4 with the coding block width nCbW set equal to nCbW / SubWidthC, the coding block height nCbH set equal to nCbH / SubHeightC, the sample arrays predSamplesL0$_{Cb}$ and predSamplesL1$_{Cb}$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx equal to 1 as inputs.

When ChromaArrayType is not equal to 0, the prediction samples inside the current chroma component Cr coding block, predSamples$_{Cr}$[x$_C$][y$_C$] with x$_C$ = 0..nCbW / SubWidthC − 1 and y$_C$ = 0..nCbH / SubHeightC − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.4.4 with the coding block width nCbW set equal to nCbW / SubWidthC, the coding block height nCbH set equal to nCbH / SubHeightC, the sample arrays

predSamplesL0$_{Cr}$ and predSamplesL1$_{Cr}$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx equal to 2 as inputs.

8.5.4.1 Reference picture selection process

Input to this process is a reference index refIdxLX.

Output of this process is a reference picture consisting of a two-dimensional array of luma samples refPicLX$_L$ and, when ChromaArrayType is not equal to 0, two two-dimensional arrays of chroma samples refPicLX$_{Cb}$ and refPicLX$_{Cr}$.

The output reference picture RefPicListX[refIdxLX] consists of a pic_width_in_luma_samples by pic_height_in_luma_samples array of luma samples refPicLX$_L$ and, when ChromaArrayType is not equal to 0, two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples refPicLX$_{Cb}$ and refPicLX$_{Cr}$.

The reference picture sample arrays refPicLX$_L$, refPicLX$_{Cb}$, and refPicLX$_{Cr}$ correspond to decoded sample arrays S_L, S_{Cb}, and S_{Cr} derived in clause 8.8 for a previously-decoded picture.

8.5.4.2 Fractional sample interpolation process

8.5.4.2.1 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 sbWidthLX specifying the width of the current coding subblock in luma samples,

– a variable sbHeightLX specifying the height of the current coding subblock in luma samples,

– a refined luma motion vector refMvLX given in 1/16-luma-sample units,

– a motion vector offset mvOffset given in 1/16-luma-sample units,

– a chroma motion vector mvCLX given in 1/32-chroma-sample units,

– the selected reference picture sample array refPicLX$_L$ and the arrays refPicLX$_{Cb}$ and refPicLX$_{Cr}$.

Outputs of this process are:

– an (sbWidthLX)x(sbHeightLX) array predSamplesLX$_L$ of prediction luma sample values,

– two (sbWidthLX / 2)x(sbHeightLX / 2) arrays predSamplesLX$_{Cb}$ and predSamplesLX$_{Cr}$ of prediction chroma sample values,

– The motion vector mvLX is set equal to (refMvLX − mvOffset).

Let (xInt$_L$, yInt$_L$) be a luma location given in full-sample units and (xFrac$_L$, yFrac$_L$) 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$_L$, refPicLX$_{Cb}$ and refPicLX$_{Cr}$.

The top-left coordinate of the bounding block for reference sample padding (xSbIntL, ySbIntL) is set equal to ((xSb + (mvLX[0] >> 4), ySb + (mvLX[1] >> 4)).

The top-left coordinate of the chroma bounding block for reference sample padding (xSbIntC, ySbIntC) is set equal to ((xSb / SubWidthC) + (mvLX[0] >> 5), (ySb / SubHeightC) + (mvLX[1] >> 5)).

For each luma sample location (x_L = 0..sbWidth − 1, y_L = 0..sbHeight − 1) inside the prediction luma sample array predSamplesLX$_L$, the corresponding prediction luma sample value predSamplesLX$_L$[x_L][y_L] is derived as follows:

– The variables xInt$_L$, yInt$_L$, xFrac$_L$ and yFrac$_L$ are derived as follows:

$$xInt_L = xSb + (refMvLX [0] >> 4) + x_L \quad (8\text{-}706)$$

$$yInt_L = ySb + (refMvLX [1] >> 4) + y_L \quad (8\text{-}707)$$

$$xFrac_L = refMvLX [0] \& 15 \quad (8\text{-}708)$$

$$yFrac_L = refMvLX [1] \& 15 \quad (8\text{-}709)$$

– The prediction luma sample value predSamplesLX$_L$[x$_L$][y$_L$] is derived by invoking the process specified in clause 8.5.4.2.2 with (xInt$_L$, yInt$_L$), (xFrac$_L$, yFrac$_L$), (xSbInt$_L$, ySbInt$_L$), sbWidth, sbHeight and refPicLX$_L$ as inputs.

Let (xInt$_C$, yInt$_C$) be a chroma location given in full-sample units and (xFrac$_C$, yFrac$_C$) 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$_{Cb}$ and refPicLX$_{Cr}$.

For each chroma sample location (x$_C$ = 0..sbWidth / 2 − 1, y$_C$ = 0.. sbHeight / 2 − 1) inside the prediction chroma sample arrays predSamplesLX$_{Cb}$ and predSamplesLX$_{Cr}$, the corresponding prediction chroma sample values predSamplesLX$_{Cb}$[x$_C$][y$_C$] and predSamplesLX$_{Cr}$[x$_C$][y$_C$] are derived as follows:

– The variables xInt$_C$, yInt$_C$, xFrac$_C$ and yFrac$_C$ are derived as follows:

$$xInt_C = (xSb / SubWidthC) + (refMvLX [0] >> 5) + x_C \quad (8\text{-}710)$$

$$yInt_C = (ySb / SubHeightC) + (refMvLX [1] >> 5) + y_C \quad (8\text{-}711)$$

$$xFrac_C = refMvLX [0] \,\&\, 31 \quad (8\text{-}712)$$

$$yFrac_C = refMvLX [1] \,\&\, 31 \quad (8\text{-}713)$$

– The prediction sample value predSamplesLX$_{Cb}$[x$_C$][y$_C$] is derived by invoking the process specified in clause 8.5.4.2.3 with (xInt$_C$, yInt$_C$), (xFrac$_C$, yFrac$_C$), (xSbIntC, ySbIntC), sbWidth, sbHeight and refPicLX$_{Cb}$ as inputs.

– The prediction sample value predSamplesLX$_{Cr}$[x$_C$][y$_C$] is derived by invoking the process specified in clause 8.5.4.2.3 with (xInt$_C$, yInt$_C$), (xFrac$_C$, yFrac$_C$), (xSbIntC, ySbIntC), sbWidth/2, sbHeight/2 and refPicLX$_{Cr}$ as inputs.

8.5.4.2.2 Luma sample interpolation process

Inputs to this process are:

– a luma location in full-sample units (xInt$_L$, yInt$_L$),

– a luma location in fractional-sample units (xFrac$_L$, yFrac$_L$),

– a luma location in full-sample units (xSbInt$_L$, ySbInt$_L$) specifying the top-left sample of the bounding block for reference sample padding relative to the top-left luma sample of the reference picture,– the luma reference sample array refPicLX$_L$,

– a variable sbWidth specifying the width of the current subblock,

– a variable sbHeight specifying the height of the current subblock.

Output of this process is a predicted luma sample value predSampleLX$_L$.

The variables shift1, shift2 and offset2 are derived as follows:

– The variable shift1 is set equal to Min(4, BitDepth$_Y$ − 8), and the variable shift2 is set equal to Max(8, 20 – BitDepth$_Y$).

– 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 f$_L$[p] for each 1/16 fractional sample position p equal to xFrac$_L$ or yFrac$_L$ are specified in Table 8-9 or Table 8-10.

The luma locations in full-sample units (xInt$_i$, yInt$_i$) are derived as follows for i = 0..7:

$$xInt_i = Clip3(0, picW − 1, xInt_L + i − 3) \quad (8\text{-}714)$$

$$yInt_i = Clip3(0, picH − 1, yInt_L + i − 3) \quad (8\text{-}715)$$

The luma locations in full-sample are further modified as follows for i = 0..7:

$$xInt_i = Clip3(xSbInt_L − 3, xSbInt_L + sbWidth + 4, xInt_i)$$

$yInt_i = Clip3(ySbInt_L - 3, ySbInt_L + sbHeight + 4, yInt_i)$

The predicted luma sample value $predSampleLX_L$ is derived as follows:

– If both $xFrac_L$and $yFrac_L$ are equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = refPicLX_L[xInt_3][yInt_3] \quad (8\text{-}716)$$

– Otherwise, if $xFrac_L$ is not equal to 0 and $yFrac_L$ is equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = Clip3(0, (1 << BitDepthY) - 1, (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_3]) >> 6) \quad (8\text{-}717)$$

– Otherwise, if $xFrac_L$ is equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = Clip3(0, (1 << BitDepthY) - 1, (\sum_{i=0}^{7} f_L[yFrac_L][i] * refPicLX_L[xInt_3][yInt_i]) >> 6) \quad (8\text{-}718)$$

– Otherwise, if $xFrac_L$ is not equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

  – The sample array temp[n] with n = 0..7, is derived as follows:

$$temp[n] = (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_n]) >> shift1 \quad (8\text{-}719)$$

  – The predicted luma sample value $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = Clip3(0, (1 << BitDepthY) - 1, (\sum_{i=0}^{7} f_L[yFrac_L][i] * temp[i]) >> shift2) \quad (8\text{-}720)$$

Table 8-9 – Specification of the luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p for sps_amis_flag = = 1

Fractional sample position p	interpolation filter coefficients							
	$f_L[p][0]$	$f_L[p][1]$	$f_L[p][2]$	$f_L[p][3]$	$f_L[p][4]$	$f_L[p][5]$	$f_L[p][6]$	$f_L[p][7]$
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

Table 8-10 – Specification of the luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p for sps_amis_flag = = 0

Fractional sample position p	interpolation filter coefficients							
	$f_L[p][0]$	$f_L[p][1]$	$f_L[p][2]$	$f_L[p][3]$	$f_L[p][4]$	$f_L[p][5]$	$f_L[p][6]$	$f_L[p][7]$
1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
4	0	1	–5	52	20	–5	1	0
5	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
6	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
7	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
8	0	2	–10	40	40	–10	2	0
9	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
10	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
11	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
12	0	1	–5	20	52	–5	1	0
13	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
14	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
15	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

8.5.4.2.3 Chroma sample interpolation process

Inputs to this process are:

– a chroma location in full-sample units ($xInt_C$, $yInt_C$),

– a chroma location in 1/32 fractional-sample units ($xFrac_C$, $yFrac_C$),

– a chroma location in full-sample units (xSbIntC, ySbIntC) specifying the top-left sample of the bounding block for reference sample padding relative to the top-left chroma sample of the reference picture,

– a variable sbWidth specifying the width of the current chroma subblock,

– a variable sbHeight specifying the height of the current chroma subblock,

– the chroma reference sample array $refPicLX_C$.

Output of this process is a predicted chroma sample value $predSampleLX_C$.

The variables shift1, shift2 and offset2 are derived as follows:

– The variable shift1 is set equal to Min(4, $BitDepth_C$ − 8), the variable shift2 is set equal to Max(8, 20 – $BitDepth_C$).

– The variable offset2 is set equal to 1 << (shift2 − 1).

– The variable $picW_C$ is set equal to pic_width_in_luma_samples / SubWidthC and the variable $picH_C$ is set equal to pic_height_in_luma_samples / SubHeightC.

The luma interpolation filter coefficients $f_C[p]$ for each 1/32 fractional sample position p equal to $xFrac_C$ or $yFrac_C$ are specified in Table 8-11.

The chroma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..3:

$$xInt_i = Clip3(0, picW_C − 1, xInt_C + i − 1) \quad (8\text{-}721)$$

$$yInt_i = Clip3(0, picH_C − 1, yInt_C + i − 1) \quad (8\text{-}722)$$

The chroma locations in full-sample units ($xInt_i$, $yInt_i$) are further modified as follows for i = 0..3:

$$xInt_i = Clip3(xSbIntC − 1, xSbIntC + sbWidth + 2, xInt_i)$$

$$yInt_i = Clip3(ySbIntC − 1, ySbIntC + sbHeight + 2, yInt_i)$$

The predicted chroma sample value $predSampleLX_C$ is derived as follows:

– If both $xFrac_C$ and $yFrac_C$ are equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = refPicLX_C[\ xInt_1\ ][\ yInt_1\ ] \quad (8\text{-}723)$$

– Otherwise if $xFrac_C$ is not equal to 0 and $yFrac_C$ is equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = Clip3(\ 0,\ (\ 1\ <<\ BitDepthC)\ -\ 1,\ (\ \textstyle\sum_{i=0}^{3} f_C[\ xFrac_C\ ][\ i\ ]\ *\ refPicLX_C[\ xInt_i\ ][\ yInt_1\ ]\ )\ >>\ 6\ ) \quad (8\text{-}724)$$

– Otherwise if $xFrac_C$ is equal to 0 and $yFrac_C$ is not equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = Clip3(\ 0,\ (\ 1\ <<\ BitDepthC)\ -\ 1,\ (\ \textstyle\sum_{i=0}^{3} f_C[\ yFrac_C\ ][\ i\ ]\ *\ refPicLX_C[\ xInt_1\ ][\ yInt_i\ ]\ )\ >>\ 6\ ) \quad (8\text{-}725)$$

– Otherwise if $xFrac_C$ is not equal to 0 and $yFrac_C$ is not equal to 0, the value of $predSampleLX_C$ is derived as follows:

  – The sample array temp[n] with n = 0..3, is derived as follows:

$$temp[\ n\ ] = (\ \textstyle\sum_{i=0}^{3} f_C[\ xFrac_C\ ][\ i\ ]\ *\ refPicLX_C[\ xInt_i\ ][\ yInt_n\ ]\ )\ >>\ shift1 \quad (8\text{-}726)$$

  – The predicted chroma sample value $predSampleLX_C$ is derived as follows:

$$\begin{aligned} predSampleLX_C = Clip3(\ 0,\ (\ 1\ <<\ BitDepth_C)\ -\ 1,\ (\ offset2\ +\ \\ f_C[\ yFrac_C\ ][\ 0\ ]\ *\ temp[\ 0\ ]\ +\ \\ f_C[\ yFrac_C\ ][\ 1\ ]\ *\ temp[\ 1\ ]\ +\ \\ f_C[\ yFrac_C\ ][\ 2\ ]\ *\ temp[\ 2\ ]\ +\ \\ f_C[\ yFrac_C\ ][\ 3\ ]\ *\ temp[\ 3\ ]\ )\ >>\ shift2\ ) \end{aligned} \quad (8\text{-}727)$$

Table 8-11 – Specification of the chroma interpolation filter coefficients $f_C[\ p\ ]$ for each 1/32 fractional sample position p for sps_amis_flag = = 1

Fractional sample position p	interpolation filter coefficients			
	$f_C[\ p\ ][\ 0\ ]$	$f_C[\ p\ ][\ 1\ ]$	$f_C[\ p\ ][\ 2\ ]$	$f_C[\ p\ ][\ 3\ ]$
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

Table 8-12 – Specification of the chroma interpolation filter coefficients f_C[p] for each 1/32 fractional sample position p for sps_amis_flag = = 0

Fractional sample position p	interpolation filter coefficients			
	f_C[p][0]	f_C[p][1]	f_C[p][2]	f_C[p][3]
1	n/a	n/a	n/a	n/a
2	n/a	n/a	n/a	n/a
3	n/a	n/a	n/a	n/a
4	−2	58	10	−2
5	n/a	n/a	n/a	n/a
6	n/a	n/a	n/a	n/a
7	n/a	n/a	n/a	n/a
8	–4	52	20	–4
9	n/a	n/a	n/a	n/a
10	n/a	n/a	n/a	n/a
11	n/a	n/a	n/a	n/a
12	–6	46	30	–6
13	n/a	n/a	n/a	n/a
14	n/a	n/a	n/a	n/a
15	n/a	n/a	n/a	n/a
16	–8	40	40	–8
17	n/a	n/a	n/a	n/a
18	n/a	n/a	n/a	n/a
19	n/a	n/a	n/a	n/a
20	−4	28	46	−6
21	n/a	n/a	n/a	n/a
22	n/a	n/a	n/a	n/a
23	n/a	n/a	n/a	n/a
24	–4	20	52	–4
25	n/a	n/a	n/a	n/a
26	n/a	n/a	n/a	n/a
27	n/a	n/a	n/a	n/a
28	−2	10	58	−2
29	n/a	n/a	n/a	n/a
30	n/a	n/a	n/a	n/a
31	n/a	n/a	n/a	n/a

8.5.4.3 Interpolation process for the enhanced interpolation filter

Inputs to this process are:

– a location (xCb, yCb) in full-sample units,

– two variables cbWidth and cbHeight specifying the width and the height of the current coding block,

– horizontal change of motion vector dX,

– vertical change of motion vector dY,
– motion vector mvBaseScaled,
– the selected reference picture sample arrays refPicLX,
– sample bit depth bitDepth
– width of the picture in samples pic_width,
– height of the picture in samples pic_height.

Outputs of this process are:

– an (cbWidth)x(cbHeight) array predSamplesLX of prediction sample values.

The variables shift1, shift2, shift3, offset1, offset2 and offset3 are derived as follows:

shift0 is set equal to bitDepth – 6, offset0 is equal to $2^{shift1-1}$,

shift1 is set equal to 11, offset1 is equal to 1024

For x = −1.. cbWidth and y = −1.. cbHeight, the following applies:

– The motion vector mvX is derived as follows :

$$mvX[0] = (mvBaseScaled[0] + dX[0] * x + dY[0] * y) \quad (8\text{-}728)$$

$$mvX[1] = (mvBaseScaled[1] + dX[1] * x + dY[1] * y) \quad (8\text{-}729)$$

– The variables xInt, yInt, xFrac and yFrac are derived as follows:

$$xInt = xCb + (mvX[0] >> 9) + x \quad (8\text{-}730)$$

$$yInt = yCb + (mvX[1] >> 9) + y \quad (8\text{-}731)$$

$$xFrac = mvX[0] \& 511 \quad (8\text{-}732)$$

$$yFrac = mvX[1] \& 511 \quad (8\text{-}733)$$

– The locations (xInt, yInt) inside the given array refPicLX are derived as follows:

$$xInt = Clip3(0, pic\_width - 1, xInt) \quad (8\text{-}734)$$

$$yInt = Clip3(0, pic\_height - 1, yInt) \quad (8\text{-}735)$$

– The variables $a_{x,y}$, $a_{x+1,y}$, $a_{x,y+1}$, $a_{x+1,y+1}$ are derived as follows:

$$a_{x,y} = ((refPicLX[xInt][yInt] * (512 - xFrac) + offset0) >> shift0) * (512 - yFrac) \quad (8\text{-}736)$$

$$a_{x+1,y} = ((refPicLX[xInt + 1][yInt] * xFrac + offset0) >> shift0) * (512 - yFrac) \quad (8\text{-}737)$$

$$a_{x,y+1} = ((refPicLX[xInt][yInt + 1] * (512 - xFrac) + offset0) >> shift0) * yFrac \quad (8\text{-}738)$$

$$a_{x+1,y+1} = (((refPicLX[xInt][yInt] * xFrac + offset0) >> shift0) * yFrac \quad (8\text{-}739)$$

– The sample value $b_{x,y}$ corresponding to location (x, y) is derived as follows:

$$b_{x,y} = (a_{x,y} + a_{x+1,y} + a_{x,y+1} + a_{x+1,y+1} + offset1) >> shift1 \quad (8\text{-}740)$$

The enhancement interpolation filter coefficients eF[] are specified as {−1, 10 , −1}.

The variables shift2, shift3, offset2 and offset3 are derived as follows:

shift2 is set equal to 4, offset2 is equal to 8,

shift3 is set equal to 15– bitDepth, offset3 is equal to $2^{shift3-1}$,

For x = 0.. cbWidth – 1 and y = –1.. cbHeight, the following applies:

– $h_{x,y} = (eF[0] * b_{x-1,y} + eF[1] * b_{x,y} + eF[2] * b_{x+1,y} + offset2) >> shift2$ (8-741)

For x = 0.. cbWidth – 1 and y = 0.. cbHeight – 1, the following applies:

– $predSamplesLX_L[x][y] = Clip3(0, (1 << bitDepth) – 1,$

$(eF[0] * h_{x,y-1} + eF[1] * h_{x,y} + eF[2] * b_{x,y+1} + offset3) >> shift3)$ (8-742)

8.5.4.4 Weighted sample prediction process

Inputs to this process are:

– two variables nCbW and nCbH specifying the width and the height of the current coding block,

– two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,

– the prediction list utilization flags, predFlagL0 and predFlagL1,

– the reference indices refIdxL0 and refIdxL1,

– a variable cIdx specifying colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

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] = predSamplesL0[x][y] (8-743)

– Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

pbSamples[x][y] = predSamplesL1[x][y] (8-744)

– Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

pbSamples[x][y] = (predSamplesL0[x][y] + predSamplesL1[x][y] + 1) >> 1 (8-745)

8.5.5 Decoder-Side Motion vector refinement process

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 1/4 fractional-sample accuracy mvL0 and mvL1,

– the selected reference picture sample arrays refPicL0L and refPicL1L.

Outputs of this process is the delta luma motion vector dMvL0.

For each X(0,1) motion vectors mvLtX are derived as conversion mvLX from ¼ accuracy to 1/16 accuracy in the clause 8.5.2.17.

For each X(0,1) motion vectors mvLsX are derived as follows:

– mvLsX[0] = mvLtX[0] − 32, mvLsX[1] = mvLtX[1] − 32.

A variable srRange is set as 2. Variables offsetH[0], offsetV[0], offsetH[1] and offsetV[1] are set equal to 2.

For each X (0,1) the (cbWidth+2*srRange)x(cbHeigh+2*srRange) array $predSamplesLX_L$ of prediction luma sample values is derived by invoking the process specified in 8.5.5.3 with luma location (xCb, yCb), width of subblock

cbWidth+2*srRange, height of subblock cbHeight+2*srRange, reference picture sample array refPicX_L and motion vector mvLsX as inputs and predSamplesLX_L as output.

The 9x1 array of sad1 and the variable centerSad are derived by invoking the process 8.5.5.2 with cbWidth, cbHeight, offsetH, offsetV, predSamplesL0_L and predSamplesL1_L as inputs and sad1 and centerSad as outputs.

Bitdepth is set equal to $BitDepth_Y$. If centerSad greater than or equal to $2^{bitdepth-9}$ then the following ordered steps are applied:

– Array entry selection process specified in 8.5.5.3 is invoked with sad1 and n=9 as inputs and bestIdx as output.

If bestIdx is equal to 4, halfPelAppliedflag is set equal to TRUE. Otherwise the following ordered steps are applied:

– dMvL0[0] is ser equal to 16*(bestIdx/3 – 1), dMvL0[1] is set equal to 16*(bestIdx%3 – 1),

– offsetH[0] = offsetH[0] + (bestIdx/3−1), offsetV[0] = offsetV[0] + (bestIdx%3−1), offsetH[1] = offsetH[1] + (1 − bestIdx/3), offsetV[1] = offsetV[1] + (1 − bestIdx%3),

– The 9x1 array of sad2 is derived by invoking the process 8.5.5.2 with cbWidth, cbHeight, offsetH, offsetV, predSamplesL0_L and predSamplesL1_L as inputs and sad2 as outputs.

– Array entry selection process specified in 8.5.5.3 is invoked with sad2 and n=9 as inputs and bestIdx as output.

If bestIdx is not equal to 4, then the following ordered steps are applied;

– dMvL0 is modified as dMvL0[0] = dMvL0[0] + 16*(bestIdx/3 – 1) and dMvL0[1] = dMvL0[1] + 16*(bestIdx%3 – 1),

If halfPelAppliedflag is true following ordered steps are applied:

– parametric motion vector refinement process which is specified in 8.5.5.4 is invoked with sad1, dMvL0 as inputs and modified dMvL0 as outputs.

8.5.5.1 Fractional sample bilinear interpolation process

8.5.5.1.1 General

Inputs to this process are:

– a luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma,sample of the current picture,

– a variable sbWidth specifying the width of the current coding subblock in luma samples,

– a variable sbHeight specifying the height of the current coding subblock in luma samples,

– a luma motion vector mvLX given in 1/16-luma-sample units,

– the selected reference picture sample array refPicLXL.

Outputs of this process are:

– an (sbWidth)x(sbHeight) array predSamplesLX_L of prediction luma sample values.

Let (xIntL, yIntL) be a luma location given in full-sample units and (xFracL, yFracL) be an offset given in 1/16-sample units. These variables are used only in this clause for specifying fractional-sample locations inside the reference sample arrays refPicLXL, refPicLXCb and refPicLXCr.

For each luma sample location (xL = 0..sbWidth − 1, yL = 0..sbHeight − 1) inside the prediction luma sample array predSamplesLXL, the corresponding prediction luma sample value predSamplesLXL[xL][yL] is derived as follows:

– The variables $xInt_L$, $yInt_L$, $xFrac_L$ and $yFrac_L$ are derived as follows:

$xInt_L = xSb + (mvLX[0] >> 4) + x_L$

$yInt_L = ySb + (mvLX[1] >> 4) + y_L$

$xFrac_L = mvLX[0] \& 15$

$yFrac_L = mvLX[1] \& 15$

The prediction luma sample value $predSamplesLX_L[x_L][y_L]$ is derived by invoking the process specified in clause 8.5.5.1.2 with ($xInt_L$, $yInt_L$), ($xFrac_L$, $yFrac_L$) and $refPicLX_L$ as inputs.

8.5.5.1.2 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 $predSampleLX_L$.

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min(4, $BitDepth_Y$ − 8), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max(2, 14 − $BitDepth_Y$).

– 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 $fb_L[p]$ for each 1/16 fractional sample position p equal to $xFrac_L$ or $yFrac_L$ are specified in Table 8-13.

The predicted luma sample value $predSampleLX_L$ is derived as follows:

– If both $xFrac_L$and $yFrac_L$ are equal to 0, the value of $predSampleLX_L$ is derived as follows:

$predSampleLX_L = refPicLX_L[xInt_L][yInt_L] << shift3$

– Otherwise if $xFrac_L$ is not equal to 0 and $yFrac_L$ is equal to 0, the value of $predSampleLX_L$ is derived as follows:

$predSampleLX_L = (fb_L[xFrac_L][0] * refPicLX_L[Clip3(0, picW − 1, xInt_L)][yInt_L] +$
$fb_L[xFrac_L][1] * refPicLX_L[Clip3(0, picW − 1, xInt_L + 1)][yInt_L]) >> shift1$

– Otherwise if $xFrac_L$ is equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

$predSampleLX_L = (fb_L[yFrac_L][0] * refPicLX_L[xInt_L][Clip3(0, picH − 1, yInt_L)] +$
$fb_L[yFrac_L][1] * refPicLX_L[xInt_L][Clip3(0, picH − 1, yInt_L + 1)]) >> shift1$

– Otherwise if $xFrac_L$ is not equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

– The sample array temp[n] with n = 0..7, is derived as follows:

$yPos_L = Clip3(0, PicH − 1, yInt_L + n − 3)$

$temp[n] = (fb_L[xFrac_L][0] * refPicLX_L[Clip3(0, picW − 1, xInt_L)][yPos_L] +$
$fb_L[xFrac_L][1] * refPicLX_L[Clip3(0, picW − 1, xInt_L + 1)][yPos_L]) >> shift1$

– The predicted luma sample value $predSampleLX_L$ is derived as follows:

$predSampleLX_L = (fb_L[yFrac_L][0] * temp[0] +$
$fb_L[yFrac_L][1] * temp[1]) >> shift2$

Table 8-13 Specification of the luma interpolation filter coefficients fbL[p] for each 1/16 fractional sample position p

Fractional sample position p	interpolation filter coefficients	
	$fb_L[p][0]$	$fb_L[p][1]$
1	60	4
2	56	8
3	52	12
4	48	16
5	44	20
6	40	24
7	36	28
8	32	32
9	28	36
10	24	40
11	20	44
12	16	48
13	12	52
14	8	56
15	4	60

8.5.5.2 Sum of absolute differences calculation 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) array predSamplesL0 and predSamplesL1,

– four variables offsetH0, offsetH1, offsetV0, offsetV1.

Outputs of this process are

– the (9) array sadValues of the sum of absolute differences,

– the variable centerPointSad.

A 2x9 array bC is set as follows;

– bC[0][0] = −1, bC [1][0] = −1, bC [0][1] = −1, bC [1][1] = 0, bC [0][2] = −1, bC [1][2] = 1,

– bC[0][3] = 0, bC [1][3] = −1, bC [0][4] = 0, bC [1][4] = 0, bC [0][5] = 0, bC [1][5] = 1,

– bC[0][6] = 1, bC [1][6] = −1, bC [0][7] = 1, bC [1][7] = 0, bC [0][8] = 1, bC [1][8] = 1

For each i (0, …, 8), sadValues [i] is derived based on predSampleL0[x][y], predSampleL1[x][y], bC[0][i], bC[1][i], offsetH0, offsetV0, offsetH1, and offsetV1 as follows:

– Set initial value of mrSadValues [i] to 0,

– The SAD value sadValues [i] is accumulated for y=bC[1][i]..nCbH+bC[1][i]:

  – for each x=bC[0][i]..nCbW+bC[0][i]:

$$\text{mrSadValues[i]} = \text{mrSadValues[i]} + \text{abs}(\text{predSamplesL0}[x + \text{offsetH0}][\text{y} + \text{offsetV0}] - \text{predSamplesL1}[x - 2 \times \text{bC}[0][\text{i}] + \text{offsetH1}][\text{y} - 2 \times \text{bC}[1][\text{i}] + \text{offsetV1}])$$

The variable centerPointSad is derived based on predSampleL0[x][y] and predSampleL1[x][y] as follows:

– Set initial value of centerPointSad to 0,

– Value of centerPointSad is accumulated for x=0..nCbW and y=0..nCbH:

$$\text{centerPointSad} = \text{centerPointSad} + \text{abs}(\text{predSamplesL0}[x][\text{y}] - \text{predSamplesL1}[x][\text{y}])$$

8.5.5.3 Array entry selection process

Inputs to this process are:

– A positive integer variable n specifying the length of the array,

– An nx1 array mrSad.

Output of this process is bestIdx specifying an index.

If n is equal to 9 the following ordered steps are applied:

– If sad[1]< sad[7] and sad[3]< sad[5], a variable idx is set equal to 0

  o If sad[1]< sad[3], a variable bestIdx is set to 1, otherwise it is set to 3.

– Else if sad[1]>= sad[7] and sad[3]< sad[5], idx is set equal to 6

  o If sad[7]< sad[3], bestIdx is set to 7, otherwise it is set to 3.

– Else if sad[1]< sad[7] and sad[3]>= sad[5], idx is set equal to 2

  o If sad[1]< sad[5], bestIdx is set to 1, otherwise it is set to 5.

– Else if sad[1]>= sad[7] and sad[3]>= sad[5], idx is set equal to 8

  o If sad[7]< sad[5], bestIdx is set to 7, otherwise it is set to 5.

– If sad[4] < sad[bestIdx], bestIdx is set equal to 4.

– If sad[bestIdx] < sad[idx], bestIdx is set equal to idx.

8.5.5.4 Parametric motion vector refinement process

Inputs to this process are:

– Evaluated cost values in SAD array (where index 4 corresponds to the bestIdx, indices 1 and 7 correspond to the integer distance positions to the left and right of the bestIdx position, and indices 3 and 5 correspond to the integer distance positions to the top and bottom of the bestIdx position)

– Luma delta motion vector dMvL0 given in 1/16-luma-sample units,

Output of this process is the modified luma motion vector dMvL0.

The following ordered steps are performed to derive the modified mvL0 and mvL1:

– Compute delta motion vector dmv as follows:

If (sad[1] + sad[7] = = (sad[4]<<1)), dmv[0] = 0;

Else, dmv[0] = ((sad[1] – sad[7])<<4)/(sad[1] + sad[7] – (sad[4]<<1))

If (sad[3] + sad[5] = = (sad[4]<<1)), dmv[1] =0;

Else, dmv[1] = ((sad[3] – sad[5])<<4)/(sad[3] + sad[5] – (sad[4]<<1))

– Using these, dMvL0 is modified as follows:

dMvL0[0] = dMvL0[0] + dmv[0], dMvL0[1] = dMvL0[1] + dmv[1],

NOTE: dmv[0] and dmv[1] are constrained to be between −8 and 8, irrespective of the bit-depth at which mrSad array values are represented. This allows the division to be performed till 4 quotient bits become available and can be implemented using compares, shifts, and subtractions.

8.5.6 Decoding process for the residual signal of 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 luma coding block relative to the top-left luma sample of the current picture,

– two variables log2CbWidth and log2CbHeight specifying the width and the height of the current luma coding block.

Outputs of this process are:

– an ($nCbW_L$)x($nCbH_L$) array $resSamples_L$ of luma residual samples, where $nCbW_L$ and $nCbH_L$ are derived as specified below,

– when ChromaArrayType is not equal to 0, an ($nCbW_C$)x($nCbH_C$) array $resSamples_{Cb}$ of chroma residual samples for the component Cb, where $nCbW_C$ and $nCbH_C$ are derived as specified below,

– when ChromaArrayType is not equal to 0, an ($nCbW_C$)x($nCbH_C$) array $resSamples_{Cr}$ of chroma residual samples for the component Cr, where $nCbW_C$ and $nCbH_C$ are derived as specified below.

The variables $nCbW_L$ and $nCbH_L$ are set equal to 1 << log2CbWidth and 1 << log2CbHeight. When ChromaArrayType is not equal to 0, the variable $nCbW_C$ is set equal to $nCbW_L$ / SubWidthC and the variable $nCbH_C$ is set equal to $nCbH_L$ / SubHeightC.

Let $resSamples_L$ be an ($nCbW_L$)x($nCbH_L$) array of luma residual samples and, when ChromaArrayType is not equal to 0, let $resSamples_{Cb}$ and $resSamples_{Cr}$ be two ($nCbW_C$)x($nCbH_C$) arrays of chroma residual samples.

Depending on the value of cbf_all, the following applies:

– If cbf_all is equal to 0 or cu_skip_flag[xCb][yCb] is equal to 1, all samples of the ($nCbW_L$)x($nCbH_L$) array $resSamples_L$ and, when ChromaArrayType is not equal to 0, all samples of the two ($nCbW_C$)x($nCbH_C$) arrays $resSamples_{Cb}$ and $resSamples_{Cr}$ are set equal to 0.

– Otherwise (cbf_all is equal to 1), the following ordered steps apply:

1. The decoding process for luma residual blocks as specified in clause 8.5.6.1 below is invoked with the luma location (xCb, yCb), the variables log2TrafoWidth and log2TrafoHeight set equal to log2CbWidth and log2CbHeight, the variables nCbW and nCbH set equal to $nCbW_L$ and $nCbH_L$, and the $(nCbW_L)x(nCbH_L)$ array $resSamples_L$ as inputs, and the output is a modified version of the $(nCbW_L)x(nCbH_L)$ array $resSamples_L$.
2. When ChromaArrayType is not equal to 0, the decoding process for chroma residual blocks as specified in clause 8.5.6.2 is invoked with the luma location (xCb, yCb), the variables log2TrafoWidth and log2TrafoHeight set equal to log2CbWidth and log2CbHeight, the variable cIdx set equal to 1, the variable nCbW set equal to $nCbW_C$, the variable nCbH set equal to $nCbH_C$ and the $(nCbW_C)x(nCbH_C)$ array $resSamples_{Cb}$ as inputs, and the output is a modified version of the $(nCbW_C)x(nCbH_C)$ array $resSamples_{Cb}$.
3. When ChromaArrayType is not equal to 0, the decoding process for chroma residual blocks as specified in clause 8.5.6.2 is invoked with the luma location (xCb, yCb), the variables log2TrafoWidth and log2TrafoHeight set equal to log2CbWidth and log3CbHeight, the variable cIdx set equal to 2, the variable nCbW set equal to $nCbW_C$, the variable nCbH set equal to $nCbH_C$ and the $(nCbW_C)x(nCbH_C)$ array $resSamples_{Cr}$ as inputs, and the output is a modified version of the $(nCbW_C)x(nCbH_C)$ array $resSamples_{Cr}$.

8.5.6.1 Decoding process for luma residual blocks

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 log2TrafoWidth and log2TrafoHeight specifying the width and the height of the current luma block,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

– an (nCbW)x(nCbH) array resSamples of luma residual samples.

Output of this process is a modified version of the (nCbW)x(nCbH) array of luma residual samples.

The following ordered steps apply:

1. The variables nTbW and nTbH are set equal to 1 << log2TrafoWidth and 1 << log2TrafoHeight, respectively.
2. The scaling and transformation process as specified in clause 8.7.2 is invoked with the luma location (xCb , yCb), the variable cIdx set equal to 0, the transform width trafoWidth set equal to nTbW and the transform height trafoHeight set equal to nTbH as inputs, and the output is an (nTbW)x(nTbH) array transformBlock.
3. The (nCbW)x(nCbH) residual sample array of the current coding block resSamples is modified as follows:

$$resSamples[i, j] = transformBlock[i, j], \text{ with } i = 0..nTbW - 1, j = 0..nTbH - 1 \quad (8\text{-}746)$$

8.5.6.2 Decoding process for chroma residual blocks

This process is only invoked when ChromaArrayType is not equal to 0.

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 log2TrafoWidth and log2TrafoHeight specifying the width and the height of the current chroma block in luma samples,

– a variable cIdx specifying the chroma component of the current block,

– the variables nCbW and nCbH specifying the width and height, respectively, of the current chroma coding block,

– an (nCbW)x(nCbH) array resSamples of chroma residual samples.

Output of this process is a modified version of the (nCbW)x(nCbH) array of chroma residual samples.

The following ordered steps apply:

1. The variable nTbW and nTbH are set equal to (1 << log2TrafoWidth) / SubWidthC and (1 << log2TrafoHeight) / SubHeightC.

2. The scaling and transformation process as specified in clause 8.7.2 is invoked with the luma location (xCb , yCb), the variable cIdx, the transform width trafoWidth set equal to nTbW and the transform width trafoHeight set equal to nTbH as inputs, and the output is an (nTbW)x(nTbH) array transformBlock.

3. The (nCbW)x(nCbH) residual sample array of the current coding block resSamples is modified as follows, for i = 0..nTbW − 1, j = 0..nTbH − 1:

   resSamples[(xCb) / SubWidthC + i, (yCb) / SubHeightC + j] = transformBlock[i, j] (8-747)

8.6 Decoding process for coding units coded in ibc prediction mode

8.6.1 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 luma coding block relative to the top-left luma sample of the current picture,

– two variables log2CbWidth and log2CbHeight specifying the width and the height of the current luma coding block.

Output of this process is a modified reconstructed block before post-reconstruction filtering.

The derivation process for quantization parameters as specified in clause 8.7.1 is invoked with the luma location (xCb, yCb) as input.

The variables nCbW and nCbH are set equal to 1 << log2CbWidth and 1 << log2CbHeight, respectively. The variables $nCbW_L$ and $nCbH_L$ are set equal to 1 << log2CbWidth and 1 << log2CbHeight, repectively. When ChromaArrayType is not equal to 0, the variable $nCbW_C$ is set equal to (1 << log2CbWidth) / SubWidthC and the variable $nCbH_C$ is set equal to (1 << log2CbHeight) / SubHeightC.

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

1. The derivation process for motion vector components as specified in clause 8.6.2 is invoked with the luma coding block location (xCb, yCb), the luma coding block width nCbW, and the luma coding block height nCbH as inputs, and the luma motion vectors mvL, when ChromaArrayType is not equal to 0, the chroma motion vector mvC as outputs.

2. The decoding process for inter sample prediction as specified in clause 8.6.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width nCbW, the luma coding block height nCbH, the luma motion vector mvL, when ChromaArrayType is not equal to 0, the chroma motion vector mvC as inputs, and the ibc prediction samples (predSamples) that are an ($nCbW_L$)x($nCbH_L$) array $predSamples_L$ of prediction luma samples and, when ChromaArrayType is not equal to 0, two ($nCbW_C$)x($nCbH_C$) arrays $predSamples_{Cr}$ and $predSamples_{Cr}$ of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

3. The decoding process for the residual signal of coding units coded in inter prediction mode specified in clause 8.5.6 is invoked with the luma location (xCb, yCb) and the luma coding block width log2CbWidth and the luma coding block height log2CbHeight as inputs, and the outputs are three arrays resSamplesL, resSamplesCb, and resSamplesCr.

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

   – The picture reconstruction process prior to in-loop filtering for a colour component as specified in clause 8.7.5 is invoked with the luma coding block location (xCb, yCb), the variable nCurrW set equal to $nCbW_L$, the variable nCurrH set equal to $nCbH_L$, the variable cIdx set equal to 0, the ($nCbW_L$)x($nCbH_L$) array predSamples set equal to $predSamples_L$, and the ($nCbW_L$)x($nCbH_L$) array resSamples set equal to $resSamples_L$ as inputs.

   – When value of cbf_luma is equalt to 1and sps_htdf_flag is equal to 1, the post-reconstruction filtering process prior to in-loop filtering for a luma component as specified in clause 8.7.6 is invoked with the luma coding block location (xCb, yCb), the variable nCurrW set equal to $nCbW_L$, the variable nCurrH set equal to $nCbH_L$, and the ($nCbW_L$)x($nCbH_L$) array recSamples set equal to output of picture reconstruction process $recSamples_L$ as inputs.

   – When ChromaArrayType is not equal to 0, the picture reconstruction process prior to in-loop filtering for a colour component as specified in clause 8.7.5 is invoked with the chroma coding block location (xCb / SubWidthC, yCb / SubHeightC), the variable nCurrW set equal to $nCbW_C$, the variable nCurrH set

equal to $nCbH_C$, the variable cIdx set equal to 1, the $(nCbW_C)x(nCbH_C)$ array predSamples set equal to $predSamples_{Cb}$, and the $(nCbW_C)x(nCbH_C)$ array resSamples set equal to $resSamples_{Cb}$ as inputs.

– When ChromaArrayType is not equal to 0, the picture reconstruction process prior to in-loop filtering for a colour component as specified in clause 8.7.5 is invoked with the chroma coding block location (xCb / SubWidthC, yCb / SubHeightC), the variable nCurrW set equal to $nCbW_C$, the variable nCurrH set equal to $nCbH_C$, the variable cIdx set equal to 2, the $(nCbW_C)x(nCbH_C)$ array predSamples set equal to $predSamples_{Cr}$, and the $(nCbW_C)x(nCbH_C)$ array resSamples set equal to $resSamples_{Cr}$ as inputs.

5. The following assignments are made for x = 0..nCbW − 1 and y = 0..nCbH − 1:

$$MvL0[xCb + x][yCb + y] = mvL \quad (8\text{-}748)$$

8.6.2 Derivation process for motion vector components

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 nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs of this process are:

– the luma motion vectors mvL,

– when ChromaArrayType is not equal to 0, the chroma motion vectors mvC.

The luma motion vector mvL is derived as follows:

1. The variable mvd is derived as follows:

$$mvd[0] = MvdL0[xCb][yCb][0] \quad (8\text{-}749)$$

$$mvd[1] = MvdL0[xCb][yCb][1] \quad (8\text{-}750)$$

2. The variables mvp[0] and mvp[1] are set equal to 0.

3. The luma motion vector mvL is derived as follows:

$$u[0] = (mvp[0] + mvd[0] + 2^{16}) \% 2^{16} \quad (8\text{-}751)$$

$$mvL[0] = (u[0] >= 2^{15}) ? (u[0] - 2^{16}) : u[0] \quad (8\text{-}752)$$

$$u[1] = (mvp[1] + mvd[1] + 2^{16}) \% 2^{16} \quad (8\text{-}753)$$

$$mvL[1] = (u[1] >= 2^{15}) ? (u[1] - 2^{16}) : u[1] \quad (8\text{-}754)$$

The top-left location inside the reference block (xRefTL, yRefTL), the top-right location inside the reference block (xRefTR, yRefTR), the bottom-left location inside the reference block (xRefBL, yRefBL), and the bottom-right location inside the reference block (xRefBR, yRefBR) are derived as follows:

$$(xRefTL, yRefTL) = (xCb + (mvL[0]), yCb + (mvL[1])) \quad (8\text{-}755)$$

$$(xRefTR, yRefTR) = (xRefTL + cbWidth - 1, yRefTL) \quad (8\text{-}756)$$

$$(xRefBL, yRefBL) = (xRefTL, yRefTL + cbHeight - 1) \quad (8\text{-}757)$$

$$(xRefBR, yRefBR) = (xRefTL + cbWidth - 1, yRefTL + cbHeight - 1) \quad (8\text{-}758)$$

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 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xRefTL, yRefTL) as inputs, and the output shall be equal to TRUE.

– When the derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xRefBR, yRefBR) as inputs, and the output shall be equal to TRUE.

– When sps_suco_flag is equal to 1, the derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xRefBL, yRefBL) as inputs, and the output shall be equal to TRUE

– When sps_suco_flag is equal to 1, the derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xRefTL + width / 2, yRefTL) as inputs, and the output shall be equal to TRUE

– At least one of the following conditions shall be true:

  – The value of mvL[0] + cbWidth is less than or equal to 0.

  – The value of mvL[1] + cbHeight is less than or equal to 0.

  – When sps_suco_flag is equal to 1, the value of mvL[0] is greater than or equal to cbWidth.

– The following conditions shall be true:

$$yRefTL >> CtbLog2SizeY = yCb >> CtbLog2SizeY \quad (8\text{-}759)$$

$$yRefBL >> CtbLog2SizeY = yCb >> CtbLog2SizeY \quad (8\text{-}760)$$

$$xRefTL >> CtbLog2SizeY >= (xCb >> CtbLog2SizeY) - 1 \quad (8\text{-}761)$$

$$xRefTR >> CtbLog2SizeY <= (xCb >> CtbLog2SizeY) \quad (8\text{-}762)$$

– When yRefTL >> CtbLog2SizeY is equal to (xCb >> CtbLog2SizeY) − 1, and CtbLog2SizeY is equal to 7, the follow conditions shall be true:

  – The derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location(xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ((xRefTL + 128) / 64 * 64, yRefTL / 64 * 64) as inputs, and the output shall be equal to FALSE.

  – The luma location ((xRefTL + 128) / 64 * 64, yRefTL / 64 * 64) shall not be equal to (xCb, yCb).

– When sps_suco_flag is equal to 1 and yRefTL >> CtbLog2SizeY is equal to (xCb >> CtbLog2SizeY) − 1, and CtbLog2SizeY is equal to 7, the follow conditions shall be true:

  – The derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location(xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ((xRefTL + 128) / 64 * 64 + 63, yRefTL / 64 * 64) as inputs, and the output shall be equal to FALSE.

  – The luma location ((xRefTL + 128) / 64 * 64 + 63, yRefTL / 64 * 64) shall not be equal to (xCb + cbWidth − 1, yCb).

  – The derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location(xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ((xRefTR + 128) / 64 * 64, yRefTR / 64 * 64) as inputs, and the output shall be equal to FALSE.

  – The luma location ((xRefTR + 128) / 64 * 64, yRefTR / 64 * 64) shall not be equal to (xCb, yCb).

  – The derivation process for block availability as specified in clause 6.4.1 is invoked with the current luma location(xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ((xRefTR + 128) / 64 * 64 + 63, yRefTR / 64 * 64) as inputs, and the output shall be equal to FALSE.

  – The luma location ((xRefTR + 128) / 64 * 64 + 63, yRefTR / 64 * 64) shall not be equal to (xCb + cbWidth − 1, yCb).

The following is applied:

$$mvL = mvL << 4 \quad (8\text{-}763)$$

The derivation process for chroma motion vectors in clause 8.6.2.1 is invoked with luma motion vector mvL as input, and chroma motion vector mvC as output.

8.6.2.1 Derivation process for chroma motion vector

Input to this process is a luma motion vector mvL.

Output of this process is a chroma motion vector mvC.

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvC, the following applies:

$$mvC[0] = ((mvL[0] >> (3 + SubWidthC)) * 32 \quad (8\text{-}764)$$

$$mvC[1] = ((mvL[1] >> (3 + SubHeightC)) * 32 \quad (8\text{-}765)$$

8.6.3 Decoding process for ibc blocks

8.6.3.1 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,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

– the luma motion vector in 1/16 fractional-sample accuracy mvL,

– the chroma motion vectors in 1/32 fractional-sample accuracy mvC.

Outputs of this process are:

– an ($nCbW_L$)x($nCbH_L$) array $predSamples_L$ of luma prediction samples, where $nCbW_L$ and $nCbH_L$ are derived as specified below,

– when ChromaArrayType is not equal to 0, an ($nCbW_C$)x($nCbH_C$) array $preSamples_{Cb}$ of chroma prediction samples for the component Cb, where $nCbW_C$ and $nCbH_C$ are derived as specified below,

– when ChromaArrayType is not equal to 0, an ($nCbW_C$)x($nCbH_C$) array $predSamples_{Cr}$ of chroma residual samples for the component Cr, where $nCbW_C$ and $nCbH_C$ are derived as specified below.

The variables $nCbW_L$ and $nCbH_L$ are set equal to nCbW and nCbH, respectively, and the variables $nCbW_C$ and $nCbH_C$ are set equal to nCbW / SubWidthC and nCbH / SubHeightC, respectively.

Let $predSamples_L$ be (nCbW)x(nCbH) array of predicted luma sample values and, when ChromaArrayType is not equal to 0, $predSamples_{Cb}$ and $predSamples_{Cr}$ be (nCbW / SubWidthC)x(nCbH / SubHeightC) arrays of predicted chroma sample values.

The current decoded picture consists of a pic_width_in_luma_samples by pic_height_in_luma_samples array of luma samples $currPic_L$ and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples $currPic_{Cb}$ and $currPic_{Cr}$. The current decoded picture sample arrays $currPic_L$, $currPic_{Cb}$ and $currPic_{Cr}$ correspond to decoded sample arrays S_L, S_{Cb} and S_{Cr} derived in clause 8.8 for the current decoded picture.

The arrays $predSamples_L$ and, when ChromaArrayType is not equal to 0, $predSamples_{Cb}$, and $predSamples_{Cr}$ are derived by invoking the fractional sample interpolation process specified in clause 8.5.4.2 with the luma locations (xCb, yCb), the luma coding block width sbWidth, the luma coding block height sbHeight, the motion vectors mvL[xCb][yCb] and, when ChromaArrayType is not equal to 0, mvC[xCb][yCb], and the reference arrays $currPic_L$, $currPic_{Cb}$, and $currPic_{Cr}$ as inputs.

8.7 Scaling, transformation and array construction process prior to deblocking filter process

8.7.1 Derivation process for quantization parameters

Input to this process is 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.

In this process, the variable Qp_Y, the luma quantization parameter Qp'_Y, and the chroma quantization parameters Qp'_{Cb} and Qp'_{Cr} are derived.

The variable Qp_Y is derived as follows:

$$Qp_Y = (Qp_{Y\_PREV} + cu\_qp\_delta + 52) \% 52 \quad (8\text{-}766)$$

where $Qp_{Y,PREV}$ is the luma quantisation parameter, Qp_Y, of the previous coding unit in decoding order in the current tile group. For the first coding unit in the slice, $Qp_{Y,PREV}$ is initially set equal to slice_qp at the start of each slice.

The luma quantization parameter Qp'_Y is derived as follows:

$$Qp'_Y = Qp_Y + QpBdOffset_Y \quad (8\text{-}767)$$

When ChromaArrayType is not equal to 0, the following applies:

– The variables qP_{Cb} and qP_{Cr} are dervied depending on the value of sps_iqt_flag as follows:

$$qPi_{Cb} = Clip3(-QpBdOffset_C, 57, Qp_Y + slice\_cb\_qp\_offset) \quad (8\text{-}768)$$

$$qPi_{Cr} = Clip3(-QpBdOffset_C, 57, Qp_Y + slice\_cr\_qp\_offset) \quad (8\text{-}769)$$

  – If sps_iqt_flag is equal to 0, the variables qP_{Cb} and qP_{Cr} are set equal to the value of Qp_C as specified in Table 8-14 based on the index qPi equal to qPi_{Cb} and qPi_{Cr}, respectively, and qPi_{Cb} and qPi_{Cr} are derived as follows:.

  – Otherwise, if sps_iqt_flag is equal to 1, the variables qP_{Cb} and qP_{Cr} are set equal to the value of Qp_C as specified in Table 8-15 based on the index qPi equal to qPi_{Cb} and qPi_{Cr}, respectively, and qPi_{Cb} and qPi_{Cr} are derived as follows:

– The chroma quantization parameters for the Cb and Cr components, Qp'_{Cb} and Qp'_{Cr}, are derived as follows:

$$Qp'_{Cb} = qP_{Cb} + QpBdOffset_C \quad (8\text{-}770)$$

$$Qp'_{Cr} = qP_{Cr} + QpBdOffset_C \quad (8\text{-}771)$$

Table 8-14 – Specification of Qp_C as a function of qPi (sps_iqt_flag = = 0)

qPi	< 30	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Qp_C	= qPi	29	29	29	30	31	32	32	33	33	34	34	35	35	36
qPi	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58
Qp_C	36	36	37	37	37	38	38	39	39	40	40	40	41	41	41

Table 8-15 – Specification of Qp_C as a function of qPi (sps_iqt_flag = = 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	34	35	36	37	37	38	39	40	40	= qPi − 3

8.7.2 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 width of the current block,

– a variable nTbH specifying the height of the current block.

Output of this process is the (nTbW)x(nTbH) array of residual samples r with elements r[x][y].

The quantization parameter qP is derived as follows:

– If cIdx is equal to 0, the following applies.

$$qP = Qp'_Y \quad (8\text{-}772)$$

– Otherwise, if cIdx is equal to 1, the following applies.

$$qP = Qp'_{Cb} \quad (8\text{-}773)$$

– Otherwise (cIdx is equal to 2), the following applies.

$$qP = Qp'_{Cr} \quad (8\text{-}774)$$

The (nTbW)x(nTbH) array of residual samples r is derived as follows:

1. The scaling process for transform coefficients as specified in subclause 8.7.3 is invoked with the transform bloc k location (xTbY, yTbY), the transform block width nTbW and the transform block height nTbH, the colour c omponent variable cIdx, and the quantization parameter qP as inputs, and the output is an (nTbW)x(nTbH) arra y of scaled transform coefficients d.
2. The (nTbW)x(nTbH) array of residual samples r is derived as follows:
   – the transformation process for scaled transform coefficients as specified in subclause 8.7.4 is invoked with the transform block location (xTbY, yTbY), the transform block width nTbW and the transform block h eight nTbH, the colour component variable cIdx, and the (nTbW)x(nTbH) array of scaled transform coeffi cients d as inputs, and the output is an (nTbW)x(nTbH) array of residual samples r.
3. The variable bdShift is derived as follows:

$$bdShift = (cIdx == 0) ? 20 - BitDepthY : 20 - BitDepthC \quad (8\text{-}775)$$

4. The residual sample values r[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are modified as follows:

$$r[x][y] = (r[x][y] + (1 << (bdShift - 1))) >> bdShift \quad (8\text{-}776)$$

8.7.3 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 width of the current block,

– a variable nTbH specifying the height of the current block,

– a variable cIdx specifying the colour component of the current block,

– a variable qP specifying the quantization parameter.

Output of this process is the (nTbW)x(nTbH) array d of scaled transform coefficients with elements d[x][y].

The variable bdShift is derived as follows:

– If cIdx is equal to 0, the following applies.

$$bdShift = BitDepth_Y + (((Log2(nTbW) + Log2(nTbH)) \& 1) * 8 + (Log2(nTbW) + Log2(nTbH)) / 2) - 5 \quad (8\text{-}777)$$

– Otherwise, the following applies.

$$bdShift = BitDepth_C + (((Log2(nTbW) + Log2(nTbH)) \& 1) * 8 + (Log2(nTbW) + Log2(nTbH)) / 2) - 5 \quad (8\text{-}778)$$

The variable rectNorm is derived as follows:

$$rectNorm = ((Log2(nTbW) + Log2(nTbH)) \& 1) == 1 ? 181 : 1 \quad (8\text{-}779)$$

The list levelScale[ ] is specified as follows:

– If sps_iqt_flag is equal to 0, levelScale[k] = { 40, 45, 51, 57, 64, 71 } with k = 0..5.

– Otherwise (if sps_iqt_flag is equal to 1), 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 scaled transform coefficient d[x][y] is derived as follows:

d[x][y] = Clip3(−32768, 32767, ((TransCoeffLevel[xTbY][yTbY][cIdx][x][y] *
levelScale[qP%6] << (qP / 6)) * rectNorm + (1 << (bdShift − 1))) >> bdShift) (8-780)

8.7.4 Transformation process for scaled transform coefficients

8.7.4.1 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 block,

– a variable nTbH specifying the height of the current block,

– a variable cIdx specifying the colour component of the current block,

– an (nTbW)x(nTbH) array d of scaled transform coefficients with elements d[x][y].

Output of this process is the (nTbW)x(nTbH) array r of residual samples with elements r[x][y].

The (nTbW)x(nTbH) array r of residual samples is derived as follows:

1. Each (vertical) column of scaled transform coefficients d[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 is transformed to e[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in subclause 8.7.4.2 for each column x = 0..nTbW − 1 with the size of the transform block nTbH, the list d[x][y] with y = 0..nTbH − 1, and the transform type variable trType set equal to trTypeVer if cIdx is equal 0, otherwise 0 as inputs and the output is the list e[x][y] with y = 0..nTbH − 1.

2. If sps_iqt_flag is equal to 1, the following applies:

   – g[x][y] = Clip3(−32768, 32767, (e[x][y] + 64) >> 7) (8-781)

3. Otherwise, (if sps_iqt_flag is equal to 0), the following applies:

   – g[x][y] = e[x]][y] (8-782)

4. Each (horizontal) row of the resulting array g[x][y] with x = 0..nTbW − 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 subclause 8.7.4.2 for each row y = 0..nTbH − 1 with the size of the transform block nTbW, the list g[x][y] with x = 0..nTbW − 1, and transform type variable trType set equal to trTypeHor if cIdx is equal 0, otherwise 0 as inputs and the output is the list r[x][y] with x = 0..nTbW − 1.

Table 8-16 – Specification of trTypeHor and trTypeVer depending on ats_cu_intra_flag[x][y], ats_hor_mode[x][y], and ats_ver_mode[x][y]

ats_cu_intra_flag[x0][y0]	**0**	**1**			
ats_hor_mode[x0][y0]	**na**	**0**	**1**	**0**	**1**
ats_ver_mode[x0][y0]	**na**	**0**	**0**	**1**	**1**
trTypeHor	0	1	2	1	2
trTypeVer	0	1	1	2	2

Table 8-17 – Specification of trTypeHor and trTypeVer depending on ats_cu_inter_flag[x][y], ats_cu_inter_horizontal_flag[x][y], and ats_cu_inter_pos_flag [x][y] and size of the block

max(nTbH, nTbW)	> 32	<= 32	
ats_cu_inter_flag[x0][y0]	X	0	1
ats_cu_inter_horizontal_flag[x0][y0]	X	na	0	1
trTypeHor	0	0	2	1
trTypeVer	0	0	1	2

8.7.4.2 Transformation process

Inputs to this process are:

– a variable nTbS specifying the sample size of scaled transform coefficients,

– a list of scaled transform coefficients x with elements x[j], with j = 0..nTbS − 1.

Output of this process is the list of transformed samples y with elements y[i], with i = 0..nTbS − 1.

The transformation matrix derivation process as specified in clause 8.7.4.3 invoked with the transform size nTbS and the transform kernel Type trType as inputs, and the transformation matrix 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:

$$y[\,i\,] = \sum_{j=0}^{nTbS-1} transMatrix[i][j] * x[j] \quad \text{with } i = 0..nTbS - 1, \tag{8-783}$$

8.7.4.3 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 and nTbs is equal to 2, the following applies:

transMatrix[m][n] = (8-784)

```
{
{  64,   64 }
{  64,  -64 }
},
```

– Otherwise, if trType is equal to 0 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (8-785)

```
{
{  64,  64,  64,  64 }
{  84,  35, -35, -84 }
{  64, -64, -64,  64 }
{  35, -84,  84, -35 }
},
```

– Otherwise, if trType is equal to 0 and nTbs is equal to 8, the following applies:

transMatrix[m][n] = (8-786)

```
{
{  64,  64,  64,  64,  64,  64,  64,  64 }
{  89,  75,  50,  18, -18, -50, -75, -89 }
{  84,  35, -35, -84, -84, -35,  35,  84 }
{  75, -18, -89, -50,  50,  89,  18, -75 }
{  64, -64, -64,  64,  64, -64, -64,  64 }
{  50, -89,  18,  75, -75, -18,  89, -50 }
{  35, -84,  84, -35, -35,  84, -84,  35 }
```

```
{  18, -50,  75, -89,  89, -75,  50, -18 }
},
```

– Otherwise, if trType is equal to 0 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-787)

```
{
{  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64},
{  90,  87,  80,  70,  57,  43,  26,   9,  -9, -26, -43, -57, -70, -80, -87, -90},
{  89,  75,  50,  18, -18, -50, -75, -89, -89, -75, -50, -18,  18,  50,  75,  89},
{  87,  57,   9, -43, -80, -90, -70, -26,  26,  70,  90,  80,  43,  -9, -57, -87},
{  84,  35, -35, -84, -84, -35,  35,  84,  84,  35, -35, -84, -84, -35,  35,  84},
{  80,   9, -70, -87, -26,  57,  90,  43, -43, -90, -57,  26,  87,  70,  -9, -80},
{  75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75},
{  70, -43, -87,   9,  90,  26, -80, -57,  57,  80, -26, -90,  -9,  87,  43, -70},
{  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64},
{  57, -80, -26,  90,  -9, -87,  43,  70, -70, -43,  87,   9, -90,  26,  80, -57},
{  50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50},
{  43, -90,  57,  26, -87,  70,   9, -80,  80,  -9, -70,  87, -26, -57,  90, -43},
{  35, -84,  84, -35, -35,  84, -84,  35,  35, -84,  84, -35, -35,  84, -84,  35},
{  26, -70,  90, -80,  43,   9, -57,  87, -87,  57,  -9, -43,  80, -90,  70, -26},
{  18, -50,  75, -89,  89, -75,  50, -18, -18,  50, -75,  89, -89,  75, -50,  18},
{   9, -26,  43, -57,  70, -80,  87, -90,  90, -87,  80, -70,  57, -43,  26,  -9}},
},
```

– Otherwise, if trType is equal to 0 and nTbs is equal to 32, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..31 (8-788)

transMatrixCol0to15 = (8-789)

```
{
{  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64 },
{  90,  90,  88,  85,  82,  78,  73,  67,  61,  54,  47,  39,  30,  22,  13,   4 },
{  90,  87,  80,  70,  57,  43,  26,   9,  -9, -26, -43, -57, -70, -80, -87, -90 },
{  90,  82,  67,  47,  22,  -4, -30, -54, -73, -85, -90, -88, -78, -61, -39, -13 },
{  89,  75,  50,  18, -18, -50, -75, -89, -89, -75, -50, -18,  18,  50,  75,  89 },
{  88,  67,  30, -13, -54, -82, -90, -78, -47,  -4,  39,  73,  90,  85,  61,  22 },
{  87,  57,   9, -43, -80, -90, -70, -26,  26,  70,  90,  80,  43,  -9, -57, -87 },
{  85,  47, -13, -67, -90, -73, -22,  39,  82,  88,  54,  -4, -61, -90, -78, -30 },
{  84,  35, -35, -84, -84, -35,  35,  84,  84,  35, -35, -84, -84, -35,  35,  84 },
{  82,  22, -54, -90, -61,  13,  78,  85,  30, -47, -90, -67,   4,  73,  88,  39 },
{  80,   9, -70, -87, -26,  57,  90,  43, -43, -90, -57,  26,  87,  70,  -9, -80 },
{  78,  -4, -82, -73,  13,  85,  67, -22, -88, -61,  30,  90,  54, -39, -90, -47 },
{  75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75 },
{  73, -30, -90, -22,  78,  67, -39, -90, -13,  82,  61, -47, -88,  -4,  85,  54 },
{  70, -43, -87,   9,  90,  26, -80, -57,  57,  80, -26, -90,  -9,  87,  43, -70 },
{  67, -54, -78,  39,  85, -22, -90,   4,  90,  13, -88, -30,  82,  47, -73, -61 },
{  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64 },
{  61, -73, -47,  82,  30, -88, -13,  90,  -4, -90,  22,  85, -39, -78,  54,  67 },
{  57, -80, -26,  90,  -9, -87,  43,  70, -70, -43,  87,   9, -90,  26,  80, -57 },
{  54, -85,  -4,  88, -47, -61,  82,  13, -90,  39,  67, -78, -22,  90, -30, -73 },
{  50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50 },
{  47, -90,  39,  54, -90,  30,  61, -88,  22,  67, -85,  13,  73, -82,   4,  78 },
{  43, -90,  57,  26, -87,  70,   9, -80,  80,  -9, -70,  87, -26, -57,  90, -43 },
{  39, -88,  73,  -4, -67,  90, -47, -30,  85, -78,  13,  61, -90,  54,  22, -82 },
{  35, -84,  84, -35, -35,  84, -84,  35,  35, -84,  84, -35, -35,  84, -84,  35 },
{  30, -78,  90, -61,   4,  54, -88,  82, -39, -22,  73, -90,  67, -13, -47,  85 },
{  26, -70,  90, -80,  43,   9, -57,  87, -87,  57,  -9, -43,  80, -90,  70, -26 },
{  22, -61,  85, -90,  73, -39,  -4,  47, -78,  90, -82,  54, -13, -30,  67, -88 },
{  18, -50,  75, -89,  89, -75,  50, -18, -18,  50, -75,  89, -89,  75, -50,  18 },
{  13, -39,  61, -78,  88, -90,  85, -73,  54, -30,   4,  22, -47,  67, -82,  90 },
{   9, -26,  43, -57,  70, -80,  87, -90,  90, -87,  80, -70,  57, -43,  26,  -9 },
{   4, -13,  22, -30,  39, -47,  54, -61,  67, -73,  78, -82,  85, -88,  90, -90 },
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-790)

transMatrixCol16to31 = (8-791)

```
{
{  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64 },
{  -4, -13, -22, -30, -39, -47, -54, -61, -67, -73, -78, -82, -85, -88, -90, -90 },
{ -90, -87, -80, -70, -57, -43, -26,  -9,   9,  26,  43,  57,  70,  80,  87,  90 },
{  13,  39,  61,  78,  88,  90,  85,  73,  54,  30,   4, -22, -47, -67, -82, -90 },
{  89,  75,  50,  18, -18, -50, -75, -89, -89, -75, -50, -18,  18,  50,  75,  89 },
{ -22, -61, -85, -90, -73, -39,   4,  47,  78,  90,  82,  54,  13, -30, -67, -88 },
{ -87, -57,  -9,  43,  80,  90,  70,  26, -26, -70, -90, -80, -43,   9,  57,  87 },
```

```
{  30,  78,  90,  61,   4, -54, -88, -82, -39,  22,  73,  90,  67,  13, -47, -85 },
{  84,  35, -35, -84, -84, -35,  35,  84,  84,  35, -35, -84, -84, -35,  35,  84 },
{ -39, -88, -73,  -4,  67,  90,  47, -30, -85, -78, -13,  61,  90,  54, -22, -82 },
{ -80,  -9,  70,  87,  26, -57, -90, -43,  43,  90,  57, -26, -87, -70,   9,  80 },
{  47,  90,  39, -54, -90, -30,  61,  88,  22, -67, -85, -13,  73,  82,   4, -78 },
{  75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75 },
{ -54, -85,   4,  88,  47, -61, -82,  13,  90,  39, -67, -78,  22,  90,  30, -73 },
{ -70,  43,  87,  -9, -90, -26,  80,  57, -57, -80,  26,  90,   9, -87, -43,  70 },
{  61,  73, -47, -82,  30,  88, -13, -90,  -4,  90,  22, -85, -39,  78,  54, -67 },
{  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64 },
{ -67, -54,  78,  39, -85, -22,  90,   4, -90,  13,  88, -30, -82,  47,  73, -61 },
{ -57,  80,  26, -90,   9,  87, -43, -70,  70,  43, -87,  -9,  90, -26, -80,  57 },
{  73,  30, -90,  22,  78, -67, -39,  90, -13, -82,  61,  47, -88,   4,  85, -54 },
{  50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50 },
{ -78,  -4,  82, -73, -13,  85, -67, -22,  88, -61, -30,  90, -54, -39,  90, -47 },
{ -43,  90, -57, -26,  87, -70,  -9,  80, -80,   9,  70, -87,  26,  57, -90,  43 },
{  82, -22, -54,  90, -61, -13,  78, -85,  30,  47, -90,  67,   4, -73,  88, -39 },
{  35, -84,  84, -35, -35,  84, -84,  35,  35, -84,  84, -35, -35,  84, -84,  35 },
{ -85,  47,  13, -67,  90, -73,  22,  39, -82,  88, -54,  -4,  61, -90,  78, -30 },
{ -26,  70, -90,  80, -43,  -9,  57, -87,  87, -57,   9,  43, -80,  90, -70,  26 },
{  88, -67,  30,  13, -54,  82, -90,  78, -47,   4,  39, -73,  90, -85,  61, -22 },
{  18, -50,  75, -89,  89, -75,  50, -18, -18,  50, -75,  89, -89,  75, -50,  18 },
{ -90,  82, -67,  47, -22,  -4,  30, -54,  73, -85,  90, -88,  78, -61,  39, -13 },
{  -9,  26, -43,  57, -70,  80, -87,  90, -90,  87, -80,  70, -57,  43, -26,   9 },
{  90, -90,  88, -85,  82, -78,  73, -67,  61, -54,  47, -39,  30, -22,  13,  -4 }
},
```

Otherwise, if trType is equal to 0 and nTbs is equal to 64, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..63 (8-792)

transMatrixCol0to15 = (8-793)

```
{
{  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64 },
{  90,  90,  90,  89,  88,  87,  86,  84,  83,  81,  79,  76,  74,  71,  69,  66 },
{  90,  90,  88,  85,  82,  78,  73,  67,  61,  54,  47,  39,  30,  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,  26,   9,  -9, -26, -43, -57, -70, -80, -87, -90 },
{  90,  84,  74,  59,  41,  20,  -2, -24, -45, -62, -76, -86, -90, -89, -83, -71 },
{  90,  82,  67,  47,  22,  -4, -30, -54, -73, -85, -90, -88, -78, -61, -39, -13 },
{  89,  79,  59,  33,   2, -28, -56, -76, -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, -89, -74, -45,  -7,  33,  66,  86,  90,  76 },
{  88,  67,  30, -13, -54, -82, -90, -78, -47,  -4,  39,  73,  90,  85,  61,  22 },
{  87,  62,  20, -28, -69, -89, -84, -56, -11,  37,  74,  90,  81,  48,   2, -45 },
{  87,  57,   9, -43, -80, -90, -70, -26,  26,  70,  90,  80,  43,  -9, -57, -87 },
{  86,  52,  -2, -56, -87, -84, -48,   7,  59,  88,  83,  45, -11, -62, -89, -81 },
{  85,  47, -13, -67, -90, -73, -22,  39,  82,  88,  54,  -4, -61, -90, -78, -30 },
{  84,  41, -24, -76, -89, -56,   7,  66,  90,  69,  11, -52, -88, -79, -28,  37 },
{  84,  35, -35, -84, -84, -35,  35,  84,  84,  35, -35, -84, -84, -35,  35,  84 },
{  83,  28, -45, -88, -74, -11,  59,  90,  62,  -7, -71, -89, -48,  24,  81,  84 },
{  82,  22, -54, -90, -61,  13,  78,  85,  30, -47, -90, -67,   4,  73,  88,  39 },
{  81,  15, -62, -90, -45,  37,  88,  69,  -7, -76, -84, -24,  56,  90,  52, -28 },
{  80,   9, -70, -87, -26,  57,  90,  43, -43, -90, -57,  26,  87,  70,  -9, -80 },
{  79,   2, -76, -81,  -7,  74,  83,  11, -71, -84, -15,  69,  86,  20, -66, -87 },
{  78,  -4, -82, -73,  13,  85,  67, -22, -88, -61,  30,  90,  54, -39, -90, -47 },
{  76, -11, -86, -62,  33,  90,  45, -52, -89, -24,  69,  83,   2, -81, -71,  20 },
{  75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75 },
{  74, -24, -90, -37,  66,  81, -11, -88, -48,  56,  86,   2, -84, -59,  45,  89 },
{  73, -30, -90, -22,  78,  67, -39, -90, -13,  82,  61, -47, -88,  -4,  85,  54 },
{  71, -37, -89,  -7,  86,  48, -62, -79,  24,  90,  20, -81, -59,  52,  84, -11 },
{  70, -43, -87,   9,  90,  26, -80, -57,  57,  80, -26, -90,  -9,  87,  43, -70 },
{  69, -48, -83,  24,  90,   2, -89, -28,  81,  52, -66, -71,  45,  84, -20, -90 },
{  67, -54, -78,  39,  85, -22, -90,   4,  90,  13, -88, -30,  82,  47, -73, -61 },
{  66, -59, -71,  52,  76, -45, -81,  37,  84, -28, -87,  20,  89, -11, -90,   2 },
{  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64 },
{  62, -69, -56,  74,  48, -79, -41,  83,  33, -86, -24,  88,  15, -90,  -7,  90 },
{  61, -73, -47,  82,  30, -88, -13,  90,  -4, -90,  22,  85, -39, -78,  54,  67 },
{  59, -76, -37,  87,  11, -90,  15,  86, -41, -74,  62,  56, -79, -33,  88,   7 },
{  57, -80, -26,  90,  -9, -87,  43,  70, -70, -43,  87,   9, -90,  26,  80, -57 },
{  56, -83, -15,  90, -28, -76,  66,  45, -87,  -2,  88, -41, -69,  74,  33, -90 },
{  54, -85,  -4,  88, -47, -61,  82,  13, -90,  39,  67, -78, -22,  90, -30, -73 },
{  52, -87,   7,  83, -62, -41,  90, -20, -76,  71,  28, -90,  33,  69, -79, -15 },
{  50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50 },
{  48, -90,  28,  66, -84,   7,  79, -74, -15,  87, -59, -37,  90, -41, -56,  88 },
{  47, -90,  39,  54, -90,  30,  61, -88,  22,  67, -85,  13,  73, -82,   4,  78 },
{  45, -90,  48,  41, -90,  52,  37, -90,  56,  33, -89,  59,  28, -88,  62,  24 },
{  43, -90,  57,  26, -87,  70,   9, -80,  80,  -9, -70,  87, -26, -57,  90, -43 },
{  41, -89,  66,  11, -79,  83, -20, -59,  90, -48, -33,  87, -71,  -2,  74, -86 },
{  39, -88,  73,  -4, -67,  90, -47, -30,  85, -78,  13,  61, -90,  54,  22, -82 },
```

```
{  37, -86,  79, -20, -52,  90, -69,   2,  66, -90,  56,  15, -76,  87, -41, -33 },
{  35, -84,  84, -35, -35,  84, -84,  35,  35, -84,  84, -35, -35,  84, -84,  35 },
{  33, -81,  87, -48, -15,  71, -90,  62,  -2, -59,  90, -74,  20,  45, -86,  83 },
{  30, -78,  90, -61,   4,  54, -88,  82, -39, -22,  73, -90,  67, -13, -47,  85 },
{  28, -74,  90, -71,  24,  33, -76,  90, -69,  20,  37, -79,  90, -66,  15,  41 },
{  26, -70,  90, -80,  43,   9, -57,  87, -87,  57,  -9, -43,  80, -90,  70, -26 },
{  24, -66,  88, -86,  59, -15, -33,  71, -90,  83, -52,   7,  41, -76,  90, -79 },
{  22, -61,  85, -90,  73, -39,  -4,  47, -78,  90, -82,  54, -13, -30,  67, -88 },
{  20, -56,  81, -90,  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, -45,  69, -84,  90, -86,  71, -48,  20,  11, -41,  66, -83,  90, -87,  74 },
{  13, -39,  61, -78,  88, -90,  85, -73,  54, -30,   4,  22, -47,  67, -82,  90 },
{  11, -33,  52, -69,  81, -88,  90, -87,  79, -66,  48, -28,   7,  15, -37,  56 },
{   9, -26,  43, -57,  70, -80,  87, -90,  90, -87,  80, -70,  57, -43,  26,  -9 },
{   7, -20,  33, -45,  56, -66,  74, -81,  86, -89,  90, -90,  87, -83,  76, -69 },
{   4, -13,  22, -30,  39, -47,  54, -61,  67, -73,  78, -82,  85, -88,  90, -90 },
{   2,  -7,  11, -15,  20, -24,  28, -33,  37, -41,  45, -48,  52, -56,  59, -62 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m][n] with m = 16..31, n = 0..63 (8-794)

transMatrixCol16to31 = (8-795)

```
{
{  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64,  64 },
{  62,  59,  56,  52,  48,  45,  41,  37,  33,  28,  24,  20,  15,  11,   7,   2 },
{  -4, -13, -22, -30, -39, -47, -54, -61, -67, -73, -78, -82, -85, -88, -90, -90 },
{ -69, -76, -83, -87, -90, -90, -89, -86, -81, -74, -66, -56, -45, -33, -20,  -7 },
{ -90, -87, -80, -70, -57, -43, -26,  -9,   9,  26,  43,  57,  70,  80,  87,  90 },
{ -56, -37, -15,   7,  28,  48,  66,  79,  87,  90,  88,  81,  69,  52,  33,  11 },
{  13,  39,  61,  78,  88,  90,  85,  73,  54,  30,   4, -22, -47, -67, -82, -90 },
{  74,  87,  90,  83,  66,  41,  11, -20, -48, -71, -86, -90, -84, -69, -45, -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,  90,  81,  56,  20 },
{ -22, -61, -85, -90, -73, -39,   4,  47,  78,  90,  82,  54,  13, -30, -67, -88 },
{ -79, -90, -76, -41,   7,  52,  83,  90,  71,  33, -15, -59, -86, -88, -66, -24 },
{ -87, -57,  -9,  43,  80,  90,  70,  26, -26, -70, -90, -80, -43,   9,  57,  87 },
{ -41,  15,  66,  90,  79,  37, -20, -69, -90, -76, -33,  24,  71,  90,  74,  28 },
{  30,  78,  90,  61,   4, -54, -88, -82, -39,  22,  73,  90,  67,  13, -47, -85 },
{  83,  86,  45, -20, -74, -90, -59,   2,  62,  90,  71,  15, -48, -87, -81, -33 },
{  84,  35, -35, -84, -84, -35,  35,  84,  84,  35, -35, -84, -84, -35,  35,  84 },
{  33, -41, -87, -76, -15,  56,  90,  66,  -2, -69, -90, -52,  20,  79,  86,  37 },
{ -39, -88, -73,  -4,  67,  90,  47, -30, -85, -78, -13,  61,  90,  54, -22, -82 },
{ -86, -74,  -2,  71,  87,  33, -48, -90, -59,  20,  83,  79,  11, -66, -89, -41 },
{ -80,  -9,  70,  87,  26, -57, -90, -43,  43,  90,  57, -26, -87, -70,   9,  80 },
{ -24,  62,  88,  28, -59, -89, -33,  56,  90,  37, -52, -90, -41,  48,  90,  45 },
{  47,  90,  39, -54, -90, -30,  61,  88,  22, -67, -85, -13,  73,  82,   4, -78 },
{  88,  56, -41, -90, -37,  59,  87,  15, -74, -79,   7,  84,  66, -28, -90, -48 },
{  75, -18, -89, -50,  50,  89,  18, -75, -75,  18,  89,  50, -50, -89, -18,  75 },
{  15, -79, -69,  33,  90,  28, -71, -76,  20,  90,  41, -62, -83,   7,  87,  52 },
{ -54, -85,   4,  88,  47, -61, -82,  13,  90,  39, -67, -78,  22,  90,  30, -73 },
{ -90, -33,  74,  69, -41, -88,  -2,  87,  45, -66, -76,  28,  90,  15, -83, -56 },
{ -70,  43,  87,  -9, -90, -26,  80,  57, -57, -80,  26,  90,   9, -87, -43,  70 },
{  -7,  88,  33, -79, -56,  62,  74, -41, -86,  15,  90,  11, -87, -37,  76,  59 },
{  61,  73, -47, -82,  30,  88, -13, -90,  -4,  90,  22, -85, -39,  78,  54, -67 },
{  90,   7, -90, -15,  88,  24, -86, -33,  83,  41, -79, -48,  74,  56, -69, -62 },
{  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64,  64, -64, -64,  64 },
{  -2, -90,  11,  89, -20, -87,  28,  84, -37, -81,  45,  76, -52, -71,  59,  66 },
{ -67, -54,  78,  39, -85, -22,  90,   4, -90,  13,  88, -30, -82,  47,  73, -61 },
{ -90,  20,  84, -45, -71,  66,  52, -81, -28,  89,   2, -90,  24,  83, -48, -69 },
{ -57,  80,  26, -90,   9,  87, -43, -70,  70,  43, -87,  -9,  90, -26, -80,  57 },
{  11,  84, -52, -59,  81,  20, -90,  24,  79, -62, -48,  86,   7, -89,  37,  71 },
{  73,  30, -90,  22,  78, -67, -39,  90, -13, -82,  61,  47, -88,   4,  85, -54 },
{  89, -45, -59,  84,   2, -86,  56,  48, -88,  11,  81, -66, -37,  90, -24, -74 },
{  50, -89,  18,  75, -75, -18,  89, -50, -50,  89, -18, -75,  75,  18, -89,  50 },
{ -20, -71,  81,   2, -83,  69,  24, -89,  52,  45, -90,  33,  62, -86,  11,  76 },
{ -78,  -4,  82, -73, -13,  85, -67, -22,  88, -61, -30,  90, -54, -39,  90, -47 },
{ -87,  66,  20, -86,  69,  15, -84,  71,  11, -83,  74,   7, -81,  76,   2, -79 },
{ -43,  90, -57, -26,  87, -70,  -9,  80, -80,   9,  70, -87,  26,  57, -90,  43 },
{  28,  52, -90,  56,  24, -84,  76,  -7, -69,  88, -37, -45,  90, -62, -15,  81 },
{  82, -22, -54,  90, -61, -13,  78, -85,  30,  47, -90,  67,   4, -73,  88, -39 },
{  84, -81,  24,  48, -89,  71,  -7, -62,  90, -59, -11,  74, -88,  45,  28, -83 },
{  35, -84,  84, -35, -35,  84, -84,  35,  35, -84,  84, -35, -35,  84, -84,  35 },
{ -37, -28,  79, -88,  52,  11, -69,  90, -66,   7,  56, -89,  76, -24, -41,  84 },
{ -85,  47,  13, -67,  90, -73,  22,  39, -82,  88, -54,  -4,  61, -90,  78, -30 },
{ -81,  89, -62,  11,  45, -83,  88, -59,   7,  48, -84,  87, -56,   2,  52, -86 },
{ -26,  70, -90,  80, -43,  -9,  57, -87,  87, -57,   9,  43, -80,  90, -70,  26 },
```

```
{  45,   2, -48,  81, -90,  74, -37, -11,  56, -84,  89, -69,  28,  20, -62,  87 },
{  88, -67,  30,  13, -54,  82, -90,  78, -47,   4,  39, -73,  90, -85,  61, -22 },
{  76, -90,  86, -66,  33,   7, -45,  74, -89,  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,  76, -56,  28,   2, -33,  59, -79,  89 },
{ -90,  82, -67,  47, -22,  -4,  30, -54,  73, -85,  90, -88,  78, -61,  39, -13 },
{ -71,  83, -89,  90, -86,  76, -62,  45, -24,   2,  20, -41,  59, -74,  84, -90 },
{  -9,  26, -43,  57, -70,  80, -87,  90, -90,  87, -80,  70, -57,  43, -26,   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,  47, -39,  30, -22,  13,  -4 },
{  66, -69,  71, -74,  76, -79,  81, -83,  84, -86,  87, -88,  89, -90,  90, -90 }
},
```

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol16to31[47 −m][n] (8-796)
with m =32..47, n = 0..63

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol0to15[63 −m][n] (8-797)
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] = (8-798)

```
{
{  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] = (8-799)

```
{
{  16,  32,  46,  59,  70,  79,  84,  87 },
{  46,  79,  87,  70,  32, -16, -59, -84 },
{  70,  84,  32, -46, -87, -59,  16,  79 },
{  84,  46, -59, -79,  16,  87,  32, -70 },
{  87, -16, -84,  32,  79, -46, -70,  59 },
{  79, -70, -16,  84, -59, -32,  87, -46 },
{  59, -87,  70, -16, -46,  84, -79,  32 },
{  32, -59,  79, -87,  84, -70,  46, -16 },
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-800)

```
{
{   8,  17,  25,  33,  41,  48,  55,  62,  67,  73,  77,  81,  84,  87,  88,  89 },
{  25,  48,  67,  81,  88,  88,  81,  67,  48,  25,   0, -25, -48, -67, -81, -88 },
{  41,  73,  88,  84,  62,  25, -17, -55, -81, -89, -77, -48,  -8,  33,  67,  87 },
{  55,  87,  81,  41, -17, -67, -89, -73, -25,  33,  77,  88,  62,   8, -48, -84 },
{  67,  88,  48, -25, -81, -81, -25,  48,  88,  67,   0, -67, -88, -48,  25,  81 },
{  77,  77,   0, -77, -77,   0,  77,  77,   0, -77, -77,   0,  77,  77,   0, -77 },
{  84,  55, -48, -87,  -8,  81,  62, -41, -88, -17,  77,  67, -33, -89, -25,  73 },
{  88,  25, -81, -48,  67,  67, -48, -81,  25,  88,   0, -88, -25,  81,  48, -67 },
{  89,  -8, -88,  17,  87, -25, -84,  33,  81, -41, -77,  48,  73, -55, -67,  62 },
{  87, -41, -67,  73,  33, -88,   8,  84, -48, -62,  77,  25, -89,  17,  81, -55 },
{  81, -67, -25,  88, -48, -48,  88, -25, -67,  81,   0, -81,  67,  25, -88,  48 },
{  73, -84,  25,  55, -89,  48,  33, -87,  67,   8, -77,  81, -17, -62,  88, -41 },
{  62, -89,  67,  -8, -55,  88, -73,  17,  48, -87,  77, -25, -41,  84, -81,  33 },
{  48, -81,  88, -67,  25,  25, -67,  88, -81,  48,   0, -48,  81, -88,  67, -25 },
{  33, -62,  81, -89,  84, -67,  41,  -8, -25,  55, -77,  88, -87,  73, -48,  17 },
{  17, -33,  48, -62,  73, -81,  87, -89,  88, -84,  77, -67,  55, -41,  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..31 (8-801)

transMatrixCol0to15 = (8-802)

```
{
{   4,   9,  13,  17,  21,  26,  30,  34,  38,  42,  46,  49,  53,  56,  60,  63 },
{  13,  26,  38,  49,  60,  69,  76,  82,  87,  89,  90,  89,  85,  81,  74,  66 },
{  21,  42,  60,  74,  84,  89,  89,  84,  74,  60,  42,  21,   0, -21, -42, -60 },
{  30,  56,  76,  88,  89,  81,  63,  38,   9, -21, -49, -71, -85, -90, -84, -69 },
{  38,  69,  87,  89,  74,  46,   9, -30, -63, -84, -90, -78, -53, -17,  21,  56 },
{  46,  78,  90,  76,  42,  -4, -49, -81, -90, -74, -38,   9,  53,  82,  89,  71 },
{  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, -88, -46,  26,  81,  84,  34, -38, -85, -78, -21,  49 },
{  71,  87,  34, -46, -89, -63,  13,  78,  82,  21, -56, -90, -53,  26,  84,  76 },
{  76,  81,   9, -71, -84, -17,  66,  87,  26, -60, -89, -34,  53,  90,  42, -46 },
{  81,  71, -17, -87, -60,  34,  90,  46, -49, -89, -30,  63,  85,  13, -74, -78 },
{  84,  60, -42, -89, -21,  74,  74, -21, -89, -42,  60,  84,   0, -84, -60,  42 },
{  87,  46, -63, -78,  21,  90,  26, -76, -66,  42,  88,   4, -85, -49,  60,  81 },
{  89,  30, -78, -56,  60,  76, -34, -88,   4,  89,  26, -81, -53,  63,  74, -38 },
{  90,  13, -88, -26,  84,  38, -78, -49,  71,  60, -63, -69,  53,  76, -42, -82 },
{  90,  -4, -90,   9,  89, -13, -89,  17,  88, -21, -87,  26,  85, -30, -84,  34 },
{  89, -21, -84,  42,  74, -60, -60,  74,  42, -84, -21,  89,   0, -89,  21,  84 },
{  88, -38, -71,  69,  42, -87,  -4,  89, -34, -74,  66,  46, -85,  -9,  89, -30 },
{  85, -53, -53,  85,   0, -85,  53,  53, -85,   0,  85, -53, -53,  85,   0, -85 },
{  82, -66, -30,  90, -42, -56,  87, -13, -76,  74,  17, -88,  53,  46, -89,  26 },
{  78, -76,  -4,  81, -74,  -9,  82, -71, -13,  84, -69, -17,  85, -66, -21,  87 },
{  74, -84,  21,  60, -89,  42,  42, -89,  60,  21, -84,  74,   0, -74,  84, -21 },
{  69, -89,  46,  30, -84,  78, -17, -56,  90, -60, -13,  76, -85,  34,  42, -88 },
{  63, -90,  66,  -4, -60,  90, -69,   9,  56, -89,  71, -13, -53,  89, -74,  17 },
{  56, -88,  81, -38, -21,  71, -90,  69, -17, -42,  82, -87,  53,   4, -60,  89 },
{  49, -82,  89, -66,  21,  30, -71,  90, -78,  42,   9, -56,  85, -87,  60, -13 },
{  42, -74,  89, -84,  60, -21, -21,  60, -84,  89, -74,  42,   0, -42,  74, -89 },
{  34, -63,  82, -90,  84, -66,  38,  -4, -30,  60, -81,  90, -85,  69, -42,   9 },
{  26, -49,  69, -82,  89, -89,  81, -66,  46, -21,  -4,  30, -53,  71, -84,  90 },
{  17, -34,  49, -63,  74, -82,  88, -90,  89, -84,  76, -66,  53, -38,  21,  -4 },
{   9, -17,  26, -34,  42, -49,  56, -63,  69, -74,  78, -82,  85, -88,  89, -90 },
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-803)

transMatrixCol16to31 = (8-804)

```
{
{  66,  69,  71,  74,  76,  78,  81,  82,  84,  85,  87,  88,  89,  89,  90,  90 },
{  56,  46,  34,  21,   9,  -4, -17, -30, -42, -53, -63, -71, -78, -84, -88, -90 },
{ -74, -84, -89, -89, -84, -74, -60, -42, -21,   0,  21,  42,  60,  74,  84,  89 },
{ -46, -17,  13,  42,  66,  82,  90,  87,  74,  53,  26,  -4, -34, -60, -78, -89 },
{  81,  90,  82,  60,  26, -13, -49, -76, -89, -85, -66, -34,   4,  42,  71,  88 },
{  34, -13, -56, -84, -89, -69, -30,  17,  60,  85,  88,  66,  26, -21, -63, -87 },
{ -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 },
{  89,  71,   9, -60, -90, -63,   4,  69,  89,  53, -17, -76, -87, -42,  30,  82 },
{   9, -66, -89, -42,  38,  88,  69,  -4, -74, -85, -30,  49,  90,  60, -17, -81 },
{ -90, -49,  38,  89,  56, -30, -88, -63,  21,  85,  69, -13, -82, -74,   4,  78 },
{   4,  82,  69, -21, -88, -56,  38,  90,  42, -53, -89, -26,  66,  84,   9, -76 },
{  89,  21, -74, -74,  21,  89,  42, -60, -84,   0,  84,  60, -42, -89, -21,  74 },
{ -17, -90, -30,  74,  69, -38, -89,  -9,  84,  53, -56, -82,  13,  89,  34, -71 },
{ -87,   9,  90,  21, -82, -49,  66,  71, -42, -85,  13,  90,  17, -84, -46,  69 },
{  30,  87, -17, -89,   4,  90,   9, -89, -21,  85,  34, -81, -46,  74,  56, -66 },
{  82, -38, -81,  42,  78, -46, -76,  49,  74, -53, -71,  56,  69, -60, -66,  63 },
{ -42, -74,  60,  60, -74, -42,  84,  21, -89,   0,  89, -21, -84,  42,  74, -60 },
{ -76,  63,  49, -84, -13,  90, -26, -78,  60,  53, -82, -17,  90, -21, -81,  56 },
{  53,  53, -85,   0,  85, -53, -53,  85,   0, -85,  53,  53, -85,   0,  85, -53 },
{  69, -81,  -4,  84, -63, -34,  90, -38, -60,  85,  -9, -78,  71,  21, -89,  49 },
{ -63, -26,  88, -60, -30,  89, -56, -34,  89, -53, -38,  90, -49, -42,  90, -46 },
{ -60,  89, -42, -42,  89, -60, -21,  84, -74,   0,  74, -84,  21,  60, -89,  42 },
{  71,  -4, -66,  89, -49, -26,  82, -81,  21,  53, -90,  63,   9, -74,  87, -38 },
{  49, -88,  76, -21, -46,  87, -78,  26,  42, -85,  81, -30, -38,  84, -82,  34 },
{ -78,  34,  26, -74,  90, -66,  13,  46, -84,  85, -49,  -9,  63, -89,  76, -30 },
{ -38,  76, -90,  74, -34, -17,  63, -88,  84, -53,   4,  46, -81,  89, -69,  26 },
{  84, -60,  21,  21, -60,  84, -89,  74, -42,   0,  42, -74,  89, -84,  60, -21 },
{  26, -56,  78, -89,  87, -71,  46, -13, -21,  53, -76,  89, -88,  74, -49,  17 },
{ -88,  78, -63,  42, -17,  -9,  34, -56,  74, -85,  90, -87,  76, -60,  38, -13 },
{ -13,  30, -46,  60, -71,  81, -87,  90, -89,  85, -78,  69, -56,  42, -26,   9 },
{  90, -89,  87, -84,  81, -76,  71, -66,  60, -53,  46, -38,  30, -21,  13,  -4 },
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (8-805)

```
{
{  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] = (8-806)

```
{
{  87,  84,  79,  70,  59,  46,  32,  16 },
{  84,  59,  16, -32, -70, -87, -79, -46 },
{  79,  16, -59, -87, -46,  32,  84,  70 },
{  70, -32, -87, -16,  79,  59, -46, -84 },
{  59, -70, -46,  79,  32, -84, -16,  87 },
{  46, -87,  32,  59, -84,  16,  70, -79 },
{  32, -79,  84, -46, -16,  70, -87,  59 },
{  16, -46,  70, -84,  87, -79,  59, -32 },
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-807)

```
{
{  89,  88,  87,  84,  81,  77,  73,  67,  62,  55,  48,  41,  33,  25,  17,   8 },
{  88,  81,  67,  48,  25,   0, -25, -48, -67, -81, -88, -88, -81, -67, -48, -25 },
{  87,  67,  33,  -8, -48, -77, -89, -81, -55, -17,  25,  62,  84,  88,  73,  41 },
{  84,  48,  -8, -62, -88, -77, -33,  25,  73,  89,  67,  17, -41, -81, -87, -55 },
{  81,  25, -48, -88, -67,   0,  67,  88,  48, -25, -81, -81, -25,  48,  88,  67 },
{  77,   0, -77, -77,   0,  77,  77,   0, -77, -77,   0,  77,  77,   0, -77, -77 },
{  73, -25, -89, -33,  67,  77, -17, -88, -41,  62,  81,  -8, -87, -48,  55,  84 },
{  67, -48, -81,  25,  88,   0, -88, -25,  81,  48, -67, -67,  48,  81, -25, -88 },
{  62, -67, -55,  73,  48, -77, -41,  81,  33, -84, -25,  87,  17, -88,  -8,  89 },
{  55, -81, -17,  89, -25, -77,  62,  48, -84,  -8,  88, -33, -73,  67,  41, -87 },
{  48, -88,  25,  67, -81,   0,  81, -67, -25,  88, -48, -48,  88, -25, -67,  81 },
{  41, -88,  62,  17, -81,  77,  -8, -67,  87, -33, -48,  89, -55, -25,  84, -73 },
{  33, -81,  84, -41, -25,  77, -87,  48,  17, -73,  88, -55,  -8,  67, -89,  62 },
{  25, -67,  88, -81,  48,   0, -48,  81, -88,  67, -25, -25,  67, -88,  81, -48 },
{  17, -48,  73, -87,  88, -77,  55, -25,  -8,  41, -67,  84, -89,  81, -62,  33 },
{   8, -25,  41, -55,  67, -77,  84, -88,  89, -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..31 (8-808)

transMatrixCol0to15 = (8-809)

```
{
{  90,  90,  89,  89,  88,  87,  85,  84,  82,  81,  78,  76,  74,  71,  69,  66 },
{  90,  88,  84,  78,  71,  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 },
{  89,  78,  60,  34,   4, -26, -53, -74, -87, -90, -82, -66, -42, -13,  17,  46 },
{  88,  71,  42,   4, -34, -66, -85, -89, -76, -49, -13,  26,  60,  82,  90,  81 },
{  87,  63,  21, -26, -66, -88, -85, -60, -17,  30,  69,  89,  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, -87, -76, -17,  53,  89,  69,   4, -63, -90, -60,   9,  71,  89 },
{  81,  17, -60, -90, -49,  30,  85,  74,   4, -69, -88, -38,  42,  89,  66,  -9 },
{  78,   4, -74, -82, -13,  69,  85,  21, -63, -88, -30,  56,  89,  38, -49, -90 },
{  76,  -9, -84, -66,  26,  89,  53, -42, -90, -38,  56,  88,  21, -69, -82,  -4 },
{  74, -21, -89, -42,  60,  84,   0, -84, -60,  42,  89,  21, -74, -74,  21,  89 },
{  71, -34, -89, -13,  82,  56, -53, -84,   9,  89,  38, -69, -74,  30,  90,  17 },
{  69, -46, -84,  17,  90,  13, -85, -42,  71,  66, -49, -82,  21,  90,   9, -87 },
{  66, -56, -74,  46,  81, -34, -85,  21,  89,  -9, -90,  -4,  89,  17, -87, -30 },
{  63, -66, -60,  69,  56, -71, -53,  74,  49, -76, -46,  78,  42, -81, -38,  82 },
{  60, -74, -42,  84,  21, -89,   0,  89, -21, -84,  42,  74, -60, -60,  74,  42 },
{  56, -81, -21,  90, -17, -82,  53,  60, -78, -26,  90, -13, -84,  49,  63, -76 },
{  53, -85,   0,  85, -53, -53,  85,   0, -85,  53,  53, -85,   0,  85, -53, -53 },
{  49, -89,  21,  71, -78,  -9,  85, -60, -38,  90, -34, -63,  84,  -4, -81,  69 },
{  46, -90,  42,  49, -90,  38,  53, -89,  34,  56, -89,  30,  60, -88,  26,  63 },
{  42, -89,  60,  21, -84,  74,   0, -74,  84, -21, -60,  89, -42, -42,  89, -60 },
{  38, -87,  74,  -9, -63,  90, -53, -21,  81, -82,  26,  49, -89,  66,   4, -71 },
{  34, -82,  84, -38, -30,  81, -85,  42,  26, -78,  87, -46, -21,  76, -88,  49 },
{  30, -76,  89, -63,   9,  49, -85,  84, -46, -13,  66, -90,  74, -26, -34,  78 },
{  26, -69,  89, -81,  46,   4, -53,  84, -88,  63, -17, -34,  74, -90,  76, -38 },
{  21, -60,  84, -89,  74, -42,   0,  42, -74,  89, -84,  60, -21, -21,  60, -84 },
{  17, -49,  74, -88,  89, -76,  53, -21, -13,  46, -71,  87, -89,  78, -56,  26 },
{  13, -38,  60, -76,  87, -90,  85, -74,  56, -34,   9,  17, -42,  63, -78,  88 },
{   9, -26,  42, -56,  69, -78,  85, -89,  90, -87,  81, -71,  60, -46,  30, -13 },
{   4, -13,  21, -30,  38, -46,  53, -60,  66, -71,  76, -81,  84, -87,  89, -90 },
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-810)

transMatrixCol16to31 = (8-811)

```
{
{  63,  60,  56,  53,  49,  46,  42,  38,  34,  30,  26,  21,  17,  13,   9,   4 },
```

```
{ -66, -74, -81, -85, -89, -90, -89, -87, -82, -76, -69, -60, -49, -38, -26, -13 },
{ -60, -42, -21,   0,  21,  42,  60,  74,  84,  89,  89,  84,  74,  60,  42,  21 },
{  69,  84,  90,  85,  71,  49,  21,  -9, -38, -63, -81, -89, -88, -76, -56, -30 },
{  56,  21, -17, -53, -78, -90, -84, -63, -30,   9,  46,  74,  89,  87,  69,  38 },
{ -71, -89, -82, -53,  -9,  38,  74,  90,  81,  49,   4, -42, -76, -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 },
{  49, -21, -78, -85, -38,  34,  84,  81,  26, -46, -88, -74, -13,  56,  90,  66 },
{ -76, -84, -26,  53,  90,  56, -21, -82, -78, -13,  63,  89,  46, -34, -87, -71 },
{ -46,  42,  90,  53, -34, -89, -60,  26,  87,  66, -17, -84, -71,   9,  81,  76 },
{  78,  74, -13, -85, -63,  30,  89,  49, -46, -90, -34,  60,  87,  17, -71, -81 },
{  42, -60, -84,   0,  84,  60, -42, -89, -21,  74,  74, -21, -89, -42,  60,  84 },
{ -81, -60,  49,  85,  -4, -88, -42,  66,  76, -26, -90, -21,  78,  63, -46, -87 },
{ -38,  74,  63, -53, -81,  26,  89,   4, -88, -34,  76,  60, -56, -78,  30,  89 },
{  82,  42, -76, -53,  69,  63, -60, -71,  49,  78, -38, -84,  26,  88, -13, -90 },
{  34, -84, -30,  85,  26, -87, -21,  88,  17, -89, -13,  89,   9, -90,  -4,  90 },
{ -84, -21,  89,   0, -89,  21,  84, -42, -74,  60,  60, -74, -42,  84,  21, -89 },
{ -30,  89,  -9, -85,  46,  66, -74, -34,  89,  -4, -87,  42,  69, -71, -38,  88 },
{  85,   0, -85,  53,  53, -85,   0,  85, -53, -53,  85,   0, -85,  53,  53, -85 },
{  26, -89,  46,  53, -88,  17,  74, -76, -13,  87, -56, -42,  90, -30, -66,  82 },
{ -87,  21,  66, -85,  17,  69, -84,  13,  71, -82,   9,  74, -81,   4,  76, -78 },
{ -21,  84, -74,   0,  74, -84,  21,  60, -89,  42,  42, -89,  60,  21, -84,  74 },
{  88, -42, -34,  85, -76,  13,  60, -90,  56,  17, -78,  84, -30, -46,  89, -69 },
{  17, -74,  89, -53, -13,  71, -89,  56,   9, -69,  90, -60,  -4,  66, -90,  63 },
{ -89,  60,  -4, -53,  87, -82,  42,  17, -69,  90, -71,  21,  38, -81,  88, -56 },
{ -13,  60, -87,  85, -56,   9,  42, -78,  90, -71,  30,  21, -66,  89, -82,  49 },
{  89, -74,  42,   0, -42,  74, -89,  84, -60,  21,  21, -60,  84, -89,  74, -42 },
{   9, -42,  69, -85,  90, -81,  60, -30,  -4,  38, -66,  84, -90,  82, -63,  34 },
{ -90,  84, -71,  53, -30,   4,  21, -46,  66, -81,  89, -89,  82, -69,  49, -26 },
{  -4,  21, -38,  53, -66,  76, -84,  89, -90,  88, -82,  74, -63,  49, -34,  17 },
{  90, -89,  88, -85,  82, -78,  74, -69,  63, -56,  49, -42,  34, -26,  17,  -9 },
},
```

8.7.5 Picture construction process

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,

– two variables nCbW and nCbH specifying the width and the height of the current block,

– a variable cIdx specifying the colour component of the current block,

– an (nCbW)x(nCbH) array predSamples specifying the predicted samples of the current block,

– an (nCbW)x(nCbH) array resSamples specifying the residual samples of the current block.

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 S_L and the function clipCidx1 corresponds to $Clip1_Y$.

– Otherwise, if cIdx is equal to 1, recSamples corresponds to the reconstructed chroma sample array S_{Cb} and the function clipCidx1 corresponds to $Clip1_C$.

– Otherwise (cIdx is equal to 2), recSamples corresponds to the reconstructed chroma sample array S_{Cr} and the function clipCidx1 corresponds to $Clip1_C$.

The (nCbW)x(nCbH) block of the reconstructed sample array recSamples at location (xCurr, yCurr) is derived as follows:

recSamples[xCurr + i][yCurr + j] = clipCidx1(predSamples[i][j] + resSamples[i][j]) (8-812)
with i = 0..nCbW − 1, j = 0..nCbH − 1

8.7.6 Post-reconstruction filter process

Inputs to this process are:

– a location (xCb, yCb) specifying the top-left sample of the current luma block relative to the top-left sample of the current picture,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

– a luma quantization parameter Qp_Y of the current block,

– an array recSamples specifying the reconstructed luma samples of the current block,

Outputs of this process is the modified luma samples of the current block before in-loop filtering.

The modified luma samples of the current block postRecSamples with x = 0..nCbW – 1, y = 0..nCbH – 1 are derived as follows:

– The modified filtered reconstructed luma picture sample postRecSamples[xCb + x][yCb + y] is derived based on accumulated filtered sample accFlt[xCb + x][yCb + y] with x = 0..nCbW – 1, y = 0..nCbH – 1 as follows:

$$postRecSamples_L[xCb + x][yCb + y] = Clip1_Y((accFlt[xCb + x][yCb + y] + 2) >> 2) \quad (8\text{-}813)$$

– The accumulated filtered samples accFlt[xCb + x][yCb + y] are initialized to be equal to zero and then derived based on padded reconstructed luma samples recSamplesPad which are derived according to padding process specified in clause 8.7.6.1, with x = -1..nCbW – 1, y = -1..nCbH – 1 as follows:

  – The input array for filtering inFilt including a current pixel and three its neighbouring pixels according to the scan template scanTmpl with i = 0..3 is derived as follows:

$$inFilt[i] = recSamplesPad[xCb + x + scanTmpl[i].x][yCb + y + scanTmpl[i].y], \quad (8\text{-}814)$$

$$\text{where } scanTmpl[4] (y, x) =\{ (0,0), (0,1), (1,0), (1,1) \} \quad (8\text{-}815)$$

  – The Hadamard spectrum components fHad are derived by performing Hadamard transform as follows:

$$fHad[0] = inFilt[0] + inFilt[2] + inFilt[1] + inFilt[3] \quad (8\text{-}816)$$

$$fHad[1] = inFilt[0] + inFilt[2] – inFilt[1] – inFilt[3] \quad (8\text{-}817)$$

$$fHad[2] = inFilt[0] – inFilt[2] + inFilt[1] – inFilt[3] \quad (8\text{-}818)$$

$$fHad[3] = inFilt[0] – inFilt[2] – inFilt[1] + inFilt[3] \quad (8\text{-}819)$$

  – The filtered Hadamard spectrum components fHadFilt are derived using look-up table as follows with i = 1..3, where variables THR, tblShift and look-up-table array LUT are derived based on luma quantization parameter Qp_Y according to 8.7.6.1:

$$fHadFilt[i] = \begin{cases} fHad[i] \;\; ; \;\; Abs(fHad[i]) \geq THR \\ LUT[(fHad[i] + (1 \ll (tblShift - 1))) \gg tblShift] \; ; \; fHad[i] > 0 \\ -LUT[(-(fHad[i] + (1 \ll (tblShift - 1))) \gg tblShift] \; ; \; fHad[i] \leq 0 \end{cases} \quad (8\text{-}820)$$

$$fHadFilt[0] = fHad[0] \quad (8\text{-}821)$$

  – The filtered pixels invHadFilt are derived by performing inverse Hadamard transform for filtered spectrum components fHadFilt as follows:

$$invHadFilt[0] = fHadFilt[0] + fHadFilt[2] + fHadFilt[1] + fHadFilt[3] \quad (8\text{-}822)$$

$$invHadFilt[1] = fHadFilt[0] + fHadFilt[2] – fHadFilt[1] – fHadFilt[3] \quad (8\text{-}823)$$

$$invHadFilt[2] = fHadFilt[0] – fHadFilt[2] + fHadFilt[1] – fHadFilt[3] \quad (8\text{-}824)$$

$$invHadFilt[3] = fHadFilt[0] – fHadFilt[2] – fHadFilt[1] + fHadFilt[3] \quad (8\text{-}825)$$

  – The filtered pixels invHadFilt are accumulated in accumulation buffer accFlt according to the scan template scanTmpl with i = 0..3 as follows:

$$accFlt[xCb + x + scanTmpl[i].x][yCb + y + scanTmpl[i].y] += invHadFilt[i] \quad (8\text{-}826)$$

When one or more of the following conditions are true, the modified sample value, r_0' is substituted by the corresponding input sample value r_0 as follows:

– (nCbW = = 4) && (nCbH = = 4) (8-827)

– pred_mode_flag[xCb][yCb] = = 0 && min(nCbW, nCbH) >= 32 (8-828)

– QpY <= 17 (8-829)

8.7.6.1 Padding process for post-reconstruction filter

Inputs to this process are:

– a location (xCb, yCb) specifying the top-left sample of the current luma block relative to the top-left sample of the current picture,

– two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

– an array recSamples specifying the reconstructed luma samples of the current block,

The output array of padded reconstructed luma samples of the current block recSamplesPad[x][y] with x = –1..nCbW, y = –1..nCbH are derrived as follows:

When 0 ⩽ x ⩽ nCbW – 1 and 0 ⩽ y ⩽ nCbH – 1, the following applies:
recSamplesPad[x][y] = recSamples[x][y] (8-830)

– Otherwise,

– The availability derivation process for a block in z-scan order as specified in clause 6.4.1 is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbY, yNbY) set equal to (xCb + x, yCb + y) as inputs, and the output is assigned to availableN.

– the variable dx is set to 0 and variable dy is set to 0,

– when x = = –1 and availableN is equal to FALSE, dx = 1,

– when x = = nCbW and availableN is equal to FALSE, dx = –1,

– when y = = –1 and availableN is equal to FALSE, dy = 1,

– when y = = nCbH and availableN is equal to FALSE, dy = –1,

recSamplesPad[x][y] = recSamples[x + dx][y + dy] (8-831)

8.7.6.2 Derivation process for post-reconstruction filter look-up table

Input of this process is:

– a luma quantization parameter Qp_Y of the current block,

Outputs of this process are:

– a lool-up-table LUT used for filtering of the block,

– a look-up-table access threshold THR

– a look-up-table index shift tblShift.

The look-up-table and corresponding parameters used for filtering of the block are selected from the set of the look-up-tables based on luma quantization parameter Qp_Y.

The index of look-up-table in the set qpIdx is derrived as follows:

if(pred_mode_flag[xCb][yCb] = = 0 && nCbW = = nCbH && min(nCbW, nCbH) >= 32)
qpIdx = Clip3(0, 4, (Qp_Y – 28 + (1 << 2)) >> 3) (8-832)
else
qpIdx = Clip3(0, 4, (Qp_Y – 20 + (1 << 2)) >> 3) (8-833)

The lool-up-table LUT used for filtering of the block is derrived by selecting array from setOfLUT based on qpIdx:

LUT = setOfLUT[qpIdx] (8-834)

setOfLUT[5][16] = (8-835)

```
{ 0, 0, 2,  6, 10, 14, 19, 23, 28, 32,  36,  41,  45,  49,  53,  57, },
{ 0, 0, 5, 12, 20, 29, 38, 47, 56, 65,  73,  82,  90,  98, 107, 115, },
{ 0, 0, 1,  4,  9, 16, 24, 32, 41, 50,  59,  68,  77,  86,  94, 103, },
{ 0, 0, 3,  9, 19, 32, 47, 64, 81, 99, 117, 135, 154, 179, 205, 230, },
{ 0, 0, 0,  2,  6, 11, 18, 27, 38, 51,  64,  96, 128, 160, 192, 224, },
```

The variable tblShift is derrived as follows:

tblShift = tblThrLog2[qpIdx] − 4 (8-836)

tblThrLog2[5] = { 6, 7, 7, 8, 8 } (8-837)

The look-up-table access threshold THR is derrived as follows:

$$THR = (1 << tblThrLog2[qpIdx]) – (1 << tblShift) \quad (8\text{-}838)$$

8.8 In-loop filter process

8.8.1 General

This clause specifies the application of two in-loop filters.

The two in-loop filters, namely deblocking filter and adaptive loop filter, are applied as specified by the following ordered steps:

1. For the deblocking filter, the following applies:
   - If sps_addb_flag is equal to 0, the deblocking filter process as specified in clause 8.8.2 is invoked .
   - Otherwise, sps_addb_flag is equal to 1, the deblocking filter process as specified in clause 8.8.3 is invoked .
2. When sps_alf_flag is equal to 1, the following applies:
   - The adaptive loop filter process as specified in clause 8.8.4 is invoked with the reconstructed picture sample arrays S_L, S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L, S'_{Cb} and S'_{Cr} application of adaptive loop filter as outputs.
   - The arrays S'_L, S'_{Cb} and S'_{Cr} are assigned to the arrays S_L, S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.

8.8.2 Deblocking filter process

This process is invoked with sps_addb_flag is set equal to 0.

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 log2CbWidth and log2CbHeight specifying the width and the height of the current luma coding block,
- a reconstructed picture sample arrays prior to deblocking $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$,.
- a luma motion vectors mvL0 and mvL1,
- a reference indices refIdxL0 and refIdxL1,

Outputs of this process are the modified reconstructed picture sample arrays after deblocking $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$.

Filtering in the horizontal direction (across the vertical block edges) is conducted first. Then filtering in the vertical direction (across the horizontal block edges) is conducted with samples modified by the filtering in the horizontal direction as input. The horizontal and vertical edges in the coding tree blocks of each coding tree unit are processed separately on a coding unit basis. 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. 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 most left side and most right side edges are filtered according to the neighbouring blocks availabilities. The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in clause 6.4.2.

The deblocking filter process is applied to all the block edges a picture, except the edges that are at the boundary of the picture, for which the deblocking filter process is disabled by slice_deblocking_filter_flag.

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 8-18.

Table 8-18 – Name of association to edgeType

edgeType	Name of edgeType
0 (vertical edge)	EDGE_VER
1 (horizontal edge)	EDGE_HOR

The filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

For vertical edge and horizontal edge, the following two cases apply separately:

- For vertical edge, edgeType is set equal to 0 and the next step applies.
- For horizontal edge, edgeType is set equal to 1 and the next step applies.

For the variable xCb', the following cases apply separately:

- If log2CbWidth <= 6, xCb' is set equal to xCb and the next step applies.
- Otherwise, if log2CbWidth > 6, the following two cases apply separately:
  - xCb' is set equal to xCb and the next step applies.
  - xCb' is set equal to (xCb + (1 << 6)) and the next step applies.

For the variable yCb', the following cases apply separately:

- If log2CbHeight <= 6, yCb' is set equal to yCb and the next step applies.
- Otherwise, if log2CbHeight > 6, the following two cases apply separately:
  - yCb' is set equal to yCb and the next step applies.
  - yCb' is set equal to (yCb + (1 << 6)) and the next step applies.

The variables xPi, yPj, xDi, yD_j, xN, and yN are derived as follows:

- xPi is set equal to i, yPj is set equal to j, xN is set equal to ((1 << log2CbWidth) – 1), and yN is set equal to ((1 << log2CbHeight) – 1).
- xDi is set equal to (xPi >> 2), yDj is set equal to (yPj >> 2).

The filtering pixels in the luma coding block to be filtered are set with the following conditions:

- If edgeType is equal to EDGE_VER, pixels for xPi with i = 0..xN and yPj with j = 0 are set to be filtered.
- Otherwise, if edgeType is equal to EDGE_HOR,
  - If availLR is equal to LR_11 or LR_10, pixels for xPi with i = 0 and yP_j with j = 0..yN are set to be filtered.
  - If availLR is equal to LR_11 or LR_01, pixels for xPi with i = xN + 1 and yP_j with j = 0..yN are set to be filtered.

The filtering process is applied as follows:

1. The sample values p_0 and q_0 are derived as follows:
   A. If edgeType is equal to EDGE_VER, p_0 is set equal to recPictureL[xCb' + xPi − 1][yCb' + yPj] and q_0 is set equal to recPictureL[xCb' + xPi][yCb' + yPj].
   B. Otherwise (edgeType is equal to EDGE_HOR), p_0 is set equal to recPictureL[xCb' + xPi][yCb' + yPj − 1] and q_0 is set equal to recPictureL[xCb' + xPi][yCb' + yPj].
2. The variable bS_L[xDi][yDj] is derived as follows:
   A. If the sample p_0 or q_0 is in the luma coding block of a coding unit coded with intra prediction mode, bS_L[xDi][yDj] is set equal to 0.
   B. Otherwise, if the sample p_0 or q_0 is in the luma coding block of a coding unit coded with cbf_luma equal to 1, bS_L[xDi][yDj] is set equal to 1.
   C. Otherwise, if the sample p_0 or q_0 is in the luma coding block of a coding unit coded with ibc prediction mode, bS_L[xDi][yDj] is set equal to 2.
   D. Otherwise, the following is applies:
      i. The variables mv0_l0_x, mv0_l0_y, mv0_l1_x , mv0_l1_y, mv1_l0_x, mv1_l0_y, mv1_l1_x, mv1_l1_y, refIdx_0_L0, refIdx_0_L1, refIdx_1_L0, and refIdx_1_L1 are derived as follows:
         1. If edgeType is equal to EDGE_VER, the following is applies:

$$mv0\_l0\_x = mvL0[xCb' + xDi][yCb' + yDj][0] \quad (8\text{-}839)$$

mv0_l0_y = mvL0[xCb' + xDi][yCb' + yDj][1] (8-840)

mv0_l1_x = mvL1[xCb' + xDi][yCb' + yDj][0] (8-841)

mv0_l1_y = mvL1[xCb' + xDi][yCb' + yDj][1] (8-842)

mv1_l0_x = mvL0[xCb' + xDi – 1][yCb' + yDj][0] (8-843)

mv1_l0_y = mvL0[xCb' + xDi – 1][yCb' + yDj][1] (8-844)

mv1_l1_x = mvL1[xCb' + xDi – 1][yCb' + yDj][0] (8-845)

mv1_l1_y = mvL1[xCb' + xDi – 1][yCb' + yDj][1] (8-846)

refIdx_0_L0 = ref_idx_l0[xCb' + xDi][yCb' + yDj] (8-847)

refIdx_0_L1 = ref_idx_l1[xCb' + xDi][yCb' + yDj] (8-848)

refIdx_1_L0 = ref_idx_l0[xCb' + xDi – 1][yCb' + yDj] (8-849)

refIdx_1_L1 = ref_idx_l1[xCb' + xDi – 1][yCb' + yDj] (8-850)

2. Otherwise, if edgeType is equal to EDGE_HOR, the following is applies:

mv0_l0_x = mvL0[xCb' + xDi][yCb' + yDj][0] (8-851)

mv0_l0_y = mvL0[xCb' + xDi][yCb' + yDj][1] (8-852)

mv0_l1_x = mvL1[xCb' + xDi][yCb' + yDj][0] (8-853)

mv0_l1_y = mvL1[xCb' + xDi][yCb' + yDj][1] (8-854)

mv1_l0_x = mvL0[xCb' + xDi][yCb' + yDj – 1][0] (8-855)

mv1_l0_y = mvL0[xCb' + xDi][yCb' + yDj – 1][1] (8-856)

mv1_l1_x = mvL1[xCb' + xDi][yCb' + yDj – 1][0] (8-857)

mv1_l1_y = mvL1[xCb' + xDi][yCb' + yDj – 1][1] (8-858)

refIdx_0_L0 = ref_idx_l0[xCb' + xDi][yCb' + yDj] (8-859)

refIdx_0_L1 = ref_idx_l1[xCb' + xDi][yCb' + yDj] (8-860)

refIdx_1_L0 = ref_idx_l0[xCb' + xDi][yCb' + yDj – 1] (8-861)

refIdx_1_L1 = ref_idx_l1[xCb' + xDi][yCb' + yDj – 1] (8-862)

3. If refIdx_0_L0 is unavailabe, mv0_l0_x and mv0_l0_y are set equal to 0.
4. If refIdx_0_L1 is unavailabe, mv0_l1_x and mv0_l1_y are set equal to 0.
5. If refIdx_1_L0 is unavailabe, mv1_l0_x and mv1_l0_y are set equal to 0.
6. If refIdx_1_L1 is unavailabe, mv1_l1_x and mv1_l1_y are set equal to 0.

ii. If refIdx_0_L0 is equal to refIdx_0_L1 and refIdx_1_L0 is equal to refIdx_1_L1,

bS_L[xDi][yDj] = (Abs(mv0_l0_x - mv1_l0_x) >= 4 ||
Abs(mv0_l0_y - mv1_l0_y) >= 4 ||
Abs(mv0_l1_x - mv1_l1_x) >= 4 ||
Abs(mv0_l1_y - mv1_l1_y) >= 4) ? 2 : 3 (8-863)

iii. Otherwise, if refIdx_0_L0 is equal to refIdx_1_L1 and refIdx_0_L1 is equal to refIdx_1_L0,

$$bS_L[xDi][yDj] = (Abs(mv0\_l0\_x - mv1\_l1\_x) >= 4 \|\| Abs(mv0\_l0\_y - mv1\_l1\_y) >= 4 \|\| Abs(mv0\_l1\_x - mv1\_l0\_x) >= 4 \|\| Abs(mv0\_l1\_y - mv1\_l0\_y) >= 4) ? 2 : 3 \quad (8\text{-}864)$$

iv. Otherwise, $bS_L[xDi][yDj] = 2$ (8-865)

3. The variable sT[xDi][yDj] and sT'[xDi][yDj] are dereived as follows:
   A. Qp_Y values of the coding units which include the luma coding blocks containing the sample q_0 and p_0.
   B. sT[xDi][yDj] is derived from Table 8-19**.**
   C. sT'[xDi][yDj] is equal to sT[xDi][yDj] << ($BitDepth_Y$ − 8).
4. When sT'[xDi][yDj] is greater than 0, the following ordered steps apply:
   A. If edgeType is equal to EDGE_VER, the sample values sA, sB, sC, and sD are derived as follows:

$$sA = recPicture_L [xCb' + xPi − 2][yCb' + yPj] \quad (8\text{-}866)$$

$$sB = recPicture_L [xCb' + xPi − 1][yCb' + yPj] \quad (8\text{-}867)$$

$$sC = recPicture_L [xCb' + xPi][yCb' + yPj] \quad (8\text{-}868)$$

$$sD = recPicture_L [xCb' + xPi + 1][yCb' + yPj] \quad (8\text{-}869)$$

   B. Otherwise (edgeType is equal to EDGE_HOR), the sample values sA, sB, sC, and sD are derived as follows:

$$sA = recPicture_L [xCb' + xPi][yCb' + yPj – 2] \quad (8\text{-}870)$$

$$sB = recPicture_L [xCb' + xPi][yCb' + yPj – 1] \quad (8\text{-}871)$$

$$sC = recPicture_L [xCb' + xPi][yCb' + yPj] \quad (8\text{-}872)$$

$$sD = recPicture_L [xCb' + xPi][yCb' + yPj + 1] \quad (8\text{-}873)$$

   C. The variables d, abs, sign, t16, clip, d1, and d2 are derrived as follows:

$$d = (sA - (sB << 2) + (sC << 2) – sD) / 8 \quad (8\text{-}874)$$

$$abs = Abs(d) \quad (8\text{-}875)$$

$$sign = Sign(d) \quad (8\text{-}876)$$

$$t16 = Max(0 , ((abs - sT'[xDi][yDj]) << 1)) \quad (8\text{-}877)$$

$$clip = Max(0, (abs - t16)) \quad (8\text{-}878)$$

$$d1 = ((sign)? - clip : clip) \quad (8\text{-}879)$$

$$clip >>= 1 \quad (8\text{-}880)$$

$$d2 = Clip3(-clip, clip, ((sA - sD) / 4)) \quad (8\text{-}881)$$

$$sA -= d2 \quad (8\text{-}882)$$

$$sB += d1 \quad (8\text{-}883)$$

$$sC -= d1 \quad (8\text{-}884)$$

sD += d2 (8-885)

5. The modified luma picture sample array $recPicture_L$ is derived as follows:
   A. If edgeType is equal to EDGE_VER, $recPicture_L$ is derived as follows:

$$recPicture_L[xCb' + xPi - 2][yCb' + yPj] = Clip3(0, (1 << BitDepth_Y) - 1, sA) \quad (8-886)$$

$$recPicture_L[xCb' + xPi - 1][yCb' + yPj] = Clip3(0, (1 << BitDepth_Y) - 1, sB) \quad (8-887)$$

$$recPicture_L[xCb' + xPi][yCb' + yPj] = Clip3(0, (1 << BitDepth_Y) - 1, sC) \quad (8-888)$$

$$recPicture_L[xCb' + xPi + 1][yCb' + yPj] = Clip3(0, (1 << BitDepth_Y) - 1, sD) \quad (8-889)$$

   B. Otherwise (edgeType is equal to EDGE_HOR), $recPicture_L$ is derived as follows:

$$recPicture_L[xCb' + xPi][yCb' + yPj - 2] = Clip3(0, (1 << BitDepth_Y) - 1, sA) \quad (8-890)$$

$$recPicture_L[xCb' + xPi][yCb' + yPj - 1] = Clip3(0, (1 << BitDepth_Y) - 1, sB) \quad (8-891)$$

$$recPicture_L[xCb' + xPi][yCb' + yPj] = Clip3(0, (1 << BitDepth_Y) - 1, sC) \quad (8-892)$$

$$recPicture_L[xCb' + xPi][yCb' + yPj + 2] = Clip3(0, (1 << BitDepth_Y) - 1, sD) \quad (8-893)$$

The filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

For Cb and Cr block, $recPicture_{Cb}$ and $recPicture_{Cr}$ are processed separately.

For vertical edge and horizontal edge, the following two cases apply separately:

- For vertical edge, edgeType is set equal to 0 and the next step applies.
- For horizontal edge, edgeType is set equal to 1 and the next step applies.

The variables xPi, yPj, xDi, yDj, xN, and yN are derived as follows:

- xPi is set equal to i, yPj is set equal to j, xN is set equal to (1 << (log2CbWidth − 1)) − 1, and yN is set equal to (1 << (log2CbHeight − 1)) − 1.
- xDi is set equal to (xPi >> 1), yDj is set equal to (yPj >> 1).

The filtering pixels in the chroma coding block to be filtered are set with the following conditions:

- If edgeType is equal to EDGE_VER, pixels for xPi with i = 0..xN and yPj with j = 0 are set to be filtered.
- Otherwise, if edgeType is equal to EDGE_HOR,
  - If availLR is equal to LR_11 or LR_10, pixels for xPi with i = 0 and yP_j with j = 0..yN are set to be filtered.
  - If availLR is equal to LR_11 or LR_01, pixels for xPi with i = xN + 1 and yP_j with j = 0..yN are set to be filtered.

The filtering process is applied as follows:

1. The sample values p_0 and q_0 are derived as follows:
   A. If edgeType is equal to EDGE_VER, p_0 is set equal to $recPicture_C[xCb'/2 + xPi - 1][yCb'/2 + yPj]$ and q_0 is set equal to $recPicture_C[xCb'/2 + xPi][yCb'/2 + yPj]$.
   B. Otherwise (edgeType is equal to EDGE_HOR), p_0 is set equal to $recPicture_C[xCb'/2 + xPi][yCb'/2 + yPj - 1]$ and q_0 is set equal to $recPicture_C[xCb'/2 + xPi][yCb'/2 + yPj]$.
2. The variable sT[xDi][yDj] and sT'[xDi][yDj] are derived as follows:
   A. Qp'_{Cb} and Qp'_{Cr} values of the coding units which include the chroma coding blocks containing the sample q_0 and p_0.
   B. The variable $bS_C[xDi][yDj]$ is derived from $bS_L[xDi][yDj]$
   C. sT[xDi][yDj] is derived from Table 8-19.

D. sT'[xDi][yDj] is equal to sT[xDi][yDj] << ($BitDepth_C$ − 8).

3. When sT'[xDi][yDj] is greater than 0, the following ordered steps apply:

A. If edgeType is equal to EDGE_VER, the sample values sA, sB, sC, and sD are derived as follows:

$$sA = recPicture_{Cx}[xCb'/2 + xPi - 2][yCb'/2 + yPj] \quad (8\text{-}894)$$

$$sB = recPicture_{Cx}[xCb'/2 + xPi - 1][yCb'/2 + yPj] \quad (8\text{-}895)$$

$$sC = recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj] \quad (8\text{-}896)$$

$$sD = recPicture_{Cx}[xCb'/2 + xPi + 1][yCb'/2 + yPj] \quad (8\text{-}897)$$

B. Otherwise (edgeType is equal to EDGE_HOR), the sample values sA, sB, sC, and sD are derived as follows:

$$sA = recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj - 2] \quad (8\text{-}898)$$

$$sB = recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj - 1] \quad (8\text{-}899)$$

$$sC = recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj] \quad (8\text{-}900)$$

$$sD = recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj + 1] \quad (8\text{-}901)$$

C. The variables d, abs, sign, t16, clip, d1, and d2 are derrived as follows:

$$d = (sA - (sB << 2) + (sC << 2) - sD) / 8 \quad (8\text{-}902)$$

$$abs = Abs(d) \quad (8\text{-}903)$$

$$sign = Sign(d) \quad (8\text{-}904)$$

$$t16 = Max(0 , ((abs - sT'[xDi][yDj]) << 1)) \quad (8\text{-}905)$$

$$clip = Max(0, (abs - t16)) \quad (8\text{-}906)$$

$$d1 = ((sign)? - clip : clip) \quad (8\text{-}907)$$

$$sB += d1 \quad (8\text{-}908)$$

$$sC -= d1 \quad (8\text{-}909)$$

4. The modified chroma picture sample array $recPicture_C$ is derived as follows:

A. If edgeType is equal to EDGE_VER, $recPicture_C$ is derived as follows:

$$recPicture_{Cx}[xCb'/2 + xPi - 1][yCb'/2 + yPj] = Clip3(0, (1 << BitDepth_C) - 1, sB) \quad (8\text{-}910)$$

$$recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj] = Clip3(0, (1 << BitDepth_C) - 1, sC) \quad (8\text{-}911)$$

B. Otherwise (edgeType is equal to EDGE_HOR), $recPicture_C$ is derived as follows:

$$recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj - 1] = Clip3(0, (1 << BitDepth_C) - 1, sB) \quad (8\text{-}912)$$

$$recPicture_{Cx}[xCb'/2 + xPi][yCb'/2 + yPj] = Clip3(0, (1 << BitDepth_C) - 1, sC) \quad (8\text{-}913)$$

Table 8-19 –Derivation of sT[xDi][yDj] from QP and bS_X[xDi][yDj] where X is color component

QP	bS_X[xDi][yDj] =1	bS_X[xDi][yDj] =2	bS_X[xDi][yDj] =3
QP <=17	0	0	0
18	1	0	0
19	1	0	0
20	1	0	0
21	1	0	0
22	1	0	0
23	1	0	0
24	1	0	0
25	1	0	0
26	1	0	0
27	2	1	0
28	2	1	0
29	2	1	0
30	2	1	0
31	2	1	0
32	3	2	1
33	3	2	1
34	3	2	1
35	4	3	2
36	4	3	2
37	4	3	2
38	5	4	3
39	5	4	3
40	6	5	4
41	6	5	4
42	7	6	5
43	8	7	6
44	9	8	7
45	10	9	8
46	11	10	9
47	12	11	10
48	12	11	10
49	12	11	10
50	12	11	10
51	12	11	10

8.8.3 Enhanced deblocking filter process

This process is invoked with sps_addb_flag is set equal to 1.

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 log2CbWidth and log2CbHeight specifying the width and the height of the current luma coding block,

– a reconstructed picture sample arrays prior to deblocking $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$,.

– a luma motion vectors mvL0 and mvL1,

– a reference indices refIdxL0 and refIdxL1,

– index chromaEdgeFlag identifying the deblocking is applied to the chroma component.

Outputs of this process are the modified reconstructed picture sample arrays after deblocking $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$.

Filtering in the horizontal direction (across the vertical block edges) is conducted first. Then filtering in the vertical direction (across the horizontal block edges) is conducted with samples modified by the filtering in the horizontal direction as input. The horizontal and vertical edges in the coding tree blocks of each coding tree unit are processed separately on a coding unit basis. 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. 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 deblocking filter process is applied to all the block edges of a picture, except the edges that are at the boundary of the picture and edges that do not correspond to 8x8 sample grid boundaries of the considered component, for which the deblocking filter process is disabled by slice_deblocking_filter_flag.

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 8-20.

Table 8-20 – Name of association to edgeType

edgeType	Name of edgeType
0 (vertical edge)	EDGE_VER
1 (horizontal edge)	EDGE_HOR

If xCb % 8 != 0, the left vertical edge will not be filtered; if (xCb + 1 << log2CbWidth) % 8 != 0, the right vertical edge will not be filtered; If yCb % 8 != 0, the top horizontal edge will not be filtered; if (yCb + 1 << log2CbHeight) % 8 != 0, the bottom horizontal edge will not be filtered.

The filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps: The variable log2CbSize is set by:

log2CbSize = (edgeType = = EDGE_VERT) ? log2CbWidth : log2CbHeight (8-914)

The variable chromaStyleFilteringFlag is set by:

chromaStyleFilteringFlag = chromaEdgeFlag && (ChromaArrayType != 3).The variables xD_i, yD_j, xN, and yN are derived as follows:

- xDi is set equal to (i << 2), yDj is set equal to (j << 2), xN is set equal to (1 << (log2CbSize − 2)) − 1, and yN is set equal to (1 << (log2CbSize − 2)) − 1.

For xD_i with i = 0..xN and yD_j with j = 0..yN, the following applies:

1. The sample values p_k and q_k with k = 0, 1, 2 are derived as follows:
   A. If edgeType is equal to EDGE_VER, p_k is set equal to recPictureL[xCb + xDi − 1 k][yCb + yDj] and q_k is set equal to recPictureL[xCb + xDi + k][yCb + yDj].
   B. Otherwise (edgeType is equal to EDGE_HOR), p_k is set equal to recPictureL[xCb + xDi][yCb + yDj − 1 − k] and q_k is set equal to recPictureL[xCb + xDi][yCb + yDj + k].
2. The variable bS[xDi][yDj] is derived as follows:
   A. If the sample p_k or q_k is in the luma coding block of a coding unit coded with intra prediction mode and p_k and q_k are from two coding units that are located in different LCU, bS[xDi][yDj] is set equal to 4.
   B. Otherwise, if the sample p_k or q_k is in the luma coding block of a coding unit coded with intra prediction mode, bS[xDi][yDj] is set equal to 3.
   C. Otherwise, if the sample p_k or q_k is in the luma coding block of a coding unit coded with cbf_luma equal to 1, bS[xDi][yDj] is set equal to 2.
   D. Otherwise, if the sample p_0 or q_0 is in the luma coding block of a coding unit coded with ibc prediction mode, bS_L[xDi][yDj] is set equal to 2.
   C. Otherwise, the following is applies:
      i. The variables mv0_l0_x, mv0_l0_y, mv1_l0_x, and mv1_l0_y are derived as follows:

1. If edgeType is equal to EDGE_VER, the following is applies:

mv0_l0_x = mvL0[xCb + xDi][yCb + yDj][0] (8-915)

mv0_l0_y = mvL0[xCb + xDi][yCb + yDj][1] (8-916)

mv0_l1_x = mvL1[xCb + xDi][yCb + yDj][0] (8-917)

mv0_l1_y = mvL1[xCb + xDi][yCb + yDj][1] (8-918)

mv1_l0_x = mvL0[xCb + xDi – 1][yCb + yDj][0] (8-919)

mv1_l0_y = mvL0[xCb + xDi – 1][yCb + yDj][1] (8-920)

mv1_l1_x = mvL1[xCb + xDi – 1][yCb + yDj][0] (8-921)

mv1_l1_y = mvL1[xCb + xDi – 1][yCb + yDj][1] (8-922)

refIdx_0_L0 = ref_idx_l0[xCb + xDi][yCb + yDj] (8-923)

refIdx_0_L1 = ref_idx_l1[xCb + xDi][yCb + yDj] (8-924)

refIdx_1_L0 = ref_idx_l0[xCb + xDi – 1][yCb + yDj] (8-925)

refIdx_1_L1 = ref_idx_l1[xCb + xDi– 1][yCb + yDj] (8-926)

2. Otherwise, if edgeType is equal to EDGE_HOR, the following is applies:

mv0_l0_x = mvL0[xCb + xDi][yCb + yDj][0] (8-927)

mv0_l0_y = mvL0[xCb + xDi][yCb + yDj][1] (8-928)

mv0_l1_x = mvL1[xCb + xDi][yCb + yDj][0] (8-929)

mv0_l1_y = mvL1[xCb + xDi][yCb + yDj][1] (8-930)

mv1_l0_x = mvL0[xCb + xDi][yCb + yDj – 1][0] (8-931)

mv1_l0_y = mvL0[xCb + xDi][yCb + yDj – 1][1] (8-932)

mv1_l1_x = mvL1[xCb + xDi][yCb + yDj – 1][0] (8-933)

mv1_l1_y = mvL1[xCb + xDi][yCb + yDj – 1][1] (8-934)

refIdx_0_L0 = ref_idx_l0[xCb + xDi][yCb + yDj] (8-935)

refIdx_0_L1 = ref_idx_l1[xCb + xDi][yCb + yDj] (8-936)

refIdx_1_L0 = ref_idx_l0[xCb + xDi][yCb + yDj – 1] (8-937)

refIdx_1_L1 = ref_idx_l1[xCb + xDi][yCb + yDj – 1] (8-938)

3. If refIdx_0_L0 is unavailabe, mv0_l0_x and mv0_l0_y are set equal to 0.
4. If refIdx_0_L1 is unavailabe, mv0_l1_x and mv0_l1_y are set equal to 0.
5. If refIdx_1_L0 is unavailabe, mv1_l0_x and mv1_l0_y are set equal to 0.
6. If refIdx_1_L1 is unavailabe, mv1_l1_x and mv1_l1_y are set equal to 0.

ii. If refIdx_0_L0 is equal to refIdx_0_L1 and refIdx_1_L0 is equal to refIdx_1_L1,

```
bS[ xDi ][ yDj ] = ( Abs( mv0_l0_x - mv1_l0_x ) >= 4 ||
                     Abs( mv0_l0_y - mv1_l0_y ) >= 4 ||
```

Abs(mv0_l1_x - mv1_l1_x) >= 4 ||
Abs(mv0_l1_y - mv1_l1_y) >= 4) ? 1 : 0 (8-939)

iii. Otherwise, if refIdx_0_L0 is equal to refIdx_1_L1 and refIdx_0_L1 is equal to refIdx_1_L0,

bS[xDi][yDj] = (Abs(mv0_l0_x - mv1_l1_x) >= 4 ||
Abs(mv0_l0_y - mv1_l1_y) >= 4 ||
Abs(mv0_l1_x - mv1_l0_x) >= 4 ||
Abs(mv0_l1_y - mv1_l0_y) >= 4) ? 1 : 0 (8-940)

D. Otherwise, bS[xDi][yDj] is set equal to 0

3. The thresholds for each block edge is derived as follows:

Let qP_{av} be a variable specifying an average quantization parameter. It is derived as:

$qP_{av} = (qP_p + qP_q + 1) >> 1$

where qP_p and qP_q are the quantization parameters that are used in the CU where sample p_k and q_k are located.

Let indexA be a variable that is used to access the α table (Table 8-21) as well as the t_{C0} table (Table 8-21), which is used in filtering of edges with bS[xDi][yDj] less than 4 as derived in the above section, and let indexB be a variable that is used to access the α table (Table 8-21). The variables indexA and indexB are derived as:

indexA = Clip3(0, 51, qP_{av} + filterOffsetA) (8-941)
indexB = Clip3(0, 51, qP_{av} + filterOffsetB) (8-942)

The variables α′ and β′ depending on the values of indexA and indexB are specified in Table 8-21. Depending on chromaEdgeFlag, the corresponding threshold variables α and β are derived as follows:

$\alpha = \alpha' * (1 << (BitDepth_Y - 8))$ (8-943)
$\beta = \beta' * (1 << (BitDepth_Y - 8))$ (8-944)

The variable filterSamplesFlag is derived by:

filterSamplesFlag = (bS[xDi][yDj] != 0 && Abs($p_0 - q_0$) < α && Abs($p_1 - p_0$) < β && Abs($q_1 - q_0$) < β) (8-945)

Table 8-21 – Derivation of offset dependent threshold variables α′ and β′ from indexA and indexB

indexA (for α′) or indexB (for β′)																										
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
α′	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	5	6	7	8	9	10	12	13
β′	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	3	3	3	3	4	4	4
indexA (for α′) or indexB (for β′)																										
	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
α′	15	17	20	22	25	28	32	36	40	45	50	56	63	71	80	90	101	113	127	144	162	182	203	226	255	255
β′	6	6	7	7	8	8	9	9	10	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17	18	18

Table 8-22 – Value of variable t'_{C0} as a function of indexA and bS

	indexA																								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
bS = 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
bS = 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
bS = 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

	indexA																								
	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
bS = 1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	4	4	4	5	6	6	7	8	9	10	11	13
bS = 2	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5	5	6	7	8	8	10	11	12	13	15	17
bS = 3	1	2	2	2	2	3	3	3	4	4	4	5	6	6	7	8	9	10	11	13	14	16	18	20	23	25

4. If filterSamplesFlag is equal to 0, the filtered result samples p′i and q'_i (i = 0..2) are replaced by the corresponding input samples p_i and q_i:

$$p'i = pi, \text{ for } i = 0..2 \quad (8\text{-}946)$$

$$q'_i = q_i, \text{ for } i = 0..2 \quad (8\text{-}947)$$

Otherwise, if bS[xDi][yDj] < 4,

Depending on the values of indexA and bS, the variable t'_{C0} is specified in Table 8-22. Depending on chromaEdgeFlag, the corresponding threshold variable t_{C0} is derived as follows:

– If chromaEdgeFlag is equal to 0,

$$t_{C0} = t'_{C0} * (1 << (BitDepth_Y - 9)) \quad (8\text{-}948)$$

– Otherwise (chromaEdgeFlag is equal to 1),

$$t_{C0} = t'_{C0} * (1 << (BitDepth_C - 9))) \quad (8\text{-}949)$$

The threshold variables a_p and a_q are derived by:

$$a_p = Abs(p_2 - p_0) \quad (8\text{-}950)$$

$$a_q = Abs(q_2 - q_0) \quad (8\text{-}951)$$

The threshold variable t_C is determined as follows:

– If chromaStyleFilteringFlag is equal to 0,

$$tcIncP = ((a_p < \beta) ? 1 : 0) * (1 << (BitDepth_Y - 9))$$

$$tcIncQ = ((a_q < \beta) ? 1 : 0) * (1 << (BitDepth_Y - 9))$$

$$t_C = t_{C0} + tcIncP + tcIncQ \quad (8\text{-}832)$$

– Otherwise (chromaStyleFilteringFlag is equal to 1),

$$t_C = t_{C0} + (1 << (BitDepth_Y - 9))) \quad (8\text{-}952)$$

Let Clip1( ) be a function that is replaced by $Clip1_Y( )$ when chromaEdgeFlag is equal to 0 and by $Clip1_C( )$ when chromaEdgeFlag is equal to 1.

The filtered result samples p'_0 and q'_0 are derived by:

$$\Delta = Clip3(-t_C, t_C, ((((q_0 - p_0) << 2) + (p_1 - q_1) + 4) >> 3)) \quad (8\text{-}953)$$

$$p'_0 = Clip1(p_0 + \Delta) \quad (8\text{-}954)$$

$$q'_0 = Clip1(q_0 - \Delta) \quad (8\text{-}955)$$

The filtered result sample p'_1 is derived as follows:

– If chromaStyleFilteringFlag is equal to 0 and a_p is less than β,

$$p'_1 = p_1 + Clip3(-t_{C0}, t_{C0}, (p_2 + ((p_0 + q_0 + 1) >> 1) - (p_1 << 1)) >> 1) \quad (8\text{-}956)$$

– Otherwise (chromaStyleFilteringFlag is equal to 1 or a_p is greater than or equal to β),

$$p'_1 = p_1 \quad (8\text{-}957)$$

The filtered result sample q'_1 is derived as follows:

– If chromaStyleFilteringFlag is equal to 0 and a_q is less than β,

$$q'_1 = q_1 + \text{Clip3}(-t_{C0}, t_{C0}, (q_2 + ((p_0 + q_0 + 1) >> 1) - (q_1 << 1)) >> 1) \quad (8\text{-}958)$$

– Otherwise (chromaStyleFilteringFlag is equal to 1 or a_q is greater than or equal to β),

$$q'_1 = q_1 \quad (8\text{-}959)$$

The filtered result samples p'_2 and q'_2 are always set equal to the input samples p_2 and q_2:

$$p'_2 = p_2 \quad (8\text{-}960)$$

$$q'_2 = q_2 \quad (8\text{-}961)$$

Otherwise, if bS[xDi][yDj] = 4,

Let a_p and a_q be two threshold variables as specified in Equations 8-950 and 8-951, respectively.

The filtered result samples p'_i (i = 0..2) are derived as follows:

– If chromaStyleFilteringFlag is equal to 0 and the following condition holds,

$$a_p < \beta \;\&\&\; \text{Abs}(p_0 - q_0) < ((\alpha >> 2) + 2) \quad (8\text{-}962)$$

then the variables p'_0, p'_1, and p'_2 are derived by:

$$p'_0 = (p_2 + 2*p_1 + 2*p_0 + 2*q_0 + q_1 + 4) >> 3 \quad (8\text{-}963)$$

$$p'_1 = (p_2 + p_1 + p_0 + q_0 + 2) >> 2 \quad (8\text{-}478)$$

$$p'_2 = (2*p_3 + 3*p_2 + p_1 + p_0 + q_0 + 4) >> 3 \quad (8\text{-}964)$$

– Otherwise (chromaStyleFilteringFlag is equal to 1 or the condition in Equation 8-962 does not hold), the variables p'_0, p'_1, and p'_2 are derived by:

$$p'_0 = (2*p_1 + p_0 + q_1 + 2) >> 2 \quad (8\text{-}965)$$

$$p'_1 = p_1 \quad (8\text{-}966)$$

$$p'_2 = p_2 \quad (8\text{-}967)$$

The filtered result samples q'_i (i = 0..2) are derived as follows:

– If chromaStyleFilteringFlag is equal to 0 and the following condition holds,

$$a_q < \beta \;\&\&\; \text{Abs}(p_0 - q_0) < ((\alpha >> 2) + 2) \quad (8\text{-}968)$$

then the variables q'_0, q'_1, and q'_2 are derived by

$$q'_0 = (p_1 + 2*p_0 + 2*q_0 + 2*q_1 + q_2 + 4) >> 3 \quad (8\text{-}969)$$

$$q'_1 = (p_0 + q_0 + q_1 + q_2 + 2) >> 2 \quad (8\text{-}485)$$

$$q'_2 = (2*q_3 + 3*q_2 + q_1 + q_0 + p_0 + 4) >> 3 \quad (8\text{-}970)$$

– Otherwise (chromaStyleFilteringFlag is equal to 1 or the condition in Equation 8-968 does not hold), the variables q'_0, q'_1, and q'_2 are derived by:

$$q'_0 = (2*q_1 + q_0 + p_1 + 2) >> 2 \quad (8\text{-}971)$$

$$q'_1 = q_1 \quad (8\text{-}972)$$

$$q'_2 = q_2 \quad (8\text{-}973)$$

5. The filtered reconstructed sample p'_k and q'_k are used to replace the reconstructed samples in the reconstructed picture recPictureL in the corresponding sample positions.

8.8.4 Adaptive Loop Filter

Inputs of this process are the reconstructed picture sample arrays prior to adaptive loop filter recPictureL, recPictureCb and recPictureCr for luma and two chroma components respecitively.

Outputs of this process are the modified reconstructed picture sample arrays after adaptive loop filter alfPictureL, alfPictureCb and alfPictureCr for luma and two chroma components respecitively.

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 slice_alf_enabled_flag is equal to 1, for every coding tree unit with luma coding tree block location (rx, ry), where rx = 0..PicWidthInCtbsY − 1 and ry = 0..PicHeightInCtbsY − 1, the following applies:

- When alf_ctb_flag[rx][ry] is equal to 1, the coding tree block filtering process for luma samples as specified in clause 8.8.4.1 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_chroma_idc is equal to 1 or 3, the coding tree block filtering process for chroma samples as specified in clause 8.8.4.3 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_chroma_idc is equal to 2 or 3, the coding tree block filtering process for chroma samples as specified in clause 8.8.4.3 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.

8.8.4.1 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 8.8.4.2 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.. NumAlfCoefs – 2:

f[j] = AlfCoeffL[filtIdx[x][y]][j] (8-974)

The luma filter coefficients filterCoeff are derived depending on transposeIdx[x][y] as follows:

– If transposeIndex[x][y] is equal to 1, the following applies:

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] } (8-975)

– Otherwise, if transposeIndex[x][y] is equal to 2, the following applies:

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] } (8-976)

– Otherwise, if transposeIndex[x][y] is equal to 3, the following applies:

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] } (8-977)

– Otherwise, the following applies:

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] } (8-978)

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) (8-979)

vy = Clip3(0, pic_height_in_luma_samples − 1, yCtb + y) (8-980)

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 ] ) +
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 ]
```

(8-981)

$$sum = (sum + 256) >> 9 \quad (8\text{-}982)$$

The modified filtered reconstructed luma picture sample alfPictureL[xCtb + x][yCtb + y] is derived as follows:

$$alfPictureL[xCtb + x][yCtb + y] = Clip3(0, (1 << BitDepthY) − 1, sum) \quad (8\text{-}983)$$

8.8.4.2 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 recPictureL of luma samples are derived as follows:

$$hx = Clip3(0, pic\_width\_in\_luma\_samples − 1, x) \quad (8\text{-}984)$$

$$vy = Clip3(0, pic\_height\_in\_luma\_samples − 1, y) \quad (8\text{-}985)$$

The classification filter index array filtIdx and the transpose index array transposeIdx are derived by the following ordered steps:

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:

$$filtH[x][y] = Abs((recPictureL[hxCtb+x, vyCtb+y] << 1) − recPictureL[hxCtb+x−1, vyCtb+y] − recPictureL[hxCtb+x+1, vyCtb+y]) \quad (8\text{-}986)$$

$$filtV[x][y] = Abs((recPictureL[hxCtb+x, vyCtb+y] << 1) − recPictureL[hxCtb+x, vyCtb+y−1] − recPictureL[hxCtb+x, vyCtb+y+1]) \quad (8\text{-}987)$$

$$filtD0[x][y] = Abs((recPictureL[hxCtb+x, vyCtb+y] << 1) − recPictureL[hxCtb+x−1, vyCtb+y−1] − recPictureL[hxCtb+x+1, vyCtb+y+1]) \quad (8\text{-}988)$$

$$filtD1[x][y] = Abs((recPictureL[hxCtb+x, vyCtb+y] << 1) − recPictureL[hxCtb+x+1, vyCtb+y−1] − recPictureL[hxCtb+x−1, vyCtb+y+1]) \quad (8\text{-}989)$$

– Otherwise, filtH[x][y], filtV[x][y], filtD0[x][y] and filtD1[x][y] are set equal to 0.

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:

$$sumH[x][y] = \Sigma i \Sigma j \; filtH[(x << 2) + i][(y << 2) + j] \text{ with } i, j = −2..5 \quad (8\text{-}990)$$

$$sumV[x][y] = \Sigma i \Sigma j \; filtV[(x << 2) + i][(y << 2) + j] \text{ with } i, j = −2..5 \quad (8\text{-}991)$$

sumD0[x][y] = ΣiΣj filtD0[(x << 2) + i][(y << 2) + j] with i, j = −2..5 (8-992)

sumD1[x][y] = ΣiΣj filtD1[(x << 2) + i][(y << 2) + j] with i, j = −2..5 (8-993)

sumOfHV[x][y] = sumH[x][y] + sumV[x][y] (8-994)

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] (8-995)

hv0 = sumH[x >> 2][y >> 2] (8-996)

dirHV = 1 (8-997)

– Otherwise, the following applies:

hv1 = sumH[x >> 2][y >> 2] (8-998)

hv0 = sumV[x >> 2][y >> 2] (8-999)

dirHV = 3 (8-1000)

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] (8-1001)

d0 = sumD1[x >> 2][y >> 2] (8-1002)

dirD = 0 (8-1003)

– Otherwise, the following applies:

d1 = sumD1[x >> 2][y >> 2] (8-1004)

d0 = sumD0[x >> 2][y >> 2] (8-1005)

dirD = 2 (8-1006)

The variables hvd1, hvd0, are derived as follows:

hvd1 = (d1 * hv0 > hv1 * d0) ? d1 : hv1 (8-1007)

hvd0 = (d1 * hv0 > hv1 * d0) ? d0 : hv0 (8-1008)

The variables dirS[x][y], dir1[x][y] and dir2[x][y] derived as follows:

dir1[x][y] = (d1 * hv0 > hv1 * d0) ? dirD : dirHV (8-1009)

dir2[x][y] = (d1 * hv0 > hv1 * d0) ? dirHV : dirD (8-1010)

dirS[x][y] = (hvd1 > 2 * hvd0) ? 1 : ((hvd1 * 2 > 9 * hvd0) ? 2 : 0) (8-1011)

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 } (8-1012)

avgVar[x][y] = varTab[Clip3(0, 15, (sumOfHV[x >> 2][y >> 2] * 64) >> (3 + BitDepthY))] (8-1013)

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 (8-1014)

8.8.4.3 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 (8-1015)

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) (8-1016)

vy = Clip3(0, pic_height_in_luma_samples / SubHeightC − 1, yCtbC + y) (8-1017)

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 ] ) +
    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 ]
```

(8-1018)

sum = (sum + 256) >> 9 (8-1019)

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) (8-1020)

9 Parsing process

9.1 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) (see clause 9.2), or ae(v) (see clause 9.3).

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

9.2.1 General

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

$$codeNum = 2^{leadingZeroBits} - 1 + read\_bits(leadingZeroBits) \quad (9\text{-}2)$$

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 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 x_i in Table 9-1, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each x_i 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 x_0	1..2
0 0 1 x_1 x_0	3..6
0 0 0 1 x_2 x_1 x_0	7..14
0 0 0 0 1 x_3 x_2 x_1 x_0	15..30
0 0 0 0 0 1 x_4 x_3 x_2 x_1 x_0	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.

9.2.2 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)

9.3 CABAC parsing process for slice data

9.3.1 General

This process is invoked when parsing syntax elements with descriptor ae(v) in clauses 7.3.6.1 through 7.3.6.6.

Inputs to this process are a request for a value of a syntax element and values of prior parsed syntax elements.

The output of this process is the value of the syntax element.

The initialization process as specified in clause 9.3.2 is invoked when starting the parsing of one or more of the following:

1. The slice data syntax specified in clause 7.3.6.1,
2. The CTU syntax specified in clause 7.3.6.2 and the CTU is the first CTU in 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 clause 9.3.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in clause 9.3.4.

9.3.2 Initialization process

9.3.2.1 General

The outputs of this process are initialized CABAC internal variables.

The context variables of the arithmetic decoding engine are initialized as follows:

The initialization process for context variables is invoked as specified in clause

– If the CTU is the first CTU in a tile, the initialization process for context variables is invoked as specified in clause 9.3.2.2.

– Otherwise, the initialization process for context variables is invoked as specified in clause 9.3.2.2.

The initialization process for the arithmetic decoding engine is invoked as specified in clause 9.3.2.3.

9.3.2.2 Initialization process for context variables

Outputs of this process are the initialized CABAC context variables indexed by ctxTable and ctxIdx.

When sps_cm_init_flag is equal to 0, the following applies:

Table 9-5 to Table 9-20 contain the values of the 10-bit variable initValue used in the initialization of context variables that are assigned to all syntax elements in clause 7.3.6.1 through 7.3.6.6, except end_of_tile_one bit.

For each context variable, from the 14-bit table entry initValue, the two variables valState and valMps are initialized as follows:

```
valState = initValue
valMps = 0                                                    (9-3)
```

When sps_cm_init_flag is equal to 1, the following applies:

Table 9-21 to Table 9-59 contain the values of the 10-bit variable initValue used in the initialization of context variables that are assigned to all syntax elements in clause 7.3.6.1 through 7.3.6.6, except end_of_tile_one bit.

For each context variable, from the 10-bit table entry initValue, the two 16-bit signed variables valSlope and valOffset are derived as follows:

```
valSlope = (initValue & 0b1110) <<4

valSlope = (initValue & 0b1)? -valSlope : valSlope

tmp = *(ctx_init_model) >> 4

valOffset  = (tmp & 0b111110) << 7

valOffset = (tmp & 0b1)? -valOffset : valOffset

valOffset += 4096                                             (9-4)
```

The two values assinged to valState and valMps for initilization are derived from slice_qp. Given the variable valSlope and valOffset, the initialization is specified as follows:

```
preValState = Clip3( 1, 511, valSlope * slice_qp + valOffset )
valMps = prevalState > 256 ? 0 : 1
valState = valMps ? preValState : ( 512 – preValState )       (9-5)
```

In Table 9-4, the ctxIdx for which initialization is needed for each of the two 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. If slice_type is P or B, the value of initType is set equal to 1. Otherwise, the value of initType is set equal to 0.

Table 9-4 – Association of ctxIdx and syntax elements for each initializationType in the initialization process

Syntax structure	Syntax element	sps_cm_init_flag = 0			sps_cm_init_flag = 1		
		ctxTable	initType		ctxTable	initType	
			0	1		0	1
split_unit( )	split_cu_flag[ ][ ]	Table 9-5	0	1	n/a	n/a	n/a
	btt_split_flag[ ][ ]	n/a	n/a	n/a	Table 9-21	0..14	15..29
	btt_split_dir[ ][ ]	n/a	n/a	n/a	Table 9-22	0..4	5..9
	btt_split_type[ ][ ]	n/a	n/a	n/a	Table 9-23	0	1
	split_unit_coding_order_flag[ ][ ]	n/a	n/a	n/a	Table 9-24	0..11	12..23
coding_unit( )	cu_skip_flag[ ][ ]	Table 9-6	0	1	Table 9-25	0..1	2..3
	mvp_idx_l0[ ][ ] mvp_idx_l1[ ][ ]	Table 9-7	0..2	3..5	n/a	n/a	n/a
	mvp_idx[ ][ ]	n/a	n/a	n/a	Table 9-26	0..4	5..9
	mmvd_flag[ ][ ]	n/a	n/a	n/a	Table 9-27	0	1
	mmvd_group_idx[ ][ ]	n/a	n/a	n/a	Table 9-28	0..1	2..3
	mmvd_merge_idx[ ][ ]	n/a	n/a	n/a	Table 9-29	0..2	3..5

	mmvd_distance_idx[ ][ ]	n/a	n/a	n/a	Table 9-30	0..6	7..13
	mmvd_direction_idx[ ][ ]	n/a	n/a	n/a	Table 9-31	0..1	2..3
	affine_flag[ ][ ]	n/a	n/a	n/a	Table 9-32	0..1	2..3
	affine_merge_idx[ ][ ]	n/a	n/a	n/a	Table 9-33	0..4	5..9
	affine_mode[ ][ ]	n/a	n/a	n/a	Table 9-34	0	1
	pred_mode_flag[ ]	Table 9-8	0	1	Table 9-35	0..2	3..5
	intra_pred_mode[ ][ ]	Table 9-9	0..1	2..3	n/a	n/a	n/a
	intra_luma_pred_mpm_flag[ ][ ]	n/a	n/a	n/a	Table 9-36	0	1
	intra_luma_pred_mpm_idx[ ][ ]	n/a	n/a	n/a	Table 9-37	0	1
	intra_chroma_pred_mode[ ][ ]	n/a	n/a	n/a	Table 9-38	0	1
	ibc_flag[ ][ ]	n/a	0	1	Table 9-39	0..1	2..3
	amvr_idx[ ][ ]	n/a	n/a	n/a	Table 9-40	0..3	4..7
	direct_mode_flag[ ][ ]	Table 9-10	0	1	n/a	n/a	n/a
	inter_pred_idc[ ][ ]	Table 9-11	0..1	2..3	Table 9-41	0..1	2..3
	merge_mode_flag[ ][ ]	n/a	n/a	n/a	Table 9-42	0	1
	bi_pred_idx[ ][ ]	n/a	n/a	n/a	Table 9-43	0..1	2..3
	ref_idx_l0[ ][ ] ref_idx_l1[ ][ ]	Table 9-12	0..1	2..3	Table 9-44	0..1	2..3
	abs_mvd_l0[ ][ ] abs_mvd_l1[ ][ ]	Table 9-13	0	1	Table 9-45	0	1
	cbf_all	Table 9-14	0	1	Table 9-46	0	1
transform_unit( )	cbf_luma	Table 9-15	0	1	Table 9-47	0	1
	cbf_cb	Table 9-16	0	1	Table 9-48	0	1
	cbf_cr	Table 9-17	0	1	Table 9-49	0	1
	ats_cu_intra_flag[ ][ ]	n/a	0..1	2..3	Table 9-50	0..7	8..15
	ats_hor_mode[ ][ ]	n/a	0..1	2..3	Table 9-51	0..1	2..3
	ats_ver_mode[ ][ ]	n/a	0..1	2..3	Table 9-52	0..1	2..3
	ats_cu_inter_flag[][]	n/a	n/a	n/a	Table 9-53	n/a	0..1
	ats_cu_inter_quad_flag[][]	n/a	n/a	n/a	Table 9-54	n/a	0
	ats_cu_inter_horizontal_flag[][]	n/a	n/a	n/a	Table 9-55	n/a	0..2
	ats_cu_inter_pos_flag[][]	n/a	n/a	n/a	Table 9-56	n/a	0
residual_coding_baseline( )	coeff_zero_run	Table 9-18	0..23	24..47	Table 9-57	0..23	24..47
	coeff_abs_level_minus1	Table 9-19	0..23	24..47	Table 9-58	0..23	24..47
	coeff_last_flag	Table 9-20	0..1	2..3	Table 9-59	0..1	2..3
residual_coding_main( )	last_sig_coeff_x_prefix	n/a	n/a	n/a	Table 9-60	0..23	24..47
	last_sig_coeff_y_prefix	n/a	n/a	n/a	Table 9-61	0..23	24..47
	sig_coeff_flag	n/a	n/a	n/a	Table 9-62	0..1	2..3
	coeff_abs_level_greaterA_flag	n/a	n/a	n/a	Table 9-63	0..23	24..47
	coeff_abs_level_greaterB_flag	n/a	n/a	n/a	Table 9-63	0..23	24..47

Table 9-5 – Values of initValue for ctxIdx of split_cu_flag

Initialization variable	ctxIdx of split_cu_flag	
	0	1
initValue	256	256

Table 9-6 – Values of initValue for ctxIdx of cu_skip_flag

Initialization variable	ctxIdx of cu_skip_flag	
	0	1
initValue	256	256

Table 9-7 – Values of initValue for ctxIdx of mvp_idx_l0 and mvp_idx_l1

Initialization variable	ctxIdx of mvp_idx_l0 and mvp_idx_l1					
	0	1	2	3	4	5
initValue	256	256	256	256	256	256

Table 9-8 – Values of initValue for ctxIdx of pred_mode_flag

Initialization variable	ctxIdx of pred_mode_flag	
	0	1
initValue	256	256

Table 9-9 – Values of initValue for ctxIdx of intra_pred_mode

Initialization variable	ctxIdx of intra_pred_mode			
	0	1	2	3
initValue	256	256	256	256

Table 9-10 – Values of initValue for ctxIdx of direct_mode_flag

Initialization variable	ctxIdx of direct_mode_flag	
	0	1
initValue	256	256

Table 9-11 – Values of initValue for ctxIdx of inter_pred_idc

Initialization variable	ctxIdx of inter_pred_idc			
	0	1	2	3
initValue	256	256	256	256

Table 9-12 – Values of initValue for ctxIdx of ref_idx_l0 and ref_idx_l1

Initialization variable	ctxIdx of ref_idx_l0 and ref_idx_l1			
	0	1	2	3
initValue	256	256	256	256

Table 9-13 – Values of initValue for ctxIdx of abs_mvd_l0 and abs_mvd_l1

Initialization variable	ctxIdx of abs_mvd_l0 and abs_mvd_l1	
	0	1
initValue	256	256

Table 9-14 – Values of initValue for ctxIdx of cbf_all

Initialization variable	ctxIdx of cbf_all	
	0	1
initValue	256	256

Table 9-15 – Values of initValue for ctxIdx of cbf_luma

Initialization variable	ctxIdx of cbf_luma	
	0	1
initValue	256	256

Table 9-16 – Values of initValue for ctxIdx of cbf_cb

Initialization variable	ctxIdx of cbf_cb	
	0	1
initValue	256	256

Table 9-17 – Values of initValue for ctxIdx of cbf_cr

Initialization variable	ctxIdx of cbf_cr	
	0	1
initValue	256	256

Table 9-18 – Values of initValue for ctxIdx of coeff_zero_run

Initialization variable	ctxIdx of coeff_zero_run															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256

Table 9-19 – Values of initValue for ctxIdx of coeff_abs_level_minus1

Initialization variable	ctxIdx of coeff_abs_level_minus1															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256	256

Table 9-20 – Values of initValue for ctxIdx of coeff_last_flag

Initialization variable	ctxIdx of coeff_last_flag			
	0	**1**	**2**	**3**
initValue	256	256	256	256

Table 9-21 – Values of initValue for ctxIdx of btt_split_flag

Initialization variable	ctxIdx of btt_split_flag							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	145	560	528	308	594	560	180	500
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	626	84	406	662	320	36	340	536
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	726	594	66	338	528	258	404	464
	24	**25**	**26**	**27**	**28**	**29**		
initValue	98	342	370	384	256	65		

Table 9-22 – Values of initValue for ctxIdx of btt_split_dir

Initialization variable	ctxIdx of btt_split_dir				
	0	**1**	**2**	**3**	**4**
initValue	0	417	389	99	0
	5	**6**	**7**	**8**	**9**
initValue	0	128	81	49	0

Table 9-23 – Values of initValue for ctxIdx of btt_split_type

Initialization variable	ctxIdx of btt_split_type	
	0	**1**
initValue	257	225

Table 9-24 – Values of initValue for ctxIdx of split_unit_coding_order_flag

Initialization variable	ctxIdx of split_unit_coding_order_flag							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	0	0	0	0	0	0	545	0
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	481	515	0	32	0	0	0	0
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	0	0	0	0	557	0	481	2
	24	**25**	**26**	**27**				
initValue	0	97	0	0				

Table 9-25 – Values of initValue for ctxIdx of cu_skip_flag (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of cu_skip_flag			
	0	**1**	**2**	**3**
initValue	0	0	711	233

Table 9-26 – Values of initValue for ctxIdx of mvp_idx

Initialization variable	ctxIdx of mvp_idx				
	0	**1**	**2**	**3**	**4**
initValue	0	0	0	496	496
	5	**6**	**7**	**8**	**9**
initValue	18	128	146	37	69

Table 9-27 – Values of initValue for ctxIdx of mmvd_flag

Initialization variable	ctxIdx of mmvd_flag	
	0	**1**
initValue	0	194

Table 9-28 – Values of initValue for ctxIdx of mmvd_group_idx

Initialization variable	ctxIdx of mmvd_group_idx			
	0	**1**	**2**	**3**
initValue	0	0	453	48

Table 9-29 – Values of initValue for ctxIdx of mmvd_merge_idx

Initialization variable	ctxIdx of mmvd_merge_idx					
	0	**1**	**2**	**3**	**4**	**5**
initValue	0	0	0	49	129	82

Table 9-30 – Values of initValue for ctxIdx of mmvd_distance_idx

Initialization variable	ctxIdx of mmvd_distance_idx						
	0	**1**	**2**	**3**	**4**	**5**	**6**
initValue	0	0	0	0	0	0	0
	7	**8**	**9**	**10**	**11**	**12**	**13**
initValue	179	5	133	131	227	64	128

Table 9-31 – Values of initValue for ctxIdx of mmvd_direction_idx

Initialization variable	ctxIdx of mmvd_direction_idx			
	0	**1**	**2**	**3**
initValue	0	0	161	33

Table 9-32 – Values of initValue for ctxIdx of affine_flag

Initialization variable	ctxIdx of affine_flag			
	0	**1**	**2**	**3**
initValue	0	0	320	210

Table 9-33 – Values of initValue for ctxIdx of affine_merge_idx

Initialization variable	ctxIdx of affine_merge_idx				
	0	**1**	**2**	**3**	**4**
initValue	0	0	0	0	0
	5	**6**	**7**	**8**	**9**
initValue	193	129	32	323	0

Table 9-34 – Values of initValue for ctxIdx of affine_mode

Initialization variable	ctxIdx of affine_mode	
	0	**1**
initValue	0	225

Table 9-35 – Values of initValue for ctxIdx of pred_mode_flag (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of pred_mode_flag					
	0	**1**	**2**	**3**	**4**	**5**
initValue	64	0	0	481	16	368

Table 9-36 – Values of initValue for ctxIdx of intra_luma_pred_mpm_flag

Initialization variable	ctxIdx of intra_luma_pred_mpm_flag	
	0	**1**
initValue	263	225

Table 9-37 – Values of initValue for ctxIdx of intra_luma_pred_mpm_idx

Initialization variable	ctxIdx of intra_luma_pred_mpm_idx	
	0	**1**
initValue	436	724

Table 9-38 – Values of initValue for ctxIdx of intra_chroma_pred_mode

Initialization variable	ctxIdx of intra_chroma_pred_mode	
	0	**1**
initValue	465	560

Table 9-39 – Values of initValue for ctxIdx of ibc_flag

Initialization variable	ctxIdx of ibc_flag			
	0	**1**	**2**	**3**
initValue	0	0	711	233

Table 9-40 – Values of initValue for ctxIdx of amvr_idx

Initialization variable	ctxIdx of amvr_idx			
	0	**1**	**2**	**3**
initValue	0	0	0	496
	4	**5**	**6**	**7**
initValue	773	101	421	199

Table 9-41 – Values of initValue for ctxIdx of inter_pred_idc (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of inter_pred_idc			
	0	**1**	**2**	**3**
initValue	0	0	242	80

Table 9-42 – Values of initValue for ctxIdx of merge_mode_flag

Initialization variable	ctxIdx of merge_mode_flag	
	0	**1**
initValue	0	464

Table 9-43 – Values of initValue for ctxIdx of bi_pred_idx

Initialization variable	ctxIdx of bi_pred_idx			
	0	**1**	**2**	**3**
initValue	0	0	49	17

Table 9-44 – Values of initValue for ctxIdx of ref_idx_l0 and ref_idx_l1 (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of ref_idx_l0 and ref_idx_l1			
	0	**1**	**2**	**3**
initValue	0	0	288	0

Table 9-45 – Values of initValue for ctxIdx of abs_mvd_l0 and abs_mvd_l1 (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of abs_mvd_l0 and abs_mvd_l1	
	0	**1**
initValue	0	18

Table 9-46 – Values of initValue for ctxIdx of cbf_all (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of cbf_all	
	0	**1**
initValue	0	794

Table 9-47 – Values of initValue for ctxIdx of cbf_luma (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of cbf_luma	
	0	**1**
initValue	664	368

Table 9-48 – Values of initValue for ctxIdx of cbf_cb (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of cbf_cb	
	0	**1**
initValue	384	416

Table 9-49 – Values of initValue for ctxIdx of cbf_cr (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of cbf_cr	
	0	**1**
initValue	320	288

Table 9-50 – Values of initValue for ctxIdx of ats_cu_intra_flag (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of ats_cu_intra_flag							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	999	514	452	546	1001	992	960	960
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
nitValue	1003	292	354	544	999	960	928	960

Table 9-51 – Values of initValue for ctxIdx of ats_hor_mode (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of ats_hor_mode	
	0	**1**
initValue	512	993

Table 9-52 – Values of initValue for ctxIdx of ats_ver_mode (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of ats_ver_mode	
	0	**1**
initValue	512	929

Table 9-53 – Values of initValue for ctxIdx of ats_cu_inter_flag

Initialization variable	ctxIdx of ats_inter_flag	
	0	**1**
initValue	0	0

Table 9-54 – Values of initValue for ctxIdx of ats_cu_inter_quad_flag

Initialization variable	ctxIdx of ats_cu_inter_quad_flag
	0
initValue	0

Table 9-55 – Values of initValue for ctxIdx of ats_cu_inter_horizontal_flag

Initialization variable	ctxIdx of ats_cu_inter_horizontal_flag		
	0	**1**	**2**
initValue	0	0	0

Table 9-56 – Values of initValue for ctxIdx of ats_cu_inter_pos_flag

Initialization variable	ctxIdx of ats_cu_inter_pos_flag
	0
initValue	0

Table 9-57 – Values of initValue for ctxIdx of coeff_zero_run (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of coeff_zero_run							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	48	112	128	0	321	82	419	160
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	385	323	353	129	225	193	387	389
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	453	227	453	161	421	161	481	225
	24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	129	178	453	97	583	259	517	259
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**
initValue	453	227	871	355	291	227	195	97
	40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	161	65	97	33	65	1	1003	227

Table 9-58 – Values of initValue for ctxIdx of coeff_abs_level_minus1 (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of coeff_abs_level_minus1							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	416	98	128	66	32	82	17	48
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	272	112	52	50	448	419	385	355
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	161	225	82	97	210	0	416	224
	24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	805	775	775	581	355	389	65	195
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**
initValue	48	33	224	225	775	227	355	161
	40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	129	97	33	65	16	1	841	355

Table 9-59 – Values of initValue for ctxIdx of coeff_last_flag (sps_cm_init_flag = 1)

Initialization variable	ctxIdx of coeff_last_flag			
	0	**1**	**2**	**3**
initValue	421	337	33	790

Table 9-60 – Values of initValue for ctxIdx of last_sig_coeff_x_prefix (sps_cm_init_flag = 1)

Initialization variable	last_sig_coeff_x_prefix							
	0	1	2	3	4	5	6	7
initValue	890	862	162	728	700	100	338	374
	8	9	10	11	12	13	14	15
initValue	474	162	176	178	342	602	128	466
	16	17	18	19	20	21	22	23
initValue	306	632	502	730	163	304	468	404
b	24	25	26	27	28	29	30	31
initValue	48	192	323	451	274	66	547	340
	32	33	34	35	36	37	38	39
initValue	256	416	242	180	162	162	144	244
	40	41	42	43	44	45	46	47
initValue	212	276	160	99	242	340	436	760
	48	49	50	51	52	53	54	55
initValue	195	131	436	306	80	224	419	547

Table 9-61 – Values of initValue for ctxIdx of last_sig_coeff_y_prefix (sps_cm_init_flag = 1)

Initialization variable	last_sig_coeff_y_prefix							
	0	1	2	3	4	5	6	7
initValue	1020	926	4	436	830	86	500	666
	8	9	10	11	12	13	14	15
initValue	636	320	272	470	504	830	615	596
	16	17	18	19	20	21	22	23
initValue	306	600	404	828	487	336	696	502
	24	25	26	27	28	29	30	31
initValue	163	128	52	288	306	180	288	0
	32	33	34	35	36	37	38	39
initValue	84	194	48	212	52	451	99	146
	40	41	42	43	44	45	46	47
initValue	212	342	743	325	210	308	242	890
	48	49	50	51	52	53	54	55
initValue	421	357	566	566	195	288	98	483

Table 9-62 – Values of initValue for ctxIdx of sig_coeff_flag (sps_cm_init_flag = 1)

Initialization variable	sig_coeff_flag							
	0	1	2	3	4	5	6	7
initValue	387	340	304	227	457	133	208	792
	8	9	10	11	12	13	14	15
initValue	419	553	391	112	534	645	83	402
	16	17	18	19	20	21	22	23
initValue	357	67	112	338	532	711	259	131
b	24	25	26	27	28	29	30	31
initValue	112	698	416	338	304	96	0	372
	32	33	34	35	36	37	38	39
initValue	468	532	352	128	64	210	338	519
	40	41	42	43	44	45	46	47
initValue	35	147	227	131	48	274	630	66
	48	49	50	51	52	53	54	55
initValue	274	240	146	144	35	370	790	483
	56	57	58	59	60	61	62	63
initValue	325	163	112	662	224	3	240	0
	64	65	66	67	68	69	70	71
initValue	274	242	338	434	613	32	32	144
	72	73	74	75	76	77	78	79
initValue	922	387	67	434	195	210	274	338
	80	81	82	83	84	85	86	87
initValue	434	451	96	114	242	208	453	99
	88	89	90	91	92	93	Na	Na
initValue	466	32	80	178	306	726	Naa	Na

Table 9-63 – Values of initValue for ctxIdx of coeff_abs_level_greaterX_flag (sps_cm_init_flag = 1)

Initialization variable	coeff_abs_level_greaterX_flag							
	0	1	2	3	4	5	6	7
initValue	480	50	208	304	400	288	178	306
	8	9	10	11	12	13	14	15
initValue	500	416	148	308	566	480	352	308
	16	17	18	19	20	21	22	23
initValue	566	858	322	246	274	304	400	130
b	24	25	26	27	28	29	30	31
initValue	146	144	434	290	2	178	500	448
	32	33	34	35	Na	Na	Na	Na
initValue	162	372	532	888	Na	Na	Na	Na

9.3.2.3 Initialization process for the arithmetic decoding engine

Outputs of this process are the initialized decoding engine registers ivlCurrRange and ivlOffset.

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 16384 and ivlOffset is set equal to the value returned from read_bits(14) interpreted as a 14-bit binary representation of an unsigned integer with the most significant bit written first.

9.3.3 Binarization process

9.3.3.1 General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 9-64 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the unary binarization process, the truncated Rice (TR) binarization process, the k-th order Exp-Golomb (EGk) binarization process, the fixed-length (FL) binarization process and the truncated binary (TB) binarization process are given in clauses 9.3.3.2 through 9.3.3.6, respectively. Other binarization is specified in clause 9.3.3.7.

Table 9-64 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	**Input parameters**
slice_data( )	end_of_tile_one_bit	FL	cMax = 1
split_unit( )	split_cu_flag[ ][ ]	FL	cMax = 1
	btt_split_flag[ ][ ]	FL	cMax = 1
	btt_split_dir[ ][ ]	FL	cMax = 1
	btt_split_type[ ][ ]	FL	cMax = 1
	split_unit_coding_order_flag[ ][ ]	FL	cMax = 1
coding_unit( )	cu_skip_flag[ ][ ]	FL	cMax = 1

Table 9-64 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
	mvp_idx_l0[ ][ ]	U	cMax = 3
	mvp_idx_l1[ ][ ]	U	cMax = 3
	mvp_idx[ ][ ]	TR	cMax = 5
	mmvd_flag[ ][ ]	FL	cMax = 1
	mmvd_group_idx[ ][ ]	TR	cMax = 2
	mmvd_merge_idx[ ][ ]	TR	cMax = 3
	mmvd_distance_idx[ ][ ]	TR	cMax = 7
	mmvd_direction_idx[ ][ ]	FL	cMax = 3
	affine_flag[ ][ ]	FL	cMax = 1
	affine_merge_idx[ ][ ]	TR	cMax = 5
	affine_mode[ ][ ]	FL	cMax = 1
	pred_mode_flag[ ][ ]	FL	cMax = 1
	intra_pred_mode[ ][ ]	U	cMax = 4
	intra_luma_pred_mpm_flag[ ][ ]	FL	cMax = 1
	intra_luma_pred_mpm_idx[ ][ ]	FL	cMax = 1
	intra_luma_pred_pims_flag[ ][ ]	FL	cMax = 1
	intra_luma_pred_pims_idx[ ][ ]	FL	cMax = 7
	intra_luma_pred_rem_mode[ ][ ]	TB	cMax = 22
	intra_chroma_pred_mode[ ][ ]	9.3.3.7	-
	direct_mode_flag[ ][ ]	FL	cMax = 1
	ibc_flag[ ][ ]	FL	cMax = 1
	amvr_idx[ ][ ]	TR	cMax = 4
	merge_mode_flag[ ][ ]	FL	cMax = 1
	inter_pred_idc[ ][ ]	TR	cMax = 2
	bi_pred_idx[ ][ ]	TR	cMax = 2
	ref_idx_l0[ ][ ]	U	cMax = num_ref_idx_active_minus1[0]
	abs_mvd_l0[ ][ ]	EG0	-
	mvd_flag_l0[ ][ ]	FL	cMax = 1
	ref_idx_l1[ ][ ]	U	cMax = num_ref_idx_active_minus1[1]
	abs_mvd_l1[ ][ ]	EG0	-
	mvd_flag_l1[ ][ ]	FL	cMax = 1
	cbf_all	FL	cMax = 1

Table 9-64 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
transform_unit( )	cbf_luma	FL	cMax = 1
	cbf_cb	FL	cMax = 1
	cbf_cr	FL	cMax = 1
	ats_cu_intra_flag[ ][ ]	FL	cMax = 1
	ats_hor_mode[ ][ ]	FL	cMax = 1
	ats_ver_mode[ ][ ]	FL	cMax = 1
	ats_cu_inter_flag	FL	cMax = 1
	ats_cu_inter_quad_flag	FL	cMax = 1
	ats_cu_inter_horizontal_flag	FL	cMax = 1
	ats_cu_inter_pos_flag	FL	cMax = 1
residual_coding_baseline( )	coeff_zero_run	U	cMax = (1 << (log2TrafoWidth + log2TrafoHeight)) – 1
	coeff_abs_level_minus1	U	-
	coeff_sign_flag	FL	cMax = 1
	coeff_last_flag	FL	cMax = 1
residual_coding_main( )	last_sig_coeff_x_prefix	TR	cMax = (log2TrafoSize << 1) − 1, cRiceParam = 0
	last_sig_coeff_y_prefix	TR	cMax = (log2TrafoSize << 1) − 1, cRiceParam = 0
	last_sig_coeff_x_suffix	FL	cMax = (1 << ((last_sig_coeff_x_prefix >> 1) − 1) − 1)
	last_sig_coeff_y_suffix	FL	cMax = (1 << ((last_sig_coeff_y_prefix >> 1) − 1) − 1)
	sig_coeff_flag[ ][ ]	FL	cMax = 1
	coeff_abs_level_greaterA_flag[ ]	FL	cMax = 1
	coeff_abs_level_greaterB_flag[ ]	FL	cMax = 1
	coeff_abs_level_remaining[ ]	9.3.3.8	current sub-block scan index i, baseLevel
	coeff_sign_flag[ ]	U	cMax = 1<<32

9.3.3.2 Unary (U) binarization process

Inputs to this process is a request for a U binarization for a syntax element.

Output of this process is the U binarization of the syntax element.

The bin string of a syntax element having (unsigned integer) value synElVal is a bit string of length synElVal + 1 indexed by binIdx. The bins for binIdx less than synElVal are equal to 1. The bin with binIdx equal to synElVal is equal to 0.

Table 9-65 illustrates the bin strings of the unary binarization for a syntax element.

Table 9-65 – Bin string of the unary binarization (informative)

Value of syntax element	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

9.3.3.3 Truncated Rice (TR) binarization process

Input to this process is a request for a TR binarization for a syntax element with value synVal, cMax, and cRiceParam.

Output of this process is the TR binarization of the syntax element.

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 synVal, prefixVal, is derived as follows:

$$prefixVal = synVal \gg cRiceParam \tag{9-6}$$

– The prefix of the TR bin string is specified as follows:

  – If prefixVal is less than cMax >> cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0.

  – Otherwise, the bin string is a bit string of length cMax >> cRiceParam with all bins being equal to 1.

When cMax is greater than synVal, the suffix of the TR bin string is present and it is derived as follows:

– The suffix value of synVal, suffixVal, is derived as follows:

$$suffixVal = synVal - ((prefixVal) \ll cRiceParam) \tag{9-7}$$

– The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as in clause 9.3.3.5 for suffixVal with a cMax value equal to (1 << cRiceParam) – 1.

NOTE – For the input parameter cRiceParam = 0 the TR binarization is exactly a truncated unary binarization and it is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

9.3.3.4 k-th order Exp-Golomb (EGk) binarization process

Inputs to this process is a request for an EGk binarization for a syntax element.

Output of this process is the EGk binarization of the syntax element.

The bin string of the EGk binarization process of a syntax element synVal 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( synVal )
stopLoop = 0
do {
    if( absV >= ( 1 << k ) ) {
        put( 1 )
        absV = absV − ( 1 << k )
```

```
            k++
        } else {
            put( 0 )                                                    (9-8)
            while( k− − )
                put( ( absV >> k ) & 1 )
            stopLoop = 1
        }
    } while( !stopLoop )
```

NOTE – The specification for the k-th order Exp-Golomb (EGk) code uses 1's and 0's in reverse meaning for the unary part of the Exp-Golomb code of 0-th order as specified in clause 9.2.

9.3.3.5 Fixed-length (FL) binarization process

Inputs to this process are a request for an FL binarization for a syntax element and cMax.

Output of this process is the FL binarization of the syntax element.

FL binarization is constructed by using a fixedLength-bit unsigned integer bin string of the syntax element value, 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.

9.3.3.6 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:

$$\begin{aligned} n &= cMax + 1 \\ k &= Floor(Log2(n)) \\ u &= (1 \ll (k + 1)) - n \end{aligned} \tag{9-9}$$

– If synVal is less than u, the TB bin string is derived by invoking the FL binarization process specified in clause 9.3.3.5 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 9.3.3.5 for (synVal + u) with a cMax value equal to (1 << (k + 1)) − 1.

9.3.3.7 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 9-66.

Table 9-66 – Binarization for intra_chroma_pred_mode

Value of intra_chroma_pred_mode	Bin string
0	1
1	00
2	010
3	0110
4	01110

9.3.3.8 Binarization process for coeff_abs_level_remaining

Input to this process is a request for a binarization for the syntax element coeff_abs_level_remaining [n], the colour component cIdx, the current sub-block index i, 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 coeff_abs_level_remaining[] as specified in clause 9.3.4.2.10 with the variable baseLevel, the colour component index cIdx, 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 numBinRem is derived with given cRiceParam from Table 9-67.

Table 9-67 –Specification of cRiceParam based on locSumAbs

cRiceParam	**0**	**1**	**2**	**3**
numBinRem	6	5	6	3

The variable cMax is derived from cRiceParam as:

$$cMax = numBinRem << cRiceParam \quad (9\text{-}10)$$

The binarization of the syntax element coeff_abs_level_remaining [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 coeff_abs_level_remaining [n], prefixVal, is derived as follows:

$$prefixVal = Min(cMax, coeff\_abs\_level\_remaining [n]) \quad (9\text{-}11)$$

– The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.3.3.3 for prefixVal with the variables cMax and cRiceParam as inputs.

When the prefix bin string is equal to the bit string of length numBinRem with all bits equal to 1, the suffix bin string is present and it is derived as follows:

– The suffix value of coeff_abs_level_remaining [n], suffixVal, is derived as follows:

$$suffixVal = coeff\_abs\_level\_remaining [n] - cMax \quad (9\text{-}12)$$

– The suffix bin string is specified by invoking the limited k-th order EGk binarization process as specified in clause 9.3.3.4 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1 and cRiceParam as input.

9.3.4 Decoding process flow

9.3.4.1 General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in clause 9.3.3.

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 9.3.4.2 is invoked with binIdx as input and ctxTable, ctxIdx and bypassFlag as outputs.
2. The arithmetic decoding process as specified in clause 9.3.4.2.6 is invoked with ctxTable, ctxIdx and bypassFlag as inputs and the value of the bin as output.

9.3.4.2 Derivation process for ctxTable, ctxIdx and bypassFlag

9.3.4.2.1 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 9-48:

– If the entry in Table 9-48 is not equal to "bypass", "terminate" or "na", the values of binIdx are decoded by invoking the DecodeDecision process as specified in clause 9.3.4.3.2 and the following applies:
  – ctxTable is specified in Table 9-4.
  – The variable ctxInc is specified by the corresponding entry in Table 9-48 and when more than one value is listed in Table 9-48 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 by the lowest value of ctxIdx in Table 9-4 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 9-48 is equal to "bypass", the values of binIdx are decoded by invoking the DecodeBypass process as specified in clause 9.3.4.3.4 and the following applies:
  – ctxTable is set equal to 0.
  – ctxIdx is set equal to 0.
  – bypassFlag is set equal to 1.
– Otherwise, if the entry in Table 9-48 is equal to "terminate", the values of binIdx are decoded by invoking the DecodeTerminate process as specified in clause 9.3.4.3.5 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 9-68 is equal to "na"), the values of binIdx do not occur for the corresponding syntax element.

Table 9-68 – Assignment of ctxInc to syntax elements with context coded bins

Syntax element	binIdx						
	0	**1**	**2**	**3**	**4**	**5**	**>= 6**
end_of_tile_one_bit	terminate	na	na	na	na	na	na
split_cu_flag[ ][ ]	0	na	na	na	na	na	na
btt_split_flag[ ][ ]	0..14 (clause 9.3.4.2.5)	na	na	na	na	na	na
btt_split_dir[ ][ ]	log2CbWidth – log2CbHeight + 2	na	na	na	na	na	na
btt_split_type[ ][ ]	0	na	na	na	na	na	na
split_unit_coding_order_flag[ ][ ]	0..11 (clause 9.3.4.2.3)	na	na	na	na	na	na
cu_skip_flag[ ][ ] (When sps_cm_init_flag = = 0)	0	na	na	na	na	na	na
cu_skip_flag[ ][ ] (When sps_cm_init_flag = = 1)	0,1 (clause 9.3.4.2.4)	na	na	na	na	na	na

mvp_idx_l0[ ][ ] (When sps_cm_init_flag = = 0)	0	1	2	2	na	na	na
mvp_idx_l1[ ][ ] (When sps_cm_init_flag = = 0)	0	1	2	2	na	na	na
mvp_idx[ ][ ]	0	1	2	3	4	bypass	bypass
mmvd_flag[ ][ ]	0	na	na	na	na	na	na
mmvd_group_idx[ ][ ]	0	1	na	na	na	na	na
mmvd_merge_idx[ ][ ]	0	1	2	bypass	bypass	bypass	bypass
mmvd_distance_idx[ ][ ]	0	1	2	3	4	5	6
mmvd_direction_idx[ ][ ]	0	1	na	na	na	na	na
affine_flag[ ][ ]	0,1 (clause 9.3.4.2.4)	na	na	na	na	na	na
affine_merge_idx[ ][ ]	0	1	2	3	4	bypass	bypass
affine_mode[ ][ ]	0	na	na	na	na	na	na
pred_mode_flag[ ][ ] (When sps_cm_init_flag = = 0)	0..2 (clause 9.3.4.2.4)	na	na	na	na	na	na
pred_mode_flag[ ][ ] (When sps_cm_init_flag = = 1)	0	na	na	na	na	na	na
intra_pred_mode[ ][ ]	0	1	1	1	1	na	na
intra_luma_pred_mpm_flag[ ][ ]	0	na	na	na	na	na	na
intra_luma_pred_mpm_idxidc[ ][ ]	0	na	na	na	na	na	na
intra_luma_pred_pims_flag[ ][ ]	bypass	na	na	na	na	na	na
intra_luma_pred_pims_idx[ ][ ]	bypass	bypass	bypass	na	na	na	na
intra_luma_pred_rem_mode [ ][ ]	bypass	bypass	bypass	bypass	bypass	bypass	bypass
intra_chroma_pred_mode[ ][ ]	0	bypass	bypass	na	na	na	na
ibc_flag[ ][ ] (When sps_cm_init_flag = = 0)	0	na	na	na	na	na	na
ibc_flag[ ][ ] (When sps_cm_init_flag = = 1)	0,1 (clause 9.3.4.2.4)	na	na	na	na	na	na
amvr_idx[ ][ ]	0	1	2	3	na	na	na
direct_mode_flag[ ][ ]	0	na	na	na	na	na	na
inter_pred_idc[ ][ ]	0	1	na	na	na	na	na
merge_mode_flag[ ][ ]	0	na	na	na	na	na	na
bi_pred_idx[ ][ ]	0	1	na	na	na	na	na
ref_idx_l0[ ][ ]	0	1	bypass	bypass	bypass	bypass	bypass
ref_idx_l1[ ][ ]	0	1	bypass	bypass	bypass	bypass	bypass
abs_mvd_l0[ ][ ]	0	bypass	bypass	bypass	bypass	bypass	bypass
abs_mvd_l1[ ][ ]	0	bypass	bypass	bypass	bypass	bypass	bypass
mvd_flag_l0[ ][ ]	bypass	na	na	na	na	na	na
mvd_flag_l1[ ][ ]	bypass	na	na	na	na	na	na
cbf_all	0	na	na	na	na	na	na
cbf_luma	0	na	na	na	na	na	na
cbf_cb	0	na	na	na	na	na	na
cbf_cr	0	na	na	na	na	na	na
ats_cu_intra_flag[ ][ ]	(subclause 9.3.4.2.11)	na	na	na	na	na	na
ats_hor_mode[ ][ ]	0	na	na	na	na	na	na

ats_ver_mode[ ][ ]	0	na	na	na	na	na	na
ats_cu_inter_flag[][]	(subclause 9.3.4.2.12)	na	na	na	na	na	na
ats_cu_inter_quad_flag[][]	0	na	na	na	na	na	na
ats_cu_inter_horizontal_flag[][]	(subclause 9.3.4.2.13)	na	na	na	na	na	na
ats_cu_inter_pos_flag[][]	0	na	na	na	na	na	na
coeff_zero_run	(subclause 9.3.4.2.2)	(subclause 9.3.4.2.2)					
coeff_abs_level_minus1	(subclause 9.3.4.2.2)	(subclause 9.3.4.2.2)					
coeff_last_flag	cIdx = = 0? 0 : 1	na	na	na	na	na	na
last_sig_coeff_x_prefix	(subclause 9.3.4.2.6)						
last_sig_coeff_y_prefix	(subclause 9.3.4.2.6)						
last_sig_coeff_x_suffix	bypass	bypass	bypass	bypass	bypass	bypass	bypass
last_sig_coeff_y_suffix	bypass	bypass	bypass	bypass	bypass	bypass	bypass
sig_coeff_flag	0..47 (subclause 9.3.4.2.7)	na	na	na	na	na	na
coeff_abs_level_greaterA_flag	0..18 (subclause 9.3.4.2.8)	na	na	na	na	na	na
coeff_abs_level_greaterB_flag	0..18 (subclause 9.3.4.2.9)	na	na	na	na	na	na
coeff_abs_level_remaining	bypass	bypass	bypass	bypass	bypass	bypass	bypass
coeff_sign_flag	bypass	na	na	na	na	na	Na

9.3.4.2.2 Derivation process of ctxInc for the coeff_zero_run and coeff_abs_level_minus1

Inputs to this process are the variable binIdx, the colour component index cIdx, and prev_level specifying the absolute value of a transform coefficient level at the previous scanning position.

Output of this process is the variable ctxInc.

If cIdx is equal to 0, the variable ctxInc for luma is derived as follows:

- ctxInc = ((Min (prev_level – 1, 5)) << 1) (9-13)

Otherwise (cIdx is greater than 0), the variable ctxInc for chroma is derived as follows:

- ctxInc = ((Min (prev_level – 1, 5)) << 1) + 12 (9-14)

9.3.4.2.3 Derivation process of ctxInc for split_unit_coding_order_flag

Inputs to this process are the variable binIdx, two variables nCbW and nCbH specifying the width and the height of the current spliting unit,

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

- If CbW is equal to CbH,

  ctxInc = ((Max (log2(CbW), log2(CbH)) – 2) << 1) (9-15)

- Otherwise

  ctxInc = ((Max (log2(CbW), log2(CbH)) – 2) << 1) + 1 (9-16)

9.3.4.2.4 Derivation process of ctxInc using neighbouring blocks syntax elements

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 nCbW and nCbH specifying the width and the height of the current luma coding block.

Output of this process is the variable ctxInc.

The luma location (xNbL, yNbL) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + nCbH − 1).

The availability derivation process for a coding block as specified below in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbL, yNbL) as inputs, and the output is assigned to the coding block availability flag availableL.

The luma location (xNbA, yNbA) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

The availability derivation process for a coding block as specified below in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbA, yNbA) as inputs, and the output is assigned to the coding block availability flag availableA.

The luma location (xNbR, yNbR) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb + nCbH − 1).

The availability derivation process for a coding block as specified below in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbR, yNbR) as inputs, and the output is assigned to the coding block availability flag availableR.

The assignment of ctxInc is specified as follows with condL, condA and condR for the syntax elements affine_flag[xCb][yCb], cu_skip_flag[xCb][yCb], pred_mode_flag[xCb][yCb], and ibc_flag[xCb][yCb] specified in Table 9-69:

- ctxInc =Min(condL && availableL + condA && availableA + condR && availableR , numCtx – 1) (9-17)

Table 9-69 – Specification of ctxInc using neighbouring block syntax elements

Syntax element	condL	condA	condR	numCtx
affine_flag[xCb][yCb]	affine_flag[xNbL][yNbL]	affine_flag[xNbL][yNbL]	affine_flag[xNbL][yNbL]	2
cu_skip_flag[xCb][yCb]	cu_skip_flag[xNbL][yNbL]	cu_skip_flag[xNbL][yNbL]	cu_skip_flag[xNbL][yNbL]	2
pred_mode_flag[xCb][yCb]	pred_mode_flag[xNbL][yNbL]	pred_mode_flag[xNbL][yNbL]	pred_mode_flag[xNbL][yNbL]	3
ibc_flag[xCb][yCb]	ibc_flag[xNbL][yNbL]	ibc_flag[xNbL][yNbL]	ibc_flag[xNbL][yNbL]	2

9.3.4.2.5 Derivation process of ctxInc for syntax element btt_split_flag

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 nCbW and nCbH specifying the width and the height of the current luma coding block.

Output of this process is the variable ctxInc.

The luma location (xNbL, yNbL) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

The availability derivation process for a coding block as specified below in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbL, yNbL) as inputs, and the output is assigned to the coding block availability flag availableL.

The luma location (xNbA, yNbA) inside the neighbouring luma coding block is set equal to (xCb , yCb − 1).

The availability derivation process for a coding block as specified below in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbA, yNbA) as inputs, and the output is assigned to the coding block availability flag availableA.

The luma location (xNbR, yNbR) inside the neighbouring luma coding block is set equal to (xCb + nCbW, yCb).

The availability derivation process for a coding block as specified below in clause 6.4.1 is invoked with the luma location (xCb, yCb), the current luma coding block width nCbW, the current luma coding block height nCbH and the luma location (xNbR, yNbR) as inputs, and the output is assigned to the coding block availability flag availableR.

A variable numSmaller is derived as following:

- numSmaller = (CbHeight[xNbL][yNbL] < nCbH) && availableL + (CbWidth[xNbA][yNbA] < nCbW) && availableA + (CbHeight[xNbR][yNbR] < nCbH) && availableR (9-18)

The assignment of ctxInc is specified as follows with ctxSetIdx specified in Table 9-70:

- ctxInc =Min(numSmaller , 2) + 3 * ctxSetIdx (9-19)

Table 9-70 – Specification of ctxSetIdx based on nCbW and nCbH

Log2(nCbW) – 2	**Log2(nCbH) – 2**					
	0	**1**	**2**	**3**	**4**	**5**
0	n/a	4	4	n/a	n/a	n/a
1	4	4	3	3	2	2
2	4	3	3	2	2	1
3	n/a	3	2	2	1	1
4	n/a	2	2	1	1	0
5	n/a	2	1	1	0	0

9.3.4.2.6 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 and the associated transform size log2TrafoSize which is log2TbWidth for last_sig_coeff_x_prefix and log2TbHeight for last_sig_coeff_y_prefix, respectively.

Output of this process is the variable ctxInc.

The variables ctxOffset and ctxShift are derived as follows:

- If cIdx is equal to 0, ctxOffset is set equal to 3 * (log2TrafoSize − 2) + ((log2TrafoSize − 1) >> 2) and ctxShift is set equal to (log2TrafoSize + 1) >> 2.
- Otherwise (cIdx is greater than 0), ctxOffset is set equal to 0 and ctxShift is set equal to 2.

When cIdx is equal to 0 and log2TrafoSize > 3, the variable ctxOffset is further updated as follows:

- ctxOffset += (max(0, log2TrafoSize – 6)) << 1 + max(0, log2TrafoSize – 7)) (9-20)

The variable ctxInc is derived as follows:

- ctxInc = (binIdx >> ctxShift) + ctxOffset (9-21)

9.3.4.2.7 Derivation process of ctxInc for the syntax element sig_coeff_flag

Inputs to this process are the colour component index cIdx, the current coefficient scan location (xC, yC) and the transform block sizes log2TbWidth and log2TbHeight.

Output of this process is the variable ctxInc.

The variable sigCtx depends on the current location (xC, yC), the colour component index cIdx and the transform block size. For the derivation of sigCtx, the following applies:

Variable numFlags is set equal to 0.

When xC is less than (1 << log2TbWidth) − 1, the following applies:

- numFlags += (coefs[xC + 1][yC] != 0) (9-22)

When xC is less than (1 << log2TbWidth) − 2, the following applies:

- numFlags += (coefs[xC + 2][yC] != 0) (9-23)

When xC is less than (1 << log2TbWidth) − 1, and yC is less than (1 << log2TbHeight) − 1,the following applies:

- numFlags += (coefs[xC + 1][yC + 1] != 0) (9-24)

When yC is less than (1 << log2TbHeight) − 1,the following applies:

- numFlags += (coefs[xC][yC + 1] != 0) (9-25)

When yC is less than (1 << log2TbHeight) − 2,the following applies:

- numFlags += (coefs[xC][yC + 2] != 0) (9-26)

The variable sigCtx is derived as follows:

- sigCtx = min (numFlags,4) + 1 (9-27)

When xC + yC < 2, variable sigCtx is updated as follows:

- sigCtx = min (sigCtx,2) (9-28)

The context index offset ctxOffset is derived using the colour component index cIdx and sigCtx as follows:

- If cIdx is equal to 0, ctxOffset is derived as follows:
- ctxOffset = ((xC + yC) < 2) ? 0 : ((xC + yC < 5) ? 2 : 7) (9-29)

- Otherwise (cIdx is greater than 0), ctxOffset is derived as follows:
- ctxOffset = ((xC + yC) < 2) ? 0 : 2 (9-30)

The output ctxInc is derived as follows:

- ctxInc = sigCtx + ctxOffset (9-31)

9.3.4.2.8 Derivation process of ctxInc for the syntax element coeff_abs_level_greaterA_flag

Inputs to this process are the colour component index cIdx, the current coefficient scan location (xC, yC), the last coefficient location(lastX, lastY) and the transform block sizes log2TbWidth and log2TbHeight.

Output of this process is the variable ctxInc.

The variable ctxInc depends on the current location (xC, yC), the colour component index cIdx, last significant position and the transform block size. For the derivation of ctxInc, the following applies:

Variable numFlags is set equal to 0.

When xC is less than (1 << log2TbWidth) − 1, the following applies:

numFlags += (Abs(coefs[xC + 1][yC]) > 1) (9-32)

When xC is less than (1 << log2TbWidth) − 2, the following applies:

numFlags += (Abs(coefs[xC + 2][yC]) > 1) (9-33)

When xC is less than (1 << log2TbWidth) − 1, and yC is less than (1 << log2TbHeight) − 1,the following applies:

numFlags += (Abs(coefs[xC + 1][yC + 1]) > 1)
(9-45)

When yC is less than (1 << log2TbHeight) − 1,the following applies:

numFlags += (Abs(coefs[xC][yC + 1]) > 1) (9-34)

When yC is less than (1 << log2TbHeight) − 2,the following applies:

numFlags += (Abs(coefs[xC][yC + 2]) > 1) (9-35)

The variable ctxInc is derived as follows:

ctxInc = min (numFlags, 3) + 1 (9-36)

If cIdx is equal to 0, ctxInc is futher updated as follows:

ctxInc += (xC = =0 && yC = = 0) ? 0 : ((xC <= lastX / 2) && (yC <= lastY / 2)) ? 4 : 8 (9-37)

9.3.4.2.9 Derivation process of ctxInc for the syntax element coeff_abs_level_greaterB_flag

Inputs to this process are the colour component index cIdx, the current coefficient scan location (xC, yC), the last coefficient location(lastX, lastY) and the transform block sizes log2TbWidth and log2TbHeight.

Output of this process is the variable ctxInc.

The variable ctxInc depends on the current location (xC, yC), the colour component index cIdx, last significant position and the transform block size. For the derivation of ctxInc, the following applies:

Variable numFlags is set equal to 0.

When xC is less than (1 << log2TbWidth) − 1, the following applies:

numFlags += (Abs(coefs[xC + 1][yC]) > 2) (9-38)

When xC is less than (1 << log2TbWidth) − 2, the following applies:

numFlags += (Abs(coefs[xC + 2][yC]) > 2) (9-39)

When xC is less than (1 << log2TbWidth) − 1, and yC is less than (1 << log2TbHeight) − 1,the following applies:

numFlags += (Abs(coefs[xC + 1][yC + 1]) > 2) (9-40)

When yC is less than (1 << log2TbHeight) − 1,the following applies:

numFlags += (Abs(coefs[xC][yC + 1]) > 2) (9-41)

When yC is less than (1 << log2TbHeight) − 2,the following applies:

numFlags += (Abs(coefs[xC][yC + 2]) > 2) (9-42)

The variable ctxInc is derived as follows:

ctxInc = min (numFlags, 3) + 1 (9-43)

If cIdx is equal to 0, ctxInc is futher updated as follows:

ctxInc += (xC = =0 && yC = = 0) ? 0 : ((xC <= lastX / 2) && (yC <= lastY / 2)) ? 4 : 8 (9-44)

9.3.4.2.10 Rice parameter derivation process for coeff_abs_level_remaining

Inputs to this process are the base level baseLevel, the current coefficient scan location (xC, yC) and the transform block sizes log2TbWidth and log2TbHeight.

Output of this process is the Rice parameter cRiceParam.

The variable cRiceParam depends on the current location (xC, yC), corresponding based level and the transform block size. For the derivation of cRiceParam, the following applies:

Variable locSumAbs is set equal to 0.

When xC is less than (1 << log2TbWidth) − 1, the following applies:

locSumAbs += Abs(coefs[xC + 1][yC]) (9-45)

When xC is less than (1 << log2TbWidth) − 2, the following applies:

locSumAbs += Abs(coefs[xC + 2][yC]) (9-46)

When xC is less than (1 << log2TbWidth) − 1, and yC is less than (1 << log2TbHeight) − 1,the following applies:

locSumAbs += Abs(coefs[xC + 1][yC + 1]) (9-47)

When yC is less than (1 << log2TbHeight) − 1,the following applies:

locSumAbs += Abs(coefs[xC][yC + 1]) (9-48)

When yC is less than (1 << log2TbHeight) − 2,the following applies:

locSumAbs += Abs(coefs[xC][yC + 2]) (9-49)

locSumAbs = Clip3(0, 31, locSumAbs − baseLevel * 5) (9-50)

Given the varialbel locSumAbs, the Rice parameter cRiceParam is derived as sepcified in Table 9-71.

Table 9-71 – Specification of cRiceParam based on locSumAbs

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
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

9.3.4.2.11 Derivation process of ctxInc for syntax element ats_cu_intra_flag

Inputs to this process are the variable binIdx, two variables nCbW and nCbH specifying the width and the height of the current splitting unit,

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

$$ctxInc = ((Max (log2(nCbW), log2(nCbH)) – 2)) \quad (9\text{-}51)$$

9.3.4.2.12 Derivation process of ctxInc for syntax element ats_cu_inter_flag

Inputs to this process are the variable binIdx, two variables nCbW and nCbH specifying the width and the height of the current splitting unit,

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

$$ctxInc = (log2(nCbW) + log2(nCbH)) >= 8 \ ? \ 0 : 1 \quad (9\text{-}52)$$

9.3.4.2.13 Derivation process of ctxInc for syntax element ats_cu_inter_horizontal _flag

Inputs to this process are the variable binIdx, two variables nCbW and nCbH specifying the width and the height of the current spliting unit,

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

$$ctxInc = (nCbW = = nCbH) \ ? \ 0 : (nCbW < nCbH \ ? \ 1 : 2) \quad (9\text{-}53)$$

9.3.4.3 Arithmetic decoding process

9.3.4.3.1 General

Inputs to this process are ctxTable, ctxIdx and bypassFlag, as derived in clause 9.3.4.2, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

For decoding the value of a bin, the context index table ctxTable and the ctxIdx are passed to the arithmetic decoding process DecodeBin(ctxTable, ctxIdx), which is specified as follows:

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in clause 9.3.4.3.4 is invoked.

– Otherwise, if bypassFlag is equal to 0, ctxTable is equal to 0 and ctxIdx is equal to 0, DecodeTerminate( ) as specified in clause 9.3.4.3.5 is invoked.

– Otherwise (bypassFlag is equal to 0 and ctxTable is not equal to 0), DecodeDecision( ) as specified in clause 9.3.4.3.2 is invoked.

9.3.4.3.2 Arithmetic decoding process for a binary decision

9.3.4.3.2.1 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.

The following applies:

1. The value of the variable ivlLpsRange is derived as follows:

$$ivlLpsRange = (valState * ivlCurrRange) >> 9 \quad (9\text{-}54)$$

$$ivlLpsRange = ivlLpsRange < 437 \ ? \ 437 : ivlLpsRange \quad (9\text{-}55)$$

2. 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 is performed as specified in clause 9.3.4.3.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in clause 9.3.4.3.3.

9.3.4.3.2.2 State transition process

Inputs to this process are the current valState, the decoded value binVal and valMps values of the context variable associated with ctxTable and ctxIdx.

Outputs of this process are the updated valState and valMps of the context variable associated with ctxIdx.

Depending on the decoded value binVal, the update of the two variables valState and valMps associated with ctxIdx is derived as follows:

```
if( binVal  = =  valMps ) {
    valState = valState – ( valState + 16 ) >> 5
}
else {
    valState = valState + ( (1 << 9) – valState + 16 ) >> 5
    if( valState > (1 << 8) ) {
        valMps = 1 – valMps
        valState = (1 << 9 ) – valState
    }
}
```
(9-56)

9.3.4.3.3 Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset.

The current value of ivlCurrRange is first compared to 8192 and then the following applies:

– If ivlCurrRange is greater than or equal to 8192, no renormalization is needed and the RenormD process is finished;

– Otherwise (ivlCurrRange is less than 8192), 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.

9.3.4.3.4 Bypass decoding process for binary decisions

Inputs to this process are bits from slice 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.

The following applies:

– The value of ivlCurrRange is left shifted by 1.

– If ivOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 and ivlOffset if decremented by ivLCurrRange.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0.

– The value of ivlCurrRange is right 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.

9.3.4.3.5 Decoding process for binary decisions before termination

Inputs to this process are bits from slice 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 the decoding of end_of_tile_on_bit corresponding to ctxTable equal to 0 and ctxIdx equal to 0.

First, the value of ivlCurrRange is decremented by 1. Then, the value of ivlOffset is compared to the value of ivlCurrRange and then the following applies:

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 clause 9.3.4.3.3.

Annex A

Profiles and levels

(This annex forms an integral part of this International Standard.)

A.1 Overview of profiles and levels

Profiles and levels specify restrictions on the bitstreams and hence limits on the capabilities needed to decode the bitstreams. Profiles and levels may also be used to indicate interoperability points between individual decoder implementations.

NOTE 1 – This document does not include individually selectable "options" at the decoder, as this would increase interoperability difficulties.

Each profile specifies a subset of algorithmic features and limits that shall be supported by all decoders conforming to that profile.

NOTE 2 – Encoders are not required to make use of any particular subset of features supported in a profile.

Each level specifies a set of limits on the values that may be taken by the syntax elements of this document. The same set of level definitions is used with all profiles, but individual implementations may support a different level for each supported profile. For any given profile, a level generally corresponds to a particular decoder processing load and memory capability.

The profiles that are specified in clause A.3 are also referred to as the profiles specified in Annex A.

A.2 Requirements on video decoder capability

Capabilities of video decoders conforming to this document are specified in terms of the ability to decode video streams conforming to the constraints of profiles and levels specified in this annex and other annexes. When expressing the capabilities of a decoder for a specified profile, the level supported for that profile should also be expressed.

Specific values are specified in this annex and other annexes for the syntax elements profile_idc and level_idc. All other values of profile_idc and level_idc are reserved for future use by ISO/IEC.

NOTE – Decoders should not infer that a reserved value of profile_idc between the values specified in this document indicates intermediate capabilities between the specified profiles, as there are no restrictions on the method to be chosen by ISO/IEC for the use of such future reserved values. However, decoders should infer that a reserved value of level_idc between the values specified in this document indicates intermediate capabilities between the specified levels.

A.3 Profiles

A.3.1 General

All constraints for PPSs that are specified are constraints for PPSs that are activated when the bitstream is decoded. All constraints for SPSs that are specified are constraints for SPSs that are activated when the bitstream is decoded.

A.3.2 Baseline profile

Bitstreams conforming to the Baseline profile shall obey the following constraints:

– Active SPSs shall have chroma_format_idc equal to 0 or 1 only.

– Active SPSs shall have bit_depth_luma_minus8 equal to 0 or 2 only.

– Active SPSs shall have bit_depth_chroma_minus8 equal to 0 or 2 only.

– Active SPSs shall have sps_btt_flag equal to 0 only.

– Active SPSs shall have sps_suco_flag equal to 0 only.

– Active SPSs shall have sps_amvr_flag equal to 0 only.

– Active SPSs shall have sps_mmvd_flag equal to 0 only.

– Active SPSs shall have sps_affine_flag equal to 0 only.

– Active SPSs shall have sps_dmvr_flag equal to 0 only.

– Active SPSs shall have sps_alf_flag equal to 0 only.

– Active SPSs shall have sps_admvp_flag equal to 0 only.
– Active SPSs shall have sps_eipd_flag equal to 0 only.
– Active SPSs shall have sps_adcc_flag equal to 0 only.
– Active SPSs shall have sps_amis_flag equal to 0 only.
– Active SPSs shall have sps_ibc_flag equal to 0 only.
– Active SPSs shall have sps_iqt_flag equal to 0 only.
– Active SPSs shall have sps_htdf_flag equal to 0 only.
– Active SPSs shall have sps_addb_flag equal to 0 only.
– Active SPSs shall have sps_cm_init_flag equal to 0 only.
– Active SPSs shall have sps_ats_flag equal to 0 only.
– Active SPSs shall have sps_rpl_flag equal to 0 only.
– Active SPSs shall have sps_pocs_flag equal to 0 only.
– Active SPSs shall have sps_dquant_flag equal to 0 only.
– Active PPSs shall have single_tile_in_pic_flag equal to 1 only.
– Each slice in the bitstream shall have single_tile_in_slice_flag equal to 1 only.
– No NAL unit in the bitsream shall have NalUnitType equal to HPS_NUT.
– The level constraints specified for the Main profile in clause A.4 shall be fulfilled.

Conformance of a bitstream to the Baseline profile is indicated by profile_idc equal to 0.

Decoders conforming to the Baseline profile at a specific level (identified by a specific value of level_idc) shall be capable of decoding all bitstreams for which all of the following conditions apply:

– The bitstream is indicated to conform to the Baseline profile.
– The bitstream is indicated to conform to a level that is lower than or equal to the specified level.

A.3.3 Main profile

Bitstreams conforming to the Main profile shall obey the following constraints:

– Active SPSs shall have chroma_format_idc equal to 0 or 1 only.
– Active SPSs shall have bit_depth_luma_minus8 equal to 0 or 2 only.
– Active SPSs shall have bit_depth_chroma_minus8 equal to 0 or 2 only.
– The level constraints specified for the Main profile in clause A.4 shall be fulfilled.

Conformance of a bitstream to the Main profile is indicated by profile_idc equal to 1.

Decoders conforming to the Main profile at a specific level (identified by a specific value of level_idc) shall be capable of decoding all bitstreams for which all of the following conditions apply:

– The bitstream is indicated to conform to the Main profile.
– The bitstream is indicated to conform to a level that is lower than or equal to the specified level.

A.4 Levels

A.4.1 General level limits

For purposes of comparison of level capabilities, a particular level is considered to be a lower level than some other level when the value of the level_idc of the particular level is less than that of the other level.

The following is specified for expressing the constraints in this annex:

– Let access unit n be the n-th access unit in decoding order, with the first access unit being access unit 0 (i.e., the 0-th access unit).
– Let picture n be the coded picture or the corresponding decoded picture of access unit n.

Bitstreams conforming to a profile at a specified level shall obey the following constraints for each bitstream conformance test as specified in Annex C:

a) PicSizeInSamplesY shall be less than or equal to MaxLumaPs, where MaxLumaPs is specified in Table A.1.

b) The value of pic_width_in_luma_samples shall be less than or equal to Sqrt(MaxLumaPs * 8).

c) The value of pic_height_in_luma_samples shall be less than or equal to Sqrt(MaxLumaPs * 8).

d) For level 5 and higher levels, the value of CtbSizeY shall be equal to 32 or 64.

e) The value of num_tile_columns_minus1 shall be less than MaxTileCols and num_tile_rows_minus1 shall be less than MaxTileRows, where MaxTileCols and MaxTileRows are specified in Table A.1.

f) For the VCL HRD parameters, CpbSize[i] shall be less than or equal to CpbVclFactor * MaxCPB for at least one value of i in the range of 0 to cpb_cnt_minus1, inclusive, where CpbSize[i] is specified in clause E.3.3 based on parameters selected as specified in clause C.1, CpbVclFactor is specified in Table A.3, and MaxCPB is specified in Table A.1 in units of CpbVclFactor bits.

g) For the NAL HRD parameters, CpbSize[i] shall be less than or equal to CpbNalFactor * MaxCPB for at least one value of i in the range of 0 to cpb_cnt_minus1, inclusive, where CpbSize[i] is specified in clause E.3.3 based on parameters selected as specified in clause C.1, CpbNalFactor is specified in Table A.3, and MaxCPB is specified in Table A.1 in units of CpbNalFactor bits.

Table A.1 specifies the limits for each level.

A level to which a bitstream conforms are indicated by the syntax element level_idc as follows:

– level_idc shall be set equal to a value of 30 times the level number specified in Table A.1.

Table A.1 – General level limits

Level	Max luma picture size MaxLumaPs (samples)	Max CPB size MaxCPB (CpbVclFactor or CpbNalFactor bits)	Max # of slices per picture MaxSlicesPerPicture	Max # of tile rows MaxTileRows	Max # of tile columns MaxTileCols
1	36 864	350	16	1	1
2	122 880	1 500	16	1	1
2.1	245 760	3 000	20	1	1
3	552 960	6 000	30	2	2
3.1	983 040	10 000	40	3	3
4	2 228 224	12 000	75	5	5
4.1	2 228 224	20 000	75	5	5
5	8 912 896	25 000	200	11	10
5.1	8 912 896	40 000	200	11	10
5.2	8 912 896	60 000	200	11	10
6	35 651 584	60 000	600	22	20
6.1	35 651 584	120 000	600	22	20
6.2	35 651 584	240 000	600	22	20

A.4.2 Profile-specific level limits

The following is specified for expressing the constraints in this annex:

– Let the variable fR be set equal to 1 ÷ 300.

The variable HbrFactor is defined as follows:

– When the bitstream is indicated to conform to the Baseline profile or the Main profile, HbrFactor is set equal to 1.

The variable BrVclFactor, which represents the VCL bit rate scale factor, is set equal to CpbVclFactor * HbrFactor.

The variable BrNalFactor, which represents the NAL bit rate scale factor, is set equal to CpbNalFactor * HbrFactor.

The variable MinCr is set equal to MinCrBase * MinCrScaleFactor ÷ HbrFactor.

The value of sps_max_dec_pic_buffering_minus1 + 1 shall be less than or equal to MaxDpbSize, which is derived as follows:

```
if( PicSizeInSamplesY <= ( MaxLumaPs >> 2 ) )
    MaxDpbSize = Min( 4 * maxDpbPicBuf, 16 )
else if( PicSizeInSamplesY <= ( MaxLumaPs >> 1 ) )
    MaxDpbSize = Min( 2 * maxDpbPicBuf, 16 )                    (A-1)
else if( PicSizeInSamplesY <= ( ( 3 * MaxLumaPs ) >> 2 ) )
    MaxDpbSize = Min( ( 4 * maxDpbPicBuf ) / 3, 16 )
else
    MaxDpbSize = maxDpbPicBuf
```

where MaxLumaPs is specified in Table A.1, and maxDpbPicBuf is equal to 6.

Bitstreams conforming to the Baseline or Main profile at a specified level shall obey the following constraints for each bitstream conformance test as specified in Annex C:

a) The nominal removal time of access unit n (with n greater than 0) from the CPB, as specified in clause C.2.3, shall satisfy the constraint that $t_{r,n}(n) - t_r(n - 1)$ is greater than or equal to Max(PicSizeInSamplesY ÷ MaxLumaSr, fR) for the value of PicSizeInSamplesY of picture n − 1, where MaxLumaSr is the value specified in Table A.2 that applies to picture n − 1.

b) The difference between consecutive output times of pictures from the DPB, as specified in clause C.3.3, shall satisfy the constraint that $\Delta t_{o,dpb}(n)$ is greater than or equal to Max(PicSizeInSamplesY ÷ MaxLumaSr, fR) for the value of PicSizeInSamplesY of picture n, where MaxLumaSr is the value specified in Table A.2 for picture n, provided that picture n is a picture that is output and is not the last picture of the bitstream that is output.

c) The removal time of access unit 0 shall satisfy the constraint that the number of slices in picture 0 is less than or equal to Min(Max(1, MaxSlicesPerPicture * MaxLumaSr / MaxLumaPs * ($t_r(0) - t_{r,n}(0)$) + MaxSlicesPerPicture * PicSizeInSamplesY / MaxLumaPs), MaxSlicesPerPicture), for the value of PicSizeInSamplesY of picture 0, where MaxSlicesPerPicture, MaxLumaPs, and MaxLumaSr are the values specified in Table A.1 and Table A.2, respectively, that apply to picture 0.

d) The difference between consecutive CPB removal times of access units n and n − 1 (with n greater than 0) shall satisfy the constraint that the number of slices in picture n is less than or equal to Min(Max(1, MaxSlicesPerPicture * MaxLumaSr / MaxLumaPs * ($t_r(n) - t_r(n - 1)$)), MaxSlicesPerPicture), where MaxSlicesPerPicture, MaxLumaPs and MaxLumaSr are the values specified in Table A.1 and Table A.2 that apply to picture n.

e) For the VCL HRD parameters, BitRate[i] shall be less than or equal to BrVclFactor * MaxBR for at least one value of i in the range of 0 to cpb_cnt_minus1, inclusive, where BitRate[i] is specified in clause E.3.3 based on parameters selected as specified in clause C.1 and MaxBR is specified in Table A.2 in units of BrVclFactor bits/s.

f) For the NAL HRD parameters, BitRate[i] shall be less than or equal to BrNalFactor * MaxBR for at least one value of i in the range of 0 to cpb_cnt_minus1, inclusive, where BitRate[i] is specified in clause E.3.3 based on parameters selected as specified in clause C.1 and MaxBR is specified in Table A.2 in units of BrNalFactor bits/s.

g) The sum of the NumBytesInNalUnit variables for access unit 0 shall be less than or equal to FormatCapabilityFactor * (Max(PicSizeInSamplesY, fR * MaxLumaSr) + MaxLumaSr * ($t_r(0) - t_{r,n}(0)$)) ÷ MinCr for the value of PicSizeInSamplesY of picture 0, where MaxLumaSr and FormatCapabilityFactor are the values specified in Table A.2 and Table A.3, respectively, that apply to picture 0.

h) The sum of the NumBytesInNalUnit variables for access unit n (with n greater than 0) shall be less than or equal to FormatCapabilityFactor * MaxLumaSr * (t_r(n) − t_r(n − 1)) ÷ MinCr, where MaxLumaSr and FormatCapabilityFactor are the values specified in Table A.2 and Table A.3, respectively, that apply to picture n.

i) The removal time of access unit 0 shall satisfy the constraint that the number of tiles in picture 0 is less than or equal to Min(Max(1, MaxTileCols * MaxTileRows * 120 * (t_r(0) − $t_{r,n}$(0)) + MaxTileCols * MaxTileRows * PicSizeInSamplesY / MaxLumaPs), MaxTileCols * MaxTileRows), for the value of PicSizeInSamplesY of picture 0, where MaxTileCols and MaxTileRows are the values specified in Table A.1 that apply to picture 0.

j) The difference between consecutive CPB removal times of access units n and n − 1 (with n greater than 0) shall satisfy the constraint that the number of tiles in picture n is less than or equal to Min(Max(1, MaxTileCols * MaxTileRows * 120 * (t_r(n) − t_r(n − 1))), MaxTileCols * MaxTileRows), where MaxTileCols and MaxTileRows are the values specified in Table A.1$_{k)}$ that apply to picture n.

Table A.2 – Level limits for the video profiles

Level	Max luma sample rate MaxLumaSr (samples/sec)	Max bit rate MaxBR (BrVclFactor or BrNalFactor bits/s)	Min compression ratio MinCrBase
1	552 960	128	2
2	3 686 400	1 500	2
2.1	7 372 800	3 000	2
3	16 588 800	6 000	2
3.1	33 177 600	10 000	2
4	66 846 720	12 000	4
4.1	133 693 440	20 000	4
5	267 386 880	25 000	6
5.1	534 773 760	40 000	8
5.2	1 069 547 520	60 000	8
6	1 069 547 520	60 000	8
6.1	2 139 095 040	120 000	8
6.2	4 278 190 080	240 000	6

Table A.3 – Specification of CpbVclFactor, CpbNalFactor, FormatCapabilityFactor and MinCrScaleFactor

Profile	CpbVclFactor	CpbNalFactor	FormatCapabilityFactor	MinCrScaleFactor
Baseline	1 000	1 100	1.875	1.0
Main	1 000	1 100	1.875	1.0

Informative clause A.4.3 shows the effect of these limits on picture rates for several example picture formats.

A.4.3 Effect of level limits on picture rate for the video profiles (informative)

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

Informative Table A.4 and Table A.5 provide examples of maximum picture rates for the Baseline and Main profiles for various picture formats when MinCbSizeY is equal to 64.

Table A.4 – Maximum picture rates (pictures per second) at level 1 to 4.1 for some example picture sizes when MinCbSizeY is equal to 64

Level:				**1**	**2**	**2.1**	**3**	**3.1**	**4**	**4.1**
Max luma picture size (samples):				36 864	122 880	245 760	552 960	983 040	2 228 224	2 228 224
Max luma sample rate (samples/sec)				552 960	3 686 400	7 372 800	16 588 800	33 177 600	66 846 720	133 693 440
Format nickname	**Luma width**	**Luma height**	**Luma picture size**							
SQCIF	**128**	**96**	16 384	33.7	225.0	300.0	300.0	300.0	300.0	300.0
QCIF	**176**	**144**	36 864	15.0	100.0	200.0	300.0	300.0	300.0	300.0
QVGA	**320**	**240**	81 920	-	45.0	90.0	202.5	300.0	300.0	300.0
525 SIF	**352**	**240**	98 304	-	37.5	75.0	168.7	300.0	300.0	300.0
CIF	**352**	**288**	122 880	-	30.0	60.0	135.0	270.0	300.0	300.0
525 HHR	**352**	**480**	196 608	-	-	37.5	84.3	168.7	300.0	300.0
625 HHR	**352**	**576**	221 184	-	-	33.3	75.0	150.0	300.0	300.0
Q720p	**640**	**360**	245 760	-	-	30.0	67.5	135.0	272.0	300.0
VGA	**640**	**480**	327 680	-	-	-	50.6	101.2	204.0	300.0
525 4SIF	**704**	**480**	360 448	-	-	-	46.0	92.0	185.4	300.0
525 SD	**720**	**480**	393 216	-	-	-	42.1	84.3	170.0	300.0
4CIF	**704**	**576**	405 504	-	-	-	40.9	81.8	164.8	300.0
625 SD	**720**	**576**	442 368	-	-	-	37.5	75.0	151.1	300.0
480p (16:9)	**864**	**480**	458 752	-	-	-	36.1	72.3	145.7	291.4
SVGA	**800**	**600**	532 480	-	-	-	31.1	62.3	125.5	251.0
QHD	**960**	**540**	552 960	-	-	-	30.0	60.0	120.8	241.7
XGA	**1 024**	**768**	786 432	-	-	-	-	42.1	85.0	170.0
720p HD	**1 280**	**720**	983 040	-	-	-	-	33.7	68.0	136.0
4VGA	**1 280**	**960**	1 228 800	-	-	-	-	-	54.4	108.8
SXGA	**1 280**	**1 024**	1 310 720	-	-	-	-	-	51.0	102.0
525 16SIF	**1 408**	**960**	1 351 680	-	-	-	-	-	49.4	98.9
16CIF	**1 408**	**1 152**	1 622 016	-	-	-	-	-	41.2	82.4
4SVGA	**1 600**	**1 200**	1 945 600	-	-	-	-	-	34.3	68.7
1080 HD	**1 920**	**1 080**	2 088 960	-	-	-	-	-	32.0	64.0
2Kx1K	**2 048**	**1 024**	2 097 152	-	-	-	-	-	31.8	63.7
2Kx1080	**2 048**	**1 080**	2 228 224	-	-	-	-	-	30.0	60.0
4XGA	**2 048**	**1 536**	3 145 728	-	-	-	-	-	-	-
16VGA	**2 560**	**1 920**	4 915 200	-	-	-	-	-	-	-
3616x1536 (2.35:1)	**3 616**	**1 536**	5 603 328	-	-	-	-	-	-	-
3672x1536 (2.39:1)	**3 680**	**1 536**	5 701 632	-	-	-	-	-	-	-
3840x2160 (4*HD)	**3 840**	**2 160**	8 355 840	-	-	-	-	-	-	-
4Kx2K	**4 096**	**2 048**	8 388 608	-	-	-	-	-	-	-
4096x2160	**4 096**	**2 160**	8 912 896	-	-	-	-	-	-	-
4096x2304 (16:9)	**4 096**	**2 304**	9 437 184	-	-	-	-	-	-	-
7680x4320	**7 680**	**4 320**	33 423 360	-	-	-	-	-	-	-
8192x4096	**8 192**	**4 096**	33 554 432	-	-	-	-	-	-	-
8192x4320	**8 192**	**4 320**	35 651 584	-	-	-	-	-	-	-

Table A.5 – Maximum picture rates (pictures per second) at level 5 to 6.2 for some example picture sizes when MinCbSizeY is equal to 64

Level:				5	5.1	5.2	6	6.1	6.2
Max luma picture size (samples):				8 912 896	8 912 896	8 912 896	35 651 584	35 651 584	35 651 584
Max luma sample rate (samples/sec)				267 386 880	534 773 760	1 069 547 520	1 069 547 520	2 139 095 040	4 278 190 080
Format nickname	Luma width	Luma height	Luma picture size						
SQCIF	128	96	16 384	300.0	300.0	300.0	300.0	300.0	300.0
QCIF	176	144	36 864	300.0	300.0	300.0	300.0	300.0	300.0
QVGA	320	240	81 920	300.0	300.0	300.0	300.0	300.0	300.0
525 SIF	352	240	98 304	300.0	300.0	300.0	300.0	300.0	300.0
CIF	352	288	122 880	300.0	300.0	300.0	300.0	300.0	300.0
525 HHR	352	480	196 608	300.0	300.0	300.0	300.0	300.0	300.0
625 HHR	352	576	221 184	300.0	300.0	300.0	300.0	300.0	300.0
Q720p	640	360	245 760	300.0	300.0	300.0	300.0	300.0	300.0
VGA	640	480	327 680	300.0	300.0	300.0	300.0	300.0	300.0
525 4SIF	704	480	360 448	300.0	300.0	300.0	300.0	300.0	300.0
525 SD	720	480	393 216	300.0	300.0	300.0	300.0	300.0	300.0
4CIF	704	576	405 504	300.0	300.0	300.0	300.0	300.0	300.0
625 SD	720	576	442 368	300.0	300.0	300.0	300.0	300.0	300.0
480p (16:9)	864	480	458 752	300.0	300.0	300.0	300.0	300.0	300.0
SVGA	800	600	532 480	300.0	300.0	300.0	300.0	300.0	300.0
QHD	960	540	552 960	300.0	300.0	300.0	300.0	300.0	300.0
XGA	1 024	768	786 432	300.0	300.0	300.0	300.0	300.0	300.0
720p HD	1 280	720	983 040	272.0	300.0	300.0	300.0	300.0	300.0
4VGA	1 280	960	1 228 800	217.6	300.0	300.0	300.0	300.0	300.0
SXGA	1 280	1 024	1 310 720	204.0	300.0	300.0	300.0	300.0	300.0
525 16SIF	1 408	960	1 351 680	197.8	300.0	300.0	300.0	300.0	300.0
16CIF	1 408	1 152	1 622 016	164.8	300.0	300.0	300.0	300.0	300.0
4SVGA	1 600	1 200	1 945 600	137.4	274.8	300.0	300.0	300.0	300.0
1080 HD	1 920	1 080	2 088 960	128.0	256.0	300.0	300.0	300.0	300.0
2Kx1K	2 048	1 024	2 097 152	127.5	255.0	300.0	300.0	300.0	300.0
2Kx1080	2 048	1 080	2 228 224	120.0	240.0	300.0	300.0	300.0	300.0
4XGA	2 048	1 536	3 145 728	85.0	170.0	300.0	300.0	300.0	300.0
16VGA	2 560	1 920	4 915 200	54.4	108.8	217.6	217.6	300.0	300.0
3616x1536 (2.35:1)	3 616	1 536	5 603 328	47.7	95.4	190.8	190.8	300.0	300.0
3672x1536 (2.39:1)	3 680	1 536	5 701 632	46.8	93.7	187.5	187.5	300.0	300.0
3840x2160 (4*HD)	3 840	2 160	8 355 840	32.0	64.0	128.0	128.0	256.0	300.0
4Kx2K	4 096	2 048	8 388 608	31.8	63.7	127.5	127.5	255.0	300.0
4096x2160	4 096	2 160	8 912 896	30.0	60.0	120.0	120.0	240.0	300.0
4096x2304 (16:9)	4 096	2 304	9 437 184	-	-	-	113.3	226.6	300.0
4096x3072	4 096	3 072	12 582 912	-	-	-	85.0	170.0	300.0
7680x4320	7 680	4 320	33 423 360	-	-	-	32.0	64.0	128.0
8192x4096	8 192	4 096	33 554 432	-	-	-	31.8	63.7	127.5
8192x4320	8 192	4 320	35 651 584	-	-	-	30.0	60.0	120.0

The following should be noted in regard to the examples shown in Table A.4 and Table A.5:

– This is a variable-picture-size Specification. The specific listed picture sizes are illustrative examples only.

– The example luma picture sizes were computed by rounding up the luma width and luma height to multiples of 64 before computing the product of these quantities, to reflect the potential use of MinCbSizeY equal to 64 for these picture sizes, as pic_width_in_luma_samples and pic_height_in_luma_samples are each required to be a multiple of

MinCbSizeY. For some illustrated values of luma width and luma height, a somewhat higher number of pictures per second can be supported when MinCbSizeY is less than 64.

– In cases where the maximum picture rate value is not an integer multiple of 0.1 pictures per second, the given maximum picture rate values have been rounded down to the largest integer multiple of 0.1 frames per second that does not exceed the exact value. For example, for level 3.1, the maximum picture rate for 720p HD has been rounded down to 33.7 from an exact value of 33.75.

– As used in the examples, "525" refers to typical use for environments using 525 analogue scan lines (of which approximately 480 lines contain the visible picture region) and "625" refers to environments using 625 analogue scan lines (of which approximately 576 lines contain the visible picture region).

– XGA is also known as (aka) XVGA, 4SVGA aka UXGA, 16XGA aka 4Kx3K, CIF aka 625 SIF, 625 HHR aka 2CIF aka half 625 D-1, aka half 625 ITU-R BT.601, 525 SD aka 525 D-1 aka 525 ITU-R BT.601, 625 SD aka 625 D-1 aka 625 ITU-R BT.601.

Annex B

Byte stream format

(This annex forms an integral part of this International Standard.)

B.1 General

This annex specifies syntax and semantics of a byte stream format specified for use by applications that deliver some or all of the NAL unit stream as an ordered stream of bytes. For bit-oriented delivery, the bit order for the byte stream format is specified to start with the MSB of the first byte, proceed to the LSB of the first byte, followed by the MSB of the second byte, etc.

The byte stream format consists of a sequence of byte stream NAL unit syntax structures. Each byte stream NAL unit syntax structure contains one 4-byte length indication followed by one nal_unit(NumBytesInNalUnit) syntax structure.

B.2 Byte stream NAL unit syntax and semantics

B.2.1 Byte stream NAL unit syntax

byte_stream_nal_unit( ) {	**Descriptor**
nal_unit_length	f(32)
nal_unit(nal_unit_length)	
}	

B.2.2 Byte stream NAL unit semantics

The order of byte stream NAL units in the byte stream shall follow the decoding order of the NAL units contained in the byte stream NAL units (see clause 7.4.2.3). The content of each byte stream NAL unit is associated with the same access unit as the NAL unit contained in the byte stream NAL unit (see clause 7.4.2.3.3).

nal_unit_length is a 4-byte length field indicating the length of the NAL unit within the nal_unit( ) syntax structure.

Annex C

Hypothetical reference decoder

(This annex forms an integral part of this International Standard.)

C.1 General

This annex specifies the hypothetical reference decoder (HRD) and its use to check bitstream and decoder conformance.

Two types of bitstreams are subject to HRD conformance checking for this International Standard. The first such type of bitstream, called Type I bitstream, is a NAL unit stream containing only the VCL NAL units and filler data NAL units for all access units in the bitstream. The second type of bitstream called a Type II bitstream, contains, in addition to the VCL NAL units and filler data NAL units for all access units in the bitstream, at least one of the following.

- additional non-VCL NAL units other than filler data NAL units
- all leading_zero_8bits, zero_byte, start_code_prefix_one_3bytes, and trailing_zero_8bits syntax elements that form a byte stream from the NAL unit stream (as specified in Annex B)

Figure C-1 shows the types of bitstream conformance points checked by the HRD.

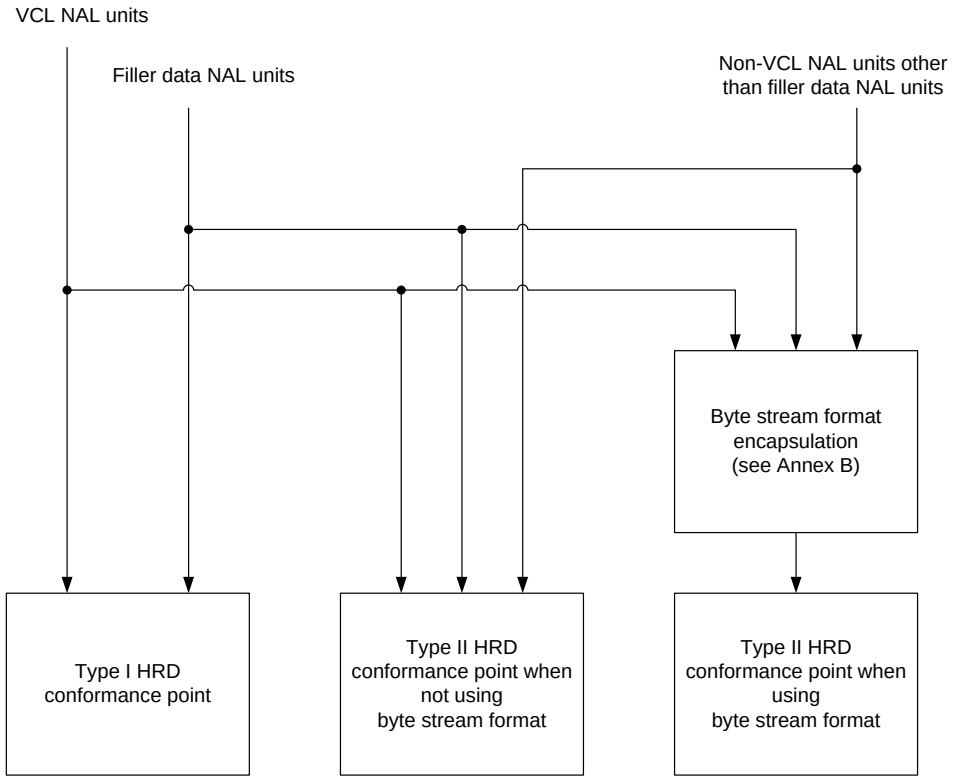

Figure C-1 – Structure of byte streams and NAL unit streams for HRD conformance checks

The syntax elements of non-VCL NAL units (or their default values for some of the syntax elements), required for the HRD, are specified in the semantic subclauses of clause 7 and Annexes D and E.

Two types of HRD parameter sets are used. The HRD parameter sets are signalled through video usability information as specified in subclauses E.2 and E.3, which is part of the sequence parameters set syntax structure.

In order to check conformance of a bitstream using the HRD, all SPSs and picture parameters sets referred to in the VCL NAL units, and corresponding buffering period and picture timing SEI messages shall be conveyed to the HRD, in a timely manner, either in the bitstream (by non-VCL NAL units), or by other means not specified in this International Standard.

In Annexes C, D and E, the specification for "presence" of non-VCL NAL units is also satisfied when those NAL units (or just some of them) are conveyed to decoders (or to the HRD) by other means not specified by this International Standard. For the purpose of counting bits, only the appropriate bits that are actually present in the bitstream are counted.

NOTE 1 – As an example, synchronization of a non-VCL NAL unit, conveyed by means other than presence in the bitstream, with the NAL units that are present in the bitstream, can be achieved by indicating two points in the bitstream, between which the non-VCL NAL unit would have been present in the bitstream, had the encoder decided to convey it in the bitstream.

When the content of a non-VCL NAL unit is conveyed for the application by some means other than presence within the bitstream, the representation of the content of the non-VCL NAL unit is not required to use the same syntax specified in this annex.

NOTE 2 – When HRD information is contained within the bitstream, it is possible to verify the conformance of a bitstream to the requirements of this subclause based solely on information contained in the bitstream. When the HRD information is not present in the bitstream, as is the case for all "stand-alone" Type I bitstreams, conformance can only be verified when the HRD data is supplied by some other means not specified in this International Standard.

The HRD contains a coded picture buffer (CPB), an instantaneous decoding process, a decoded picture buffer (DPB), and output cropping as shown in Figure C-2.

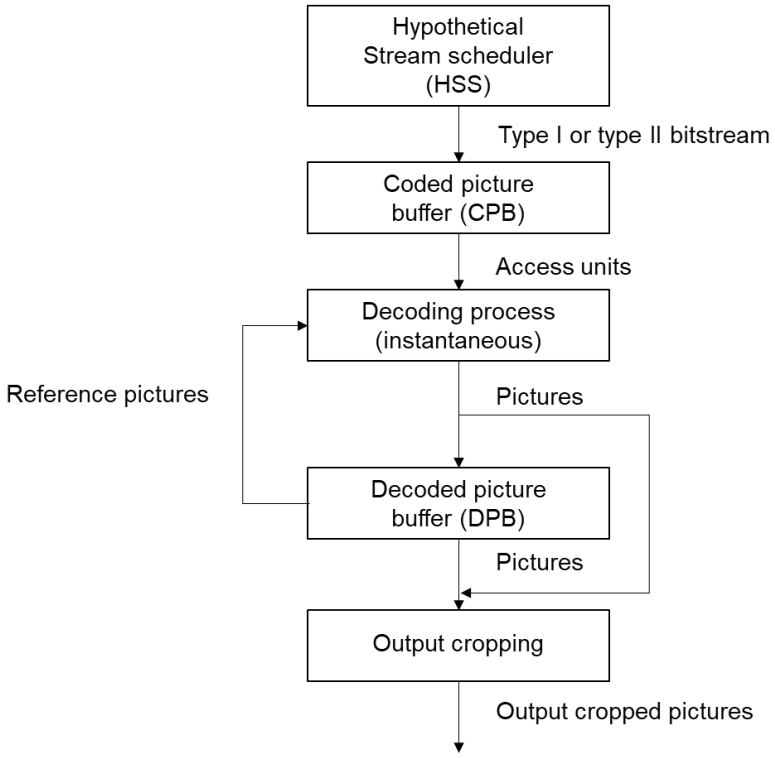

Figure C-2 – HRD buffer model

The CPB size (number of bits) is CpbSize[SchedSelIdx]. The DPB size (number of picture storage buffers) is Max(1, max_dec_pic_buffering).

The HRD operates as follows. Data associated with access units that flow into the CPB according to a specified arrival schedule are delivered by the HSS. The data associated with each access unit are removed and decoded instantaneously by the instantaneous decoding process at CPB removal times. Each decoded picture is placed in the DPB at its CPB removal time unless it is output at its CPB removal time and is a non-reference picture. When a picture is placed in the DPB it is removed from the DPB at the later of the DPB output time or the time that it is marked as "unused for reference".

The operation of the CPB is specified in subclause C.2. The instantaneous decoder operation is specified in clauses 8 and 9. The operation of the DPB is specified in subclause C.3. The output cropping is specified in subclause C.3.2.

HSS and HRD information concerning the number of enumerated delivery schedules and their associated bit rates and buffer sizes is specified in subclauses E.2.1, E.2.2, E.3.1 and E.3.2. The HRD is initialised as specified by the buffering period SEI message as specified in subclauses D.2.2 and D.3.2. The removal timing of access units from the CPB and

output timing from the DPB are specified in the picture timing SEI message as specified in subclauses D.2.3 and D.3.3. All timing information relating to a specific access unit shall arrive prior to the CPB removal time of the access unit.

The HRD is used to check conformance of bitstreams and decoders as specified in subclauses C.4 and C.5, respectively.

NOTE 3 – While conformance is guaranteed under the assumption that all frame-rates and clocks used to generate the bitstream match exactly the values signalled in the bitstream, in a real system each of these may vary from the signalled or specified value.

All the arithmetic in this annex is done with real values, so that no rounding errors can propagate. For example, the number of bits in a CPB just prior to or after removal of an access unit is not necessarily an integer.

The variable t_c is derived as follows and is called a clock tick.

$$t_c = num\_units\_in\_tick \div time\_scale \quad (C\text{-}1)$$

The following is specified for expressing the constraints in this Annex.

– Let access unit n be the n-th access unit in decoding order with the first access unit being access unit 0.

– Let picture n be the coded picture or the decoded picture of access unit n.

C.2 Operation of coded picture buffer (CPB)

C.2.1 General

The specifications in this subclause apply independently to each set of CPB parameters that is present and to both the Type I and Type II conformance points shown in Figure C-1.

C.2.2 Timing of bitstream arrival

The HRD may be initialised at any one of the buffering period SEI messages. Prior to initialisation, the CPB is empty.

NOTE – After initialisation, the HRD is not initialised again by subsequent buffering period SEI messages.

Each access unit is referred to as access unit n, where the number n identifies the particular access unit. The access unit that is associated with the buffering period SEI message that initializes the CPB is referred to as access unit 0. The value of n is incremented by 1 for each subsequent access unit in decoding order.

The time at which the first bit of access unit n begins to enter the CPB is referred to as the initial arrival time $t_{ai}(n)$.

The initial arrival time of access units is derived as follows.

– If the access unit is access unit 0, $t_{ai}(0) = 0$,

– Otherwise (the access unit is access unit n with n > 0), the following applies.

  – If cbr_flag[SchedSelIdx] is equal to 1, the initial arrival time for access unit n, is equal to the final arrival time (which is derived below) of access unit n - 1, i.e.

$$t_{ai}(n) = t_{af}(n - 1) \quad (C\text{-}2)$$

  – Otherwise (cbr_flag[SchedSelIdx] is equal to 0), the initial arrival time for access unit n is derived by

$$t_{ai}(n) = Max(t_{af}(n - 1), t_{ai,earliest}(n)) \quad (C\text{-}3)$$

    where $t_{ai,earliest}(n)$ is derived as follows

    – If access unit n is not the first access unit of a subsequent buffering period, $t_{ai,earliest}(n)$ is derived as

$$t_{ai,earliest}(n) = t_{r,n}(n) - (initial\_cpb\_removal\_delay[SchedSelIdx] + initial\_cpb\_removal\_delay\_offset[SchedSelIdx]) \div 90000 \quad (C\text{-}4)$$

      with $t_{r,n}(n)$ being the nominal removal time of access unit n from the CPB as specified in subclause C.1.2 and initial_cpb_removal_delay[SchedSelIdx] and initial_cpb_removal_delay_offset[SchedSelIdx] being specified in the previous buffering period SEI message.

    – Otherwise (access unit n is the first access unit of a subsequent buffering period), $t_{ai,earliest}(n)$ is derived as

$$t_{ai,earliest}(n) = t_{r,n}(n) - (initial\_cpb\_removal\_delay[SchedSelIdx] \div 90000) \quad (C\text{-}5)$$

with initial_cpb_removal_delay[SchedSelIdx] being specified in the buffering period SEI message associated with access unit n.

The final arrival time for access unit n is derived by

$$t_{af}(n) = t_{ai}(n) + b(n) \div BitRate[SchedSelIdx] \quad (C\text{-}6)$$

where b(n) is the size in bits of access unit n, counting the bits of the VCL NAL units and the filler data NAL units for the Type I conformance point or all bits of the Type II bitstream for the Type II conformance point, where the Type I and Type II conformance points are as shown in Figure C-1.

The values of SchedSelIdx, BitRate[SchedSelIdx], and CpbSize[SchedSelIdx] are constrained as follows.

– If access unit n and access unit n - 1 are part of different coded video sequences and the content of the active SPSs of the two coded video sequences differ, the HSS selects a value SchedSelIdx1 of SchedSelIdx from among the values of SchedSelIdx provided for the coded video sequence containing access unit n that results in a BitRate[SchedSelIdx1] or CpbSize[SchedSelIdx1] for the second of the two coded video sequences (which contains access unit n). The value of BitRate[SchedSelIdx1] or CpbSize[SchedSelIdx1] may differ from the value of BitRate[SchedSelIdx0] or CpbSize[SchedSelIdx0] for the value SchedSelIdx0 of SchedSelIdx that was in use for the coded video sequence containing access unit n - 1.

– Otherwise, the HSS continues to operate with the previous values of SchedSelIdx, BitRate[SchedSelIdx] and CpbSize[SchedSelIdx].

When the HSS selects values of BitRate[SchedSelIdx] or CpbSize[SchedSelIdx] that differ from those of the previous access unit, the following applies.

– the variable BitRate[SchedSelIdx] comes into effect at time $t_{ai}(n)$

– the variable CpbSize[SchedSelIdx] comes into effect as follows.

  – If the new value of CpbSize[SchedSelIdx] exceeds the old CPB size, it comes into effect at time $t_{ai}(n)$,

  – Otherwise, the new value of CpbSize[SchedSelIdx] comes into effect at the time $t_r(n)$.

C.2.3 Timing of coded picture removal

For access unit 0, the nominal removal time of the access unit from the CPB is specified by

$$t_{r,n}(0) = initial\_cpb\_removal\_delay[SchedSelIdx] \div 90000 \quad (C\text{-}7)$$

For the first access unit of a buffering period that does not initialise the HRD, the nominal removal time of the access unit from the CPB is specified by

$$t_{r,n}(n) = t_{r,n}(n_b) + t_c * cpb\_removal\_delay(n) \quad (C\text{-}8)$$

where $t_{r,n}(n_b)$ is the nominal removal time of the first access unit of the previous buffering period and cpb_removal_delay(n) is the value of cpb_removal_delay specified in the picture timing SEI message associated with access unit n.

When an access unit n is the first access unit of a buffering period, n_b is set equal to n at the removal time of access unit n.

The nominal removal time $t_{r,n}(n)$ of an access unit n that is not the first access unit of a buffering period is given by

$$t_{r,n}(n) = t_{r,n}(n_b) + t_c * cpb\_removal\_delay(n) \quad (C\text{-}9)$$

where $t_{r,n}(n_b)$ is the nominal removal time of the first access unit of the current buffering period and cpb_removal_delay(n) is the value of cpb_removal_delay specified in the picture timing SEI message associated with access unit n.

The removal time of access unit n is specified as follows.

– If low_delay_hrd_flag is equal to 0 or $t_{r,n}(n) >= t_{af}(n)$, the removal time of access unit n is specified by

$$t_r(n) = t_{r,n}(n) \quad (C\text{-}10)$$

– Otherwise (low_delay_hrd_flag is equal to 1 and $t_{r,n}(n) < t_{af}(n)$), the removal time of access unit n is specified by

$$t_r(n) = t_{r,n}(n) + t_c * Ceil((t_{af}(n) - t_{r,n}(n)) \div t_c) \quad \text{(C-11)}$$

NOTE – The latter case indicates that the size of access unit n, b(n), is so large that it prevents removal at the nominal removal time.

C.3 Operation of the decoded picture buffer (DPB)

C.3.1 General

The decoded picture buffer contains picture storage buffers. Each of the picture storage buffers may contain a decoded picture that is marked as "used for reference" or is held for future output. Prior to initialisation, the DPB is empty (the DPB fullness is set to zero). The following steps of the subclauses of this subclause all happen instantaneously at $t_r(n)$ and in the sequence listed.

C.3.2 Removal of pictures from the DPB

The removal of pictures from the DPB proceeds as follows.

– If the current picture is an IDR picture, the following applies:
  – All reference pictures in the DPB are marked as "unused for reference" as specifid in subclause 8.3.3.
  – When the IDR picture is not the first IDR picture decoded and the value of PicWidthInCtbsY, PicHeightInCtbsY, or max_dec_pic_buffering derived from the active SPS is different from the value of PicWidthInCtbsY, PicHeightInCtbsY, or max_dec_pic_buffering derived from the SPS that was active for the preceding sequence, respectively, no_output_of_prior_pics_flag is inferred to be equal to 1 by the HRD, regardless of the actual value of no_output_of_prior_pics_flag.

    NOTE – Decoder implementations should try to handle frame or DPB size changes more gracefully than the HRD in regard to changes in PicWidthInCtbsY or PicHeightInCtbsY.
  – When no_output_of_prior_pics_flag is equal to 1 or is inferred to be equal to 1, all picture storage buffers in the DPB are emptied without output of the pictures they contain, and DPB fullness is set to 0.

– Otherwise (the current picture is not an IDR picture), the decoded reference picture marking process specified in subclause 8.3.3 is invoked.

All pictures m in the DPB, for which all of the following conditions are true, are removed from the DPB.

– picture m is marked as "unused for reference".

– picture m's DPB output time is less than or equal to the CPB removal time of the current picture n; i.e., $t_{o,dpb}(m) <= t_r(n)$.

When a picture in a picture storage buffer is removed from the DPB, the DPB fullness is decremented by one.

C.3.3 Picture decoding and output

Picture n is decoded and its DPB output time $t_{o,dpb}(n)$ is derived by

$$t_{o,dpb}(n) = t_r(n) + t_c * dpb\_output\_delay(n) \quad \text{(C-12)}$$

The output of the current picture is specified as follows.

– If $t_{o,dpb}(n) = t_r(n)$, the current picture is output.

– Otherwise ($t_{o,dpb}(n) > t_r(n)$), the current picture is output later and will be stored in the DPB (as specified in subclause C.2.4) and is output at time $t_{o,dpb}(n)$ unless indicated not to be output by the decoding or inference of no_output_of_prior_pics_flag equal to 1 at a time that precedes $t_{o,dpb}(n)$.

The output picture shall be cropped, using the cropping rectangle specified in the SPS for the sequence.

When picture n is a picture that is output and is not the last picture of the bitstream that is output, the value of $\Delta t_{o,dpb}(n)$ is defined as:

$$\Delta t_{o,dpb}(n) = t_{o,dpb}(n_n) - t_{o,dpb}(n) \quad \text{(C-13)}$$

where n_n indicates the picture that follows after picture n in output order.

The decoded picture is stored in the DPB.

C.3.4 Current decoded picture marking and storage

The current decoded picture is stored in the DPB in an empty picture storage buffer, the DPB fullness is incremented by one, and the current picture is marked as "used for short-term reference".

C.4 Bitstream conformance

A bitstream of coded data conforming to this International Standard fulfils the following requirements.

The bitstream is constructed according to the syntax, semantics, and constraints specified in this International Standard outside of this Annex.

The bitstream is tested by the HRD as specified below:

For Type I bitstreams, the number of tests carried out is equal to cpb_cnt_minus1 + 1 where cpb_cnt_minus1 is either the syntax element of hrd_parameters( ) following the vcl_hrd_parameters_present_flag or is determined by the application by other means not specified in this International Standard. One test is carried out for each bit rate and CPB size combination specified by hrd_parameters( ) following the vcl_hrd_parameters_present_flag. Each of these tests is conducted at the Type I conformance point shown in Figure C-1.

For Type II bitstreams there are two sets of tests. The number of tests of the first set is equal to cpb_cnt_minus1 + 1 where cpb_cnt_minus1 is either the syntax element of hrd_parameters( ) following the vcl_hrd_parameters_present_flag or is determined by the application by other means not specified in this International Standard. One test is carried out for each bit rate and CPB size combination. Each of these tests is conducted at the Type I conformance point shown in Figure C-1. For these tests, only VCL and filler data NAL units are counted for the input bit rate and CPB storage.

The number of tests of the second set, for Type II bitstreams, is equal to cpb_cnt_minus1 + 1 where cpb_cnt_minus1 is either the syntax element of hrd_parameters( ) following the nal_hrd_parameters_present_flag or is determined by the application by other means not specified in this International Standard. One test is carried out for each bit rate and CPB size combination specified by hrd_parameters( ) following the nal_hrd_parameters_present_flag. Each of these tests is conducted at the Type II conformance point shown in Figure C-1. For these tests, all NAL units (of a Type II NAL unit stream) or all bytes (of a byte stream) are counted for the input bit rate and CPB storage.

NOTE 1 – NAL HRD parameters established by a value of SchedSelIdx for the Type II conformance point shown in Figure C-1 are sufficient to also establish VCL HRD conformance for the Type I conformance point shown in Figure C-1 for the same values of initial_cpb_removal_delay[SchedSelIdx], BitRate[SchedSelIdx], and CpbSize[SchedSelIdx] for the VBR case (cbr_flag[SchedSelIdx] equal to 0). This is because the data flow into the Type I conformance point is a subset of the data flow into the Type II conformance point and because, for the VBR case, the CPB is allowed to become empty and stay empty until the time a next picture is scheduled to begin to arrive. For example, when NAL HRD parameters are provided for the Type II conformance point that not only fall within the bounds set for NAL HRD parameters for profile conformance in item j of subclause A.3.1 or item i of subclause A.3.3 (depending on the profile in use) but also fall within the bounds set for VCL HRD parameters for profile conformance in item i of subclause A.3.1 or item h of subclause A.3.3 (depending on the profile in use), conformance of the VCL HRD for the Type I conformance point is also assured to fall within the bounds of item i of subclause A.3.1.

For conforming bitstreams, all of the following conditions shall be fulfilled for each of the tests.

– For each access unit n, with n>0, associated with a buffering period SEI message, with $\Delta t_{g,90}(n)$ specified by

$$\Delta t_{g,90}(n) = 90000 * (t_{r,n}(n) - t_{af}(n - 1)) \quad \text{(C-14)}$$

The value of initial_cpb_removal_delay[SchedSelIdx] shall be constrained as follows.

– If cbr_flag[SchedSelIdx] is equal to 0,

$$\text{initial\_cpb\_removal\_delay[SchedSelIdx]} <= \text{Ceil}(\Delta t_{g,90}(n)) \quad \text{(C-15)}$$

– Otherwise (cbr_flag[SchedSelIdx] is equal to 1),

$$\text{Floor}(\Delta t_{g,90}(n)) <= \text{initial\_cpb\_removal\_delay[SchedSelIdx]} <= \text{Ceil}(\Delta t_{g,90}(n)) \quad \text{(C-16)}$$

NOTE 2 – The exact number of bits in the CPB at the removal time of each picture may depend on which buffering period SEI message is selected to initialize the HRD. Encoders must take this into account to ensure that all specified constraints must be obeyed regardless of which buffering period SEI message is selected to initialize the HRD, as the HRD may be initialised at any one of the buffering period SEI messages.

– A CPB overflow is specified as the condition in which the total number of bits in the CPB is larger than the CPB size. The CPB shall never overflow.

– A CPB underflow is specified as the condition in which tr,n(n) is less than taf(n). When low_delay_hrd_flag is equal to 0, the CPB shall never underflow.

– The nominal removal times of pictures from the CPB (starting from the second picture in decoding order), shall satisfy the constraints on tr,n(n) and tr(n) expressed in subclauses A.3.1 through A.3.3 for the profile and level specified in the bitstream.

– Immediately after any decoded picture is added to the DPB, the fullness of the DPB shall be less than or equal to the DPB size as constrained by Annexes A, D, and E for the profile and level specified in the bitstream.

– All reference pictures shall be present in the DPB when needed for prediction. Each picture shall be present in the DPB at its DPB output time unless it is not stored in the DPB at all, or is removed from the DPB before its output time by one of the processes specified in subclause C.2.

– The value of $\Delta_{to,dpb}(n)$ as given by Equation C-13, which is the difference between the output time of a picture and that of the picture immediately following it in output order, shall satisfy the constraint expressed in subclause A.3.1 for the profile and level specified in the bitstream.

C.5 Decoder conformance

C.5.1 General

A decoder conforming to this International Standard fulfils the following requirements.

A decoder claiming conformance to a specific profile and level shall be able to decode successfully all conforming bitstreams specified for decoder conformance in subclause C.3, provided that all SPSs and picture parameters sets referred to in the VCL NAL units, and appropriate buffering period and picture timing SEI messages are conveyed to the decoder, in a timely manner, either in the bitstream (by non-VCL NAL units), or by external means not specified by this International Standard.

There are two types of conformance that can be claimed by a decoder: output timing conformance and output order conformance.

To check conformance of a decoder, test bitstreams conforming to the claimed profile and level, as specified by subclause C.3 are delivered by a hypothetical stream scheduler (HSS) both to the HRD and to the decoder under test (DUT). All pictures output by the HRD shall also be output by the DUT and, for each picture output by the HRD, the values of all samples that are output by the DUT for the corresponding picture shall be equal to the values of the samples output by the HRD.

For output timing decoder conformance, the HSS operates as described above, with delivery schedules selected only from the subset of values of SchedSelIdx for which the bit rate and CPB size are restricted as specified in Annex A for the specified profile and level, or with "interpolated" delivery schedules as specified below for which the bit rate and CPB size are restricted as specified in Annex A. The same delivery schedule is used for both the HRD and DUT.

When the HRD parameters and the buffering period SEI messages are present with cpb_cnt_minus1 greater than 0, the decoder shall be capable of decoding the bitstream as delivered from the HSS operating using an "interpolated" delivery schedule specified as having peak bit rate r, CPB size c(r), and initial CPB removal delay (f(r) ÷ r) as follows

$$\alpha = (r - BitRate[SchedSelIdx - 1]) \div (BitRate[SchedSelIdx] – BitRate[SchedSelIdx - 1]), \quad \text{(C-17)}$$

$$c(r) = \alpha * CpbSize[SchedSelIdx] + (1 – \alpha) * CpbSize[SchedSelIdx-1], \quad \text{(C-18)}$$

$$f(r) = \alpha * initial\_cpb\_removal\_delay[SchedSelIdx] * BitRate[SchedSelIdx] + (1 – \alpha) * initial\_cpb\_removal\_delay[SchedSelIdx - 1] * BitRate[SchedSelIdx - 1] \quad \text{(C-19)}$$

for any SchedSelIdx > 0 and r such that BitRate[SchedSelIdx - 1] <= r <= BitRate[SchedSelIdx] such that r and c(r) are within the limits as specified in Annex A for the maximum bit rate and buffer size for the specified profile and level.

NOTE 1 – initial_cpb_removal_delay[SchedSelIdx] can be different from one buffering period to another and have to be re-calculated.

For output timing decoder conformance, an HRD as described above is used and the timing (relative to the delivery time of the first bit) of picture output is the same for both HRD and the DUT up to a fixed delay.

For output order decoder conformance, the HSS delivers the bitstream to the DUT "by demand" from the DUT, meaning that the HSS delivers bits (in decoding order) only when the DUT requires more bits to proceed with its processing.

NOTE 2 – This means that for this test, the coded picture buffer of the DUT could be as small as the size of the largest access unit.

A modified HRD as described below is used, and the HSS delivers the bitstream to the HRD by one of the schedules specified in the bitstream such that the bit rate and CPB size are restricted as specified in Annex A. The order of pictures output shall be the same for both HRD and the DUT.

For output order decoder conformance, the HRD CPB size is equal to CpbSize[SchedSelIdx] for the selected schedule and the DPB size is equal to MaxDpbSize. Removal time from the CPB for the HRD is equal to final bit arrival time and decoding is immediate. The operation of the DPB of this HRD is described below.

C.5.2 Operation of the output order DPB

C.5.2.1 General

The decoded picture buffer contains picture storage buffers. Each of the picture storage buffers may contain a decoded picture that is marked as "used for reference" or is held for future output. Prior to initialisation, the DPB is empty (the DPB fullness is set to zero). The following steps all happen instantaneously when an access unit is removed from the CPB, and in the order listed.

C.5.2.2 Removal of pictures from the DPB

The removal of pictures from the DPB proceeds as follows.

– If the decoded picture is an IDR picture the following applies.
  – All reference pictures in the DPB are marked as "unused for reference" as specified in subclause 8.3.3.
  – When the IDR picture is not the first IDR picture decoded and the value of PicWidthInCtbsY, PicHeightInCtbsY, or max_dec_pic_buffering derived from the active SPS is different from the value of PicWidthInCtbsY, PicHeightInCtbsY, or max_dec_pic_buffering derived from the SPS that was active for the preceding sequence, respectively, no_output_of_prior_pics_flag is inferred to be equal to 1 by the HRD, regardless of the actual value of no_output_of_prior_pics_flag.

    NOTE – Decoder implementations should try to handle changes in the value of PicWidthInCtbsY, PicHeightInCtbsY, or max_dec_pic_buffering more gracefully than the HRD.

  – When no_output_of_prior_pics_flag is equal to 1 or is inferred to be equal to 1, all picture storage buffers in the DPB are emptied without output of the pictures they contain, and DPB fullness is set to 0.

– Otherwise (the current picture is not an IDR picture), the decoded reference picture marking process is invoked as specified in subclause 8.3.3. When there is no empty picture storage buffer (i.e., DPB fullness is equal to DPB size), the "bumping" process specified in subclause C.5.2.4 is invoked repeatedly until there is an empty picture storage buffer. Picture storage buffers containing pictures that are marked as "not needed for output" and "unused for reference" are emptied (without output), and the DPB fullness is decremented by the number of picture storage buffers emptied.

When the current picture is an IDR picture for which no_output_of_prior_pics_flag is not equal to 1 and is not inferred to be equal to 1, the following two steps are performed.

1. Picture storage buffers containing pictures that are marked as "not needed for output" and "unused for reference" are emptied (without output), and the DPB fullness is decremented by the number of picture storage buffers emptied.
2. All non-empty picture storage buffers in the DPB are emptied by repeatedly invoking the “bumping” process specified in subclause C.5.2.4, and the DPB fullness is set to 0.

C.5.2.3 Current picture decoding, storage, and marking

The current picture is decoded, stored in an empty picture storage buffer in the DPB, and marked as "needed for output" and as "used for short-term reference".

C.5.2.4 "Bumping" process

The "bumping" process is invoked in the following cases.

– The current picture is an IDR picture and no_output_of_prior_pics_flag is not equal to 1 and is not inferred to be equal to 1, as specified in subclause C.5.2.2.
– There is no empty picture storage buffer (i.e., DPB fullness is equal to DPB size) and an empty picture storage buffer is needed for storage of a decoded picture, as specified in subclause .

The "bumping" process consists of the following ordered steps:

1. The picture that is first for output is selected as the one having the smallest value of PicOrderCntVal of all pictures in the DPB marked as "needed for output".
2. The picture is cropped, using the conformance cropping window specified in the active SPS for the picture, the cropped picture is output, and the picture is marked as "not needed for output".
3. When the picture storage buffer that included the picture that was cropped and output contains a picture marked as "unused for reference", the picture storage buffer is emptied.

Annex D

Supplemental enhancement information

(This annex forms an integral part of this International Standard.)

D.1 General

This annex specifies syntax and semantics for SEI message payloads.

SEI messages assist in processes related to decoding, display or other purposes. However, SEI messages are not required for constructing the luma or chroma samples by the decoding process. Conforming decoders are not required to process this information for output order conformance to this document (see Annex C for the specification of conformance). Some SEI message information is required to check bitstream conformance and for output timing decoder conformance.

In Annex C including its subclauses, specification for presence of SEI messages are also satisfied when those messages (or some subset of them) are conveyed to decoders (or to the HRD) by other means not specified in this document. When present in the bitstream, SEI messages shall obey the syntax and semantics specified in clause 7.3.5 and this annex. When the content of an SEI message is conveyed for the application by some means other than presence within the bitstream, the representation of the content of the SEI message is not required to use the same syntax specified in this annex. For the purpose of counting bits, only the appropriate bits that are actually present in the bitstream are counted.

D.2 SEI payload syntax

D.2.1 General SEI message syntax

sei_payload(payloadType, payloadSize) {	**Descriptor**
if(payloadType = = 0)	
buffering_period(payloadSize)	
else if(payloadType = = 1)	
pic_timing(payloadSize)	
esel if(payloadType = = 6)	
recovery_point(payloadSize)	
else if(payloadType = = 137)	
mastering_display_colour_volume(payloadSize)	
else if(payloadType = = 144)	
content_light_level_info(payloadSize)	
else	
reserved_sei_message(payloadSize)	
if(more_data_in_payload( )) {	
if(payload_extension_present( ))	
reserved_payload_extension_data	u(v)
payload_bit_equal_to_one /* equal to 1 */	f(1)
while(!byte_aligned( ))	
payload_bit_equal_to_zero /* equal to 0 */	f(1)
}	
}	

D.2.2 Buffering period SEI message syntax

buffering_period(payloadSize) {	**Descriptor**
seq_parameter_set_id	ue(v)
if(NalHrdBpPresentFlag) {	
for(SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++) {	
initial_cpb_removal_delay[SchedSelIdx **]**	u(v)
initial_cpb_removal_delay_offset[SchedSelIdx **]**	u(v)
}	
}	
if(VclHrdBpPresentFlag) {	
for(SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++) {	
initial_cpb_removal_delay[SchedSelIdx **]**	u(v)
initial_cpb_removal_delay_offset[SchedSelIdx **]**	u(v)
}	
}	
}	

D.2.3 Picture timing SEI message syntax

pic_timing(payloadSize) {	**Descriptor**
if(CpbDpbDelaysPresentFlag) {	
cpb_removal_delay	u(v)
dpb_output_delay	u(v)
}	
if(pic_struct_present_flag) {	
pic_struct	u(4)
for(i = 0; i < NumClockTS ; i++) {	
clock_timestamp_flag[i]	u(1)
if(clock_timestamp_flag[i]) {	
ct_type	u(2)
nuit_field_based_flag	u(1)
counting_type	u(5)
full_timestamp_flag	u(1)
discontinuity_flag	u(1)
cnt_dropped_flag	u(1)
n_frames	u(8)
if(full_timestamp_flag) {	
seconds_value /* 0..59 */	u(6)
minutes_value /* 0..59 */	u(6)
hours_value /* 0..23 */	u(5)
} else {	
seconds_flag	u(1)
if(seconds_flag) {	

seconds_value /* range 0..59 */	
minutes_flag	
if(minutes_flag) {	
minutes_value /* 0..59 */	
hours_flag	
if(hours_flag)	
hours_value /* 0..23 */	
}	
}	
}	
if(time_offset_length > 0)	
time_offset	i(v)
}	
}	
}	
}	

D.2.4 Recovery point SEI message syntax

recovery_point(payloadSize) {	**Descriptor**
recovery_poc_cnt	se(v)
exact_match_flag	u(1)
broken_link_flag	u(1)
}	

D.2.5 Mastering display colour volume SEI message syntax

mastering_display_colour_volume(payloadSize) {	**Descriptor**
for(c = 0; c < 3; c++) {	
display_primaries_x[c]	u(16)
display_primaries_y[c]	u(16)
}	
white_point_x	u(16)
white_point_y	u(16)
max_display_mastering_luminance	u(32)
min_display_mastering_luminance	u(32)
}	

D.2.6 Content light level information SEI message syntax

content_light_level_info(payloadSize) {	Descriptor
max_content_light_level	u(16)
max_pic_average_light_level	u(16)
}	

D.3 SEI payload semantics

D.3.1 General SEI payload semantics

reserved_payload_extension_data shall not be present in bitstreams conforming to this version of this document. However, decoders conforming to this version of this document shall ignore the presence and value of reserved_payload_extension_data. When present, the length, in bits, of reserved_payload_extension_data is equal to 8 * payloadSize − nEarlierBits − nPayloadZeroBits − 1, where nEarlierBits is the number of bits in the sei_payload( ) syntax structure that precede the reserved_payload_extension_data syntax element and nPayloadZeroBits is the number of payload_bit_equal_to_zero syntax elements at the end of the sei_payload( ) syntax structure.

payload_bit_equal_to_one shall be equal to 1.

payload_bit_equal_to_zero shall be equal to 0.

The semantics and persistence scope for each SEI message are specified in the semantics specification for each particular SEI message.

NOTE – Persistence information for SEI messages is informatively summarized in Table D.1.

Table D.1 – Persistence scope of SEI messages (informative)

SEI message	Persistence scope
Buffering period	The remainder of the bitstream
Picture timing	The access unit containing the SEI message
Recovery point	Specified by the syntax of the SEI message

D.3.2 Buffering period SEI message semantics

When NalHrdBpPresentFlag or VclHrdBpPresentFlag are equal to 1, a buffering period SEI message can be associated with any access unit in the bitstream, and a buffering period SEI message shall be associated with each IDR access unit and with each access unit associated with a recovery point SEI message.

NOTE – For some applications, the frequent presence of a buffering period SEI message may be desirable.

A buffering period is specified as the set of access units between two instances of the buffering period SEI message in decoding order.

seq_parameter_set_id specifies the SPS that contains the sequence HRD attributes. The value of seq_parameter_set_id shall be equal to the value of seq_parameter_set_id in the PPS referenced by the coded picture associated with the buffering period SEI message. The value of seq_parameter_set_id shall be in the range of 0 to 31, inclusive.

initial_cpb_removal_delay[SchedSelIdx] specifies the delay for the SchedSelIdx-th CPB between the time of arrival in the CPB of the first bit of the coded data associated with the access unit associated with the buffering period SEI message and the time of removal from the CPB of the coded data associated with the same access unit, for the first buffering period after HRD initialisation. The syntax element has a length in bits given by initial_cpb_removal_delay_length_minus1 + 1. It is in units of a 90 kHz clock. initial_cpb_removal_delay[SchedSelIdx] shall not be equal to 0 and shall not exceed 90000 * (CpbSize[SchedSelIdx] ÷ BitRate[SchedSelIdx]), the time-equivalent of the CPB size in 90 kHz clock units.

initial_cpb_removal_delay_offset[SchedSelIdx] is used for the SchedSelIdx-th CPB in combination with the cpb_removal_delay to specify the initial delivery time of coded access units to the CPB. initial_cpb_removal_delay_offset[SchedSelIdx] is in units of a 90 kHz clock. The initial_cpb_removal_delay_offset[SchedSelIdx] syntax element is a fixed length code whose length in bits is given by

initial_cpb_removal_delay_length_minus1 + 1. This syntax element is not used by decoders and is needed only for the delivery scheduler (HSS) specified in Annex C.

Over the entire coded video sequence, the sum of initial_cpb_removal_delay[SchedSelIdx] and initial_cpb_removal_delay_offset[SchedSelIdx] shall be constant for each value of SchedSelIdx.

D.3.3 Picture timing SEI message semantics

The presence of picture timing SEI message in the bitstream is specified as follows.

– If CpbDpbDelaysPresentFlag is equal to 1 or pic_struct_present_flag is equal to 1, one picture timing SEI message shall be present in every access unit of the coded video sequence.

– Otherwise (CpbDpbDelaysPresentFlag is equal to 0 and pic_struct_present_flag is equal to 0), no picture timing SEI messages shall be present in any access unit of the coded video sequence.

cpb_removal_delay specifies how many clock ticks (see subclause E.2.1) to wait after removal from the CPB of the access unit associated with the most recent buffering period SEI message before removing from the buffer the access unit data associated with the picture timing SEI message. This value is also used to calculate an earliest possible time of arrival of access unit data into the CPB for the HSS, as specified in Annex C. The syntax element is a fixed length code whose length in bits is given by cpb_removal_delay_length_minus1 + 1. The cpb_removal_delay is the remainder of a $2^{(\text{cpb\_removal\_delay\_length\_minus1} + 1)}$ counter.

The value of cpb_removal_delay for the first picture in the bitstream shall be equal to 0.

dpb_output_delay is used to compute the DPB output time of the picture. It specifies how many clock ticks to wait after removal of an access unit from the CPB before the decoded picture can be output from the DPB (see subclause C.2).

NOTE 1 – A picture is not removed from the DPB at its output time when it is still marked as "used for short-term reference" or "used for long-term reference".

NOTE 2 – Only one dpb_output_delay is specified for a decoded picture.

The size of the syntax element dpb_output_delay is given in bits by dpb_output_delay_length_minus1 + 1. When max_dec_pic_buffering is equal to 0, dpb_output_delay shall be equal to 0.

The output time derived from the dpb_output_delay of any picture that is output from an output timing conforming decoder as specified in subclause C.2 shall precede the output time derived from the dpb_output_delay of all pictures in any subsequent coded video sequence in decoding order.

The output time derived from the dpb_output_delay of the second field, in decoding order, of a complementary non-reference field pair shall exceed the output time derived from the dpb_output_delay of the first field of the same complementary non-reference field pair.

The picture output order established by the values of this syntax element shall be the same order as established by the values of PicOrderCnt( ) as specified by subclauses C.4.1 to C.4.5, except that when the two fields of a complementary reference field pair have the same value of PicOrderCnt( ), the two fields have different output times.

For pictures that are not output by the "bumping" process of subclause C.4.5 because they precede, in decoding order, an IDR picture with no_output_of_prior_pics_flag equal to 1 or inferred to be equal to 1, the output times derived from dpb_output_delay shall be increasing with increasing value of PicOrderCnt( ) relative to all pictures within the same coded video sequence.

pic_struct indicates whether a picture should be displayed as a frame or one or more fields, according to Table D.2. Frame doubling (pic_struct equal to 7) indicates that the frame should be displayed two times consecutively, and frame tripling (pic_struct equal to 8) indicates that the frame should be displayed three times consecutively.

NOTE 3 – Frame doubling can facilitate the display, for example, of 25p video on a 50p display and 29.97p video on a 59.94p display. Using frame doubling and frame tripling in combination on every other frame can facilitate the display of 23.98p video on a 59.94p display.

Table D.2 – Interpretation of pic_struct

Value	Indicated display of picture	Restrictions	NumClockTS
0	frame	field_pic_flag shall be 0	1
1	top field	field_pic_flag shall be 1, bottom_field_flag shall be 0	1
2	bottom field	field_pic_flag shall be 1, bottom_field_flag shall be 1	1
3	top field, bottom field, in that order	field_pic_flag shall be 0	2
4	bottom field, top field, in that order	field_pic_flag shall be 0	2
5	top field, bottom field, top field repeated, in that order	field_pic_flag shall be 0	3
6	bottom field, top field, bottom field repeated, in that order	field_pic_flag shall be 0	3
7	frame doubling	field_pic_flag shall be 0 fixed_frame_rate_flag shall be 1	2
8	frame tripling	field_pic_flag shall be 0 fixed_frame_rate_flag shall be 1	3
9..15	reserved		

NumClockTS is determined by pic_struct as specified in Table D.2. There are up to NumClockTS sets of clock timestamp information for a picture, as specified by clock_timestamp_flag[i] for each set. The sets of clock timestamp information apply to the field(s) or the frame(s) associated with the picture by pic_struct.

The contents of the clock timestamp syntax elements indicate a time of origin, capture, or alternative ideal display. This indicated time is computed as

$$\begin{aligned} \text{clockTimestamp} = &((hH * 60 + mM) * 60 + sS) * \text{time\_scale} + \\ &\text{nFrames} * (\text{num\_units\_in\_tick} * (1 + \text{nuit\_field\_based\_flag})) + \text{tOffset}, \end{aligned} \tag{D-1}$$

in units of clock ticks of a clock with clock frequency equal to time_scale Hz, relative to some unspecified point in time for which clockTimestamp is equal to 0. Output order and DPB output timing are not affected by the value of clockTimestamp. When two or more frames with pic_struct equal to 0 are consecutive in output order and have equal values of clockTimestamp, the indication is that the frames represent the same content and that the last such frame in output order is the preferred representation.

NOTE 4 – clockTimestamp time indications may aid display on devices with refresh rates other than those well-matched to DPB output times.

clock_timestamp_flag[i] equal to 1 indicates that a number of clock timestamp syntax elements are present and follow immediately. clock_timestamp_flag[i] equal to 0 indicates that the associated clock timestamp syntax elements are not present. When NumClockTS is greater than 1 and clock_timestamp_flag[i] is equal to 1 for more than one value of i, the value of clockTimestamp shall be non-decreasing with increasing value of i.

ct_type indicates the scan type (interlaced or progressive) of the source material as specified in Table D.3.

Two fields of a coded frame may have different values of ct_type.

When clockTimestamp is equal for two fields of opposite parity that are consecutive in output order, both with ct_type equal to 0 (progressive) or ct_type equal to 2 (unknown), the two fields are indicated to have come from the same original progressive frame. Two consecutive fields in output order shall have different values of clockTimestamp when the value of ct_type for either field is 1 (interlaced).

Table D.3 – Mapping of ct_type to source picture scan

Value	Original picture scan
0	progressive
1	interlaced
2	unknown
3	reserved

nuit_field_based_flag is used in calculating clockTimestamp, as specified in Equation D-1.

counting_type specifies the method of dropping values of the n_frames as specified in Table D.4.

Table D.4 – Definition of counting_type values

Value	Interpretation
0	no dropping of n_frames count values and no use of time_offset
1	no dropping of n_frames count values
2	dropping of individual zero values of n_frames count
3	dropping of individual MaxFPS-1 values of n_frames count
4	dropping of the two lowest (value 0 and 1) n_frames counts when seconds_value is equal to 0 and minutes_value is not an integer multiple of 10
5	dropping of unspecified individual n_frames count values
6	dropping of unspecified numbers of unspecified n_frames count values
7..31	reserved

full_timestamp_flag equal to 1 specifies that the n_frames syntax element is followed by seconds_value, minutes_value, and hours_value. full_timestamp_flag equal to 0 specifies that the n_frames syntax element is followed by seconds_flag.

discontinuity_flag equal to 0 indicates that the difference between the current value of clockTimestamp and the value of clockTimestamp computed from the previous clock timestamp in output order can be interpreted as the time difference between the times of origin or capture of the associated frames or fields. discontinuity_flag equal to 1 indicates that the difference between the current value of clockTimestamp and the value of clockTimestamp computed from the previous clock timestamp in output order should not be interpreted as the time difference between the times of origin or capture of the associated frames or fields. When discontinuity_flag is equal to 0, the value of clockTimestamp shall be greater than or equal to all values of clockTimestamp present for the preceding picture in DPB output order.

cnt_dropped_flag specifies the skipping of one or more values of n_frames using the counting method specified by counting_type.

n_frames specifies the value of nFrames used to compute clockTimestamp. n_frames shall be less than

$$MaxFPS = Ceil(time\_scale \div num\_units\_in\_tick) \quad \text{(D-2)}$$

NOTE 5 – n_frames is a frame-based counter. For field-specific timing indications, time_offset should be used to indicate a distinct clockTimestamp for each field.

When counting_type is equal to 2 and cnt_dropped_flag is equal to 1, n_frames shall be equal to 1 and the value of n_frames for the previous picture in output order shall not be equal to 0 unless discontinuity_flag is equal to 1.

NOTE 6 – When counting_type is equal to 2, the need for increasingly large magnitudes of tOffset in Equation D-1 when using fixed non-integer frame rates (e.g., 12.5 frames per second with time_scale equal to 25 and num_units_in_tick equal to 2 and nuit_field_based_flag equal to 0) can be avoided by occasionally skipping over the value n_frames equal to 0 when counting (e.g., counting n_frames from 0 to 12, then incrementing seconds_value and counting n_frames from 1 to 12, then incrementing seconds_value and counting n_frames from 0 to 12, etc.).

When counting_type is equal to 3 and cnt_dropped_flag is equal to 1, n_frames shall be equal to 0 and the value of n_frames for the previous picture in output order shall not be equal to MaxFPS – 1 unless discontinuity_flag is equal to 1.

NOTE 7 – When counting_type is equal to 3, the need for increasingly large magnitudes of tOffset in Equation D-1 when using fixed non-integer frame rates (e.g., 12.5 frames per second with time_scale equal to 25 and num_units_in_tick equal to 2 and nuit_field_based_flag equal to 0) can be avoided by occasionally skipping over the value n_frames equal to MaxFPS when counting (e.g., counting n_frames from 0 to 12, then incrementing seconds_value and counting n_frames from 0 to 11, then incrementing seconds_value and counting n_frames from 0 to 12, etc.).

When counting_type is equal to 4 and cnt_dropped_flag is equal to 1, n_frames shall be equal to 2 and the specified value of sS shall be zero and the specified value of mM shall not be an integer multiple of ten and n_frames for the previous picture in output order shall not be equal to 0 or 1 unless discontinuity_flag is equal to 1.

NOTE 8 – When counting_type is equal to 4, the need for increasingly large magnitudes of tOffset in Equation D-1 when using fixed non-integer frame rates (e.g., 30000÷1001 frames per second with time_scale equal to 60000 and num_units_in_tick equal to 1 001 and nuit_field_based_flag equal to 1) can be reduced by occasionally skipping over the value n_frames equal to MaxFPS when counting (e.g., counting n_frames from 0 to 29, then incrementing seconds_value and counting n_frames from 0 to 29, etc., until the seconds_value is zero and minutes_value is not an integer multiple of ten, then counting n_frames from 2 to 29, then incrementing seconds_value and counting n_frames from 0 to 29, etc.). This counting method is well known in industry and is often referred to as "NTSC drop-frame" counting.

When counting_type is equal to 5 or 6 and cnt_dropped_flag is equal to 1, n_frames shall not be equal to 1 plus the value of n_frames for the previous picture in output order modulo MaxFPS unless discontinuity_flag is equal to 1.

NOTE 9 – When counting_type is equal to 5 or 6, the need for increasingly large magnitudes of tOffset in Equation D-1 when using fixed non-integer frame rates can be avoided by occasionally skipping over some values of n_frames when counting. The specific values of n_frames that are skipped are not specified when counting_type is equal to 5 or 6.

seconds_flag equal to 1 specifies that seconds_value and minutes_flag are present when full_timestamp_flag is equal to 0. seconds_flag equal to 0 specifies that seconds_value and minutes_flag are not present.

seconds_value specifies the value of sS used to compute clockTimestamp. The value of seconds_value shall be in the range of 0 to 59, inclusive. When seconds_value is not present, the previous seconds_value in decoding order shall be used as sS to compute clockTimestamp.

minutes_flag equal to 1 specifies that minutes_value and hours_flag are present when full_timestamp_flag is equal to 0 and seconds_flag is equal to 1. minutes_flag equal to 0 specifies that minutes_value and hours_flag are not present.

minutes_value specifies the value of mM used to compute clockTimestamp. The value of minutes_value shall be in the range of 0 to 59, inclusive. When minutes_value is not present, the previous minutes_value in decoding order shall be used as mM to compute clockTimestamp.

hours_flag equal to 1 specifies that hours_value is present when full_timestamp_flag is equal to 0 and seconds_flag is equal to 1 and minutes_flag is equal to 1.

hours_value specifies the value of hH used to compute clockTimestamp. The value of hours_value shall be in the range of 0 to 23, inclusive. When hours_value is not present, the previous hours_value in decoding order shall be used as hH to compute clockTimestamp.

time_offset specifies the value of tOffset used to compute clockTimestamp. The number of bits used to represent time_offset shall be equal to time_offset_length. When time_offset is not present, the value 0 shall be used as tOffset to compute clockTimestamp.

D.3.4 Recovery point SEI message semantics

The recovery point SEI message assists a decoder in determining when the decoding process will produce acceptable pictures for display after the decoder initiates random access or after the encoder indicates a broken link in the CVS. When the decoding process is started with the access unit in decoding order associated with the recovery point SEI message, all decoded pictures at or subsequent to the recovery point in output order specified in this SEI message are indicated to be correct or approximately correct in content. Decoded pictures produced by random access at or before the picture associated with the recovery point SEI message need not be correct in content until the indicated recovery point, and the operation of the decoding process starting at the picture associated with the recovery point SEI message may contain references to pictures unavailable in the decoded picture buffer.

In addition, by use of the broken_link_flag, the recovery point SEI message can indicate to the decoder the location of some pictures in the bitstream that can result in serious visual artefacts when displayed, even when the decoding process was begun at the location of a previous IDR access unit in decoding order.

NOTE 1 – The broken_link_flag can be used by encoders to indicate the location of a point after which the decoding process for the decoding of some pictures may cause references to pictures that, though available for use in the decoding process, are not the pictures that were used for reference when the bitstream was originally encoded (e.g., due to a splicing operation performed during the generation of the bitstream).

When random access is performed to start decoding from the access unit associated with the recovery point SEI message, the decoder operates as if the associated picture was the first picture in the bitstream in decoding order, and the variables prevPicOrderCntLsb and prevPicOrderCntMsb used in derivation of PicOrderCntVal are both set equal to 0.

NOTE 2 – When HRD information is present in the bitstream, a buffering period SEI message should be associated with the access unit associated with the recovery point SEI message in order to establish initialization of the HRD buffer model after a random access.

Any SPS or PPS RBSP that is referred to by a picture associated with a recovery point SEI message or by any picture following such a picture in decoding order shall be available to the decoding process prior to its activation, regardless of whether or not the decoding process is started at the beginning of the bitstream or with the access unit, in decoding order, that is associated with the recovery point SEI message.

recovery_poc_cnt specifies the recovery point of decoded pictures in output order. If there is a picture picA that follows the current picture (i.e., the picture associated with the current SEI message) in decoding order in the CVS and that has PicOrderCntVal equal to the PicOrderCntVal of the current picture plus the value of recovery_poc_cnt, the picture picA is referred to as the recovery point picture. Otherwise, the first picture in output order that has PicOrderCntVal greater than the PicOrderCntVal of the current picture plus the value of recovery_poc_cnt is referred to as the recovery point picture. The recovery point picture shall not precede the current picture in decoding order. All decoded pictures in output order are indicated to be correct or approximately correct in content starting at the output order position of the recovery point picture. The value of recovery_poc_cnt shall be in the range of −MaxPicOrderCntLsb / 2 to MaxPicOrderCntLsb / 2 − 1, inclusive.

exact_match_flag indicates whether decoded pictures at and subsequent to the specified recovery point in output order derived by starting the decoding process at the access unit associated with the recovery point SEI message will be an exact match to the pictures that would be produced by starting the decoding process at the location of a previous IDR access unit, if any, in the bitstream. The value 0 indicates that the match may not be exact and the value 1 indicates that the match will be exact. When exact_match_flag is equal to 1, it is a requirement of bitstream conformance that the decoded pictures at and subsequent to the specified recovery point in output order derived by starting the decoding process at the access unit associated with the recovery point SEI message shall be an exact match to the pictures that would be produced by starting the decoding process at the location of a previous IDR access unit, if any, in the bitstream.

NOTE 3 – When performing random access, decoders should infer all references to unavailable pictures as references to pictures containing only intra coding blocks and having sample values given by Y equal to (1 << ($BitDepth_Y$ − 1)), Cb and Cr both equal to (1 << ($BitDepth_C$ − 1)) (mid-level grey), regardless of the value of exact_match_flag.

When exact_match_flag is equal to 0, the quality of the approximation at the recovery point is chosen by the encoding process and is not specified in this Specification.

broken_link_flag indicates the presence or absence of a broken link in the NAL unit stream at the location of the recovery point SEI message and is assigned further semantics as follows:

– If broken_link_flag is equal to 1, pictures produced by starting the decoding process at the location of a previous IDR access unit may contain undesirable visual artefacts to the extent that decoded pictures at and subsequent to the access unit associated with the recovery point SEI message in decoding order should not be displayed until the specified recovery point in output order.

– Otherwise (broken_link_flag is equal to 0), no indication is given regarding any potential presence of visual artefacts.

Regardless of the value of the broken_link_flag, pictures subsequent to the specified recovery point in output order are specified to be correct or approximately correct in content.

D.3.5 Mastering display colour volume SEI message semantics

This SEI message identifies the colour volume (the colour primaries, white point and luminance range) of a display considered to be the mastering display for the associated video content – e.g., the colour volume of a display that was used for viewing while authoring the video content. The described mastering display is a three-colour additive display system that has been configured to use the indicated mastering colour volume.

This SEI message does not specify the measurement methodologies and procedures used for determining the indicated values or any description of the mastering environment. It also does not provide information on colour transformations that would be appropriate to preserve creative intent on displays with colour volumes different from that of the described mastering display.

The information conveyed in this SEI message is intended to be adequate for purposes corresponding to the use of SMPTE ST 2086 (2014).

When a mastering display colour volume SEI message is present for any picture of a CLVS of a particular layer, a mastering display colour volume SEI message shall be present for the first picture of the CLVS. The mastering display colour volume SEI message persists for the current layer in decoding order from the current picture until the end of the CLVS. All mastering display colour volume SEI messages that apply to the same CLVS shall have the same content.

display_primaries_x[c] and **display_primaries_y**[c] specify the normalized x and y chromaticity coordinates, respectively, of the colour primary component c of the mastering display, according to the CIE 1931 definition of x and y as specified in ISO 11664-1 (see also ISO 11664-3 and CIE 15), in increments of 0.00002. For describing mastering displays that use red, green, and blue colour primaries, it is suggested that index value c equal to 0 should correspond to the green primary, c equal to 1 should correspond to the blue primary and c equal to 2 should correspond to the red colour primary (see also Annex E and Table E.3). The values of display_primaries_x[c] and display_primaries_y[c] shall be in the range of 0 to 50 000, inclusive.

white_point_x and **white_point_y** specify the normalized x and y chromaticity coordinates, respectively, of the white point of the mastering display, according to the CIE 1931 definition of x and y as specified in ISO 11664-1 (see also ISO 11664-3 and CIE 15), in normalized increments of 0.00002. The values of white_point_x and white_point_y shall be in the range of 0 to 50 000.

max_display_mastering_luminance and **min_display_mastering_luminance** specify the nominal maximum and minimum display luminance, respectively, of the mastering display in units of 0.0001 candelas per square metre. min_display_mastering_luminance shall be less than max_display_mastering_luminance.

At minimum luminance, the mastering display is considered to have the same nominal chromaticity as the white point.

D.3.6 Content light level information SEI message semantics

This SEI message identifies upper bounds for the nominal target brightness light level of the pictures of the CLVS.

The information conveyed in this SEI message is intended to be adequate for purposes corresponding to the use of the Consumer Electronics Association 861.3 specification.

The semantics of the content light level information SEI message are defined in relation to the values of samples in a 4:4:4 representation of red, green, and blue colour primary intensities in the linear light domain for the pictures of the CLVS, in units of candelas per square metre. However, this SEI message does not, by itself, identify a conversion process for converting the sample values of a decoded picture to the samples in a 4:4:4 representation of red, green, and blue colour primary intensities in the linear light domain for the picture.

NOTE 1 – – Other syntax elements, such as colour_primaries, transfer_characteristics, matrix_coeffs, and the chroma resampling filter hint SEI message, when present, may assist in the identification of such a conversion process.

Given the red, green, and blue colour primary intensities in the linear light domain for the location of a luma sample in a corresponding 4:4:4 representation, denoted as E_R, E_G, and E_B, the maximum component intensity is defined as E_{Max} = Max(E_R, Max(E_G, E_B)). The light level corresponding to the stimulus is then defined as the CIE 1931 luminance corresponding to equal amplitudes of E_{Max} for all three colour primary intensities for red, green, and blue (with appropriate scaling to reflect the nominal luminance level associated with peak white – e.g., ordinarily scaling to associate peak white with 10 000 candelas per square metre when transfer_characteristics is equal to 16).

NOTE 2 – – Since the maximum value E_{Max} is used in this definition at each sample location, rather than a direct conversion from E_R, E_G, and E_B to the corresponding CIE 1931 luminance, the CIE 1931 luminance at a location may in some cases be less than the indicated light level. This situation would occur, for example, when E_R and E_G are very small and E_B is large, in which case the indicated light level would be much larger than the true CIE 1931 luminance associated with the (E_R, E_G, E_B) triplet.

All content light level information SEI messages that apply to the same CLVS shall have the same content.

max_content_light_level, when not equal to 0, indicates an upper bound on the maximum light level among all individual samples in a 4:4:4 representation of red, green, and blue colour primary intensities (in the linear light domain) for the pictures of the CLVS, in units of candelas per square metre. When equal to 0, no such upper bound is indicated by max_content_light_level.

max_pic_average_light_level, when not equal to 0, indicates an upper bound on the maximum average light level among the samples in a 4:4:4 representation of red, green, and blue colour primary intensities (in the linear light domain) for any

individual picture of the CLVS, in units of candelas per square metre. When equal to 0, no such upper bound is indicated by max_pic_average_light_level.

NOTE 3 – When the visually relevant region does not correspond to the entire cropped decoded picture, such as for "letterbox" encoding of video content with a wide picture aspect ratio within a taller cropped decoded picture, the indicated average should be performed only within the visually relevant region.

Annex E

Video usability information

(This annex forms an integral part of this International Standard.)

E.1 General

This Annex specifies syntax and semantics of the VUI parameters of the SPSs.

VUI parameters are not required for constructing the luma or chroma samples by the decoding process. Conforming decoders are not required to process this information for output order conformance to this International Standard (see Annex C for the specification of conformance). Some VUI parameters are required to check bitstream conformance and for output timing decoder conformance.

In Annex E, specification for presence of VUI parameters is also satisfied when those parameters (or some subset of them) are conveyed to decoders (or to the HRD) by other means not specified by this International Standard. When present in the bitstream, VUI parameters shall follow the syntax and semantics specified in subclauses 7.3.2.1 and 7.4.3.1 and this annex. When the content of VUI parameters is conveyed for the application by some means other than presence within the bitstream, the representation of the content of the VUI parameters is not required to use the same syntax specified in this annex. For the purpose of counting bits, only the appropriate bits that are actually present in the bitstream are counted.

E.2 VUI syntax

E.2.1 VUI parameters syntax

vui_parameters( ) {	C	Descriptor
aspect_ratio_info_present_flag	0	u(1)
if(aspect_ratio_info_present_flag) {		
aspect_ratio_idc	0	u(8)
if(aspect_ratio_idc = = Extended_SAR) {		
sar_width	0	u(16)
sar_height	0	u(16)
}		
}		
overscan_info_present_flag	0	u(1)
if(overscan_info_present_flag)		
overscan_appropriate_flag	0	u(1)
video_signal_type_present_flag	0	u(1)
if(video_signal_type_present_flag) {		
video_format	0	u(3)
video_full_range_flag	0	u(1)
colour_description_present_flag	0	u(1)
if(colour_description_present_flag) {		
colour_primaries	0	u(8)

transfer_characteristics	0	u(8)
matrix_coefficients	0	u(8)
}		
}		
chroma_loc_info_present_flag	0	u(1)
if(chroma_loc_info_present_flag) {		
chroma_sample_loc_type_top_field	0	ue(v)
chroma_sample_loc_type_bottom_field	0	ue(v)
}		
timing_info_present_flag	0	u(1)
if(timing_info_present_flag) {		
num_units_in_tick	0	u(32)
time_scale	0	u(32)
fixed_pic_rate_flag	0	u(1)
}		
nal_hrd_parameters_present_flag	0	u(1)
if(nal_hrd_parameters_present_flag)		
hrd_parameters( )		
vcl_hrd_parameters_present_flag	0	u(1)
if(vcl_hrd_parameters_present_flag)		
hrd_parameters( )		
if(nal_hrd_parameters_present_flag || vcl_hrd_parameters_present_flag)		
low_delay_hrd_flag	0	u(1)
pic_struct_present_flag	0	u(1)
bitstream_restriction_flag	0	u(1)
if(bitstream_restriction_flag) {		
motion_vectors_over_pic_boundaries_flag	0	u(1)
max_bytes_per_pic_denom	0	ue(v)
max_bits_per_mb_denom	0	ue(v)
log2_max_mv_length_horizontal	0	ue(v)
log2_max_mv_length_vertical	0	ue(v)
num_reorder_pics	0	ue(v)
max_dec_pic_buffering	0	ue(v)
}		
}		

E.2.2 HRD parameters syntax

hrd_parameters( ) {	C	Descriptor
cpb_cnt_minus1	0	ue(v)
bit_rate_scale	0	u(4)
cpb_size_scale	0	u(4)
for(SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++) {		
bit_rate_value_minus1[SchedSelIdx **]**	0	ue(v)
cpb_size_value_minus1[SchedSelIdx **]**	0	ue(v)
cbr_flag[SchedSelIdx **]**	0	u(1)
}		
initial_cpb_removal_delay_length_minus1	0	u(5)
cpb_removal_delay_length_minus1	0	u(5)
dpb_output_delay_length_minus1	0	u(5)
time_offset_length	0	u(5)
}		

E.3 VUI semantics

E.3.1 VUI parameters semantics

aspect_ratio_info_present_flag equal to 1 specifies that aspect_ratio_idc is present. aspect_ratio_info_present_flag equal to 0 specifies that aspect_ratio_idc is not present.

aspect_ratio_idc specifies the value of the sample aspect ratio of the luma samples. Table E.1 shows the meaning of the code. When aspect_ratio_idc indicates Extended_SAR, the sample aspect ratio is represented by sar_width and sar_height. When the aspect_ratio_idc syntax element is not present, aspect_ratio_idc value shall be inferred to be equal to 0.

Table E.1 – Meaning of sample aspect ratio indicator

aspect_ratio_idc	Sample aspect ratio	(informative) Examples of use
0	Unspecified	
1	1:1 ("square")	1280x720 16:9 frame without horizontal overscan 1920x1080 16:9 frame without horizontal overscan (cropped from 1920x1088) 640x480 4:3 frame without horizontal overscan
2	12:11	720x576 4:3 frame with horizontal overscan 352x288 4:3 frame without horizontal overscan
3	10:11	720x480 4:3 frame with horizontal overscan 352x240 4:3 frame without horizontal overscan
4	16:11	720x576 16:9 frame with horizontal overscan 528x576 4:3 frame without horizontal overscan
5	40:33	720x480 16:9 frame with horizontal overscan 528x480 4:3 frame without horizontal overscan
6	24:11	352x576 4:3 frame without horizontal overscan 480x576 16:9 frame with horizontal overscan
7	20:11	352x480 4:3 frame without horizontal overscan 480x480 16:9 frame with horizontal overscan
8	32:11	352x576 16:9 frame without horizontal overscan
9	80:33	352x480 16:9 frame without horizontal overscan
10	18:11	480x576 4:3 frame with horizontal overscan
11	15:11	480x480 4:3 frame with horizontal overscan
12	64:33	528x576 16:9 frame without horizontal overscan
13	160:99	528x480 16:9 frame without horizontal overscan
14	4:3	1440x1080 16:9 frame without horizontal overscan
15	3:2	1280x1080 16:9 frame without horizontal overscan
16	2:1	960x1080 16:9 frame without horizontal overscan
17..254	Reserved	
255	Extended_SAR	

sar_width indicates the horizontal size of the sample aspect ratio (in arbitrary units).

sar_height indicates the vertical size of the sample aspect ratio (in the same arbitrary units as sar_width).

sar_width and sar_height shall be relatively prime or equal to 0. When aspect_ratio_idc is equal to 0 or sar_width is equal to 0 or sar_height is equal to 0, the sample aspect ratio shall be considered unspecified by this International Standard.

overscan_info_present_flag equal to 1 specifies that the overscan_appropriate_flag is present. When overscan_info_present_flag is equal to 0 or is not present, the preferred display method for the video signal is unspecified.

overscan_appropriate_flag equal to 1 indicates that the cropped decoded pictures output are suitable for display using overscan. overscan_appropriate_flag equal to 0 indicates that the cropped decoded pictures output contain visually important information in the entire region out to the edges of the cropping rectangle of the picture, such that the cropped decoded pictures output should not be displayed using overscan. Instead, they should be displayed using either an exact match between the display area and the cropping rectangle, or using underscan.

NOTE 1 – For example, overscan_appropriate_flag equal to 1 might be used for entertainment television programming, or for a live view of people in a videoconference, and overscan_appropriate_flag equal to 0 might be used for computer screen capture or security camera content.

video_signal_type_present_flag equal to 1 specifies that video_format, video_full_range_flag and colour_description_present_flag are present. video_signal_type_present_flag equal to 0, specify that video_format, video_full_range_flag and colour_description_present_flag are not present.

video_format indicates the representation of the pictures as specified in Table E.2, before being coded in accordance with this International Standard. When the video_format syntax element is not present, video_format value shall be inferred to be equal to 5.

Table E.2 – Meaning of video_format

video_format	Meaning
0	Component
1	PAL
2	NTSC
3	SECAM
4	MAC
5	Unspecified video format
6	Reserved
7	Reserved

video_full_range_flag indicates the black level and range of the luma and chroma signals as derived from E'_Y, E'_{PB}, and E'_{PR} or E'_R, E'_G, and E'_B analogue component signals.

When the video_full_range_flag syntax element is not present, the value of video_full_range_flag shall be inferred to be equal to 0.

colour_description_present_flag equal to 1 specifies that colour_primaries, transfer_characteristics and matrix_coefficients are present. colour_description_present_flag equal to 0 specifies that colour_primaries, transfer_characteristics and matrix_coefficients are not present.

colour_primaries indicates the chromaticity coordinates of the source primaries as specified in Table E.3 in terms of the CIE 1931 definition of x and y as specified by ISO/CIE 10527.

When the colour_primaries syntax element is not present, the value of colour_primaries shall be inferred to be equal to 2 (the chromaticity is unspecified or is determined by the application).

Table E.3 – Colour primaries

Value	Primaries			Informative Remark
0	Reserved			For future use by ITU / ISO/IEC
1	primary green blue red white D65	x 0.300 0.150 0.640 0.3127	y 0.600 0.060 0.330 0.3290	ITU-R Recommendation BT.709-5
2	Unspecified			Image characteristics are unknown or are determined by the application.
3	Reserved			
4	primary green blue red white C	x 0.21 0.14 0.67 0.310	y 0.71 0.08 0.33 0.316	ITU-R Recommendation BT.470-6 System M
5	primary green blue red white D65	x 0.29 0.15 0.64 0.3127	y 0.60 0.06 0.33 0.3290	ITU-R Recommendation BT.470-6 System B, G
6	primary green blue red white D65	x 0.310 0.155 0.630 0.3127	y 0.595 0.070 0.340 0.3290	Society of Motion Picture and Television Engineers 170M (1999)
7	primary green blue red white D65	x 0.310 0.155 0.630 0.3127	y 0.595 0.070 0.340 0.3290	Society of Motion Picture and Television Engineers 240M (1999)
8	primary green blue red white C	x 0.243 0.145 0.681 0.310	y 0.692 (Wratten 58) 0.049 (Wratten 47) 0.319 (Wratten 25) 0.316	Generic film (colour filters using Illuminant C)
9	primary green blue red white D65	x 0.170 0.131 0.708 0.3127	y 0.797 0.046 0.292 0.3290	Rec. ITU-R BT.2020-2 Rec. ITU-R BT.2100-0
10-255	Reserved			For future use by ISO/IEC

transfer_characteristics indicates the opto-electronic transfer characteristic of the source picture as specified in Table E.4 as a function of a linear optical intensity input L_c with an analogue range of 0 to 1.

When the transfer_characteristics syntax element is not present, the value of transfer_characteristics shall be inferred to be equal to 2 (the transfer characteristics are unspecified or are determined by the application).

NOTE 1 – As indicated in Table E.4, some values of transfer_characteristics are defined in terms of a reference opto-electronic transfer characteristic function and others are defined in terms of a reference electro-optical transfer characteristic function, according to the convention that has been applied in other Specifications. In the cases of Rec. ITU-R BT.709-6 and Rec. ITU-R BT.2020-2 (which may be indicated by transfer_characteristics equal to 1, 6, 14, or 15), although the value is defined in terms of a reference opto-electronic transfer characteristic function, a suggested corresponding reference electro-optical transfer characteristic function for flat panel displays used in HDTV studio production has been specified in Rec. ITU-R BT.1886-0.

Table E.4 – Transfer characteristics

Value	Transfer Characteristic	Informative Remark
0	Reserved	For future use by ITU-T / ISO/IEC
1	$V = 1.099\ L_c^{0.45} - 0.099$ for $1 >= L_c >= 0.018$ $V = 4.500\ L_c$ for $0.018 > L_c >= 0$	ITU-R Recommendation BT.709-5
2	Unspecified	Image characteristics are unknown or are determined by the application.
3	Reserved	For future use by ITU-T / ISO/IEC
4	Assumed display gamma 2.2	ITU-R Recommendation BT.470-6 System M
5	Assumed display gamma 2.8	ITU-R Recommendation BT.470-6 System B, G
6	$V = 1.099\ L_c^{0.45} - 0.099$ for $1 >= L_c >= 0.018$ $V = 4.500\ L_c$ for $0.018 > L_c >= 0$	Society of Motion Picture and Television Engineers 170M (1999)
7	$V = 1.1115\ L_c^{0.45} - 0.1115$ for $1 >= L_c >= 0.0228$ $V = 4.0\ L_c$ for $0.0228 > L_c >= 0$	Society of Motion Picture and Television Engineers 240M (1999)
8	$V = L_C$ for $1 > L_c >= 0$	Linear transfer characteristics
9	$V = 1.0 - \text{Log10}(L_c) \div 2$ for $1 >= L_c >= 0.01$ $V = 0.0$ for $0.01 > L_c >= 0$	Logarithmic transfer characteristic (100:1 range)
10	$V = 1.0 - \text{Log10}(L_c) \div 2.5$ for $1 >= L_c >= 0.0031622777$ $V = 0.0$ for $0.0031622777 > L_c >= 0$	Logarithmic transfer characteristic (316.22777:1 range)
11..13	Reserved	For future use by ISO/IEC
14	$V = \alpha * L_c^{0.45} - (\alpha - 1)$ for $1 >= L_c >= \beta$ $V = 4.500 * L_c$ for $\beta > L_c >= 0$	Rec. ITU-R BT.2020-2 (functionally the same as the values 1, 6 and 15)
15	$V = \alpha * L_c^{0.45} - (\alpha - 1)$ for $1 >= L_c >= \beta$ $V = 4.500 * L_c$ for $\beta > L_c >= 0$	Rec. ITU-R BT.2020-2 (functionally the same as the values 1, 6 and 14)
16	$V = ((c_1 + c_2 * L_o^n) \div (1 + c_3 * L_o^n))^m$ for all values of L_o $c_1 = c_3 - c_2 + 1 = 3424 \div 4096 = 0.8359375$ $c_2 = 32 * 2413 \div 4096 = 18.8515625$ $c_3 = 32 * 2392 \div 4096 = 18.6875$ $m = 128 * 2523 \div 4096 = 78.84375$ $n = 0.25 * 2610 \div 4096 = 0.1593017578125$ for which L_o equal to 1 for peak white is ordinarily intended to correspond to a reference output luminance level of 10 000 candelas per square metre	SMPTE ST 2084 for 10, 12, 14 and 16-bit systems Rec. ITU-R BT.2100-0 perceptual quantization (PQ) system
17	Reserved	For future use by ISO/IEC
18	$V = a * \text{Ln}(12 * L_c - b) + c$ for $1 >= L_c > 1 \div 12$ $V = \text{Sqrt}(3) * L_c^{0.5}$ for $1 \div 12 >= L_c >= 0$ $a = 0.17883277$, $b = 0.28466892$, $c = 0.55991073$	Association of Radio Industries and Businesses (ARIB) STD-B67 Rec. ITU-R BT.2100-0 hybrid log-gamma (HLG) system
19..255	Reserved	For future use by ISO/IEC

NOTE 2 – For transfer_characteristics equal to 18, the equations given in Table E.4 are normalized for a source input linear optical intensity L_c with a nominal real-valued range of 0 to 1. An alternative scaling that is mathematically equivalent is used in ARIB STD-B67 with the source input linear optical intensity having a nominal real-valued range of 0 to 12.

matrix_coefficients describes the matrix coefficients used in deriving luma and chroma signals from the green, blue, and red primaries, as specified in Table E.5.

matrix_coefficients shall not be equal to 0 unless both of the following conditions are true

– $BitDepth_C$ is equal to $BitDepth_Y$

– chroma_format_idc is equal to 3 (4:4:4)

The specification of the use of matrix_coefficients equal to 0 under all other conditions is reserved for future use by ITU-T | ISO/IEC.

matrix_coefficients shall not be equal to 8 unless one or both of the following conditions are true

– $BitDepth_C$ is equal to $BitDepth_Y$

– $BitDepth_C$ is equal to $BitDepth_Y + 1$ and chroma_format_idc is equal to 3 (4:4:4)

The specification of the use of matrix_coefficients equal to 8 under all other conditions is reserved for future use by ITU-T | ISO/IEC.

When the matrix_coefficients syntax element is not present, the value of matrix_coefficients shall be inferred to be equal to 2.

The interpretation of matrix_coefficients is defined as follows.

– E'_R, E'_G, and E'_B are analogue with values in the range of 0 to 1.

– White is specified as having E'_R equal to 1, E'_G equal to 1, and E'_B equal to 1.

– Black is specified as having E'_R equal to 0, E'_G equal to 0, and E'_B equal to 0.

– If video_full_range_flag is equal to 0, the following equations apply.

  – If matrix_coefficients is equal to 1, 4, 5, 6, or 7, the following equations apply.

$$Y = Round((1 << (BitDepth_Y – 8)) * (219 * E'_Y + 16)) \quad (E\text{-}1)$$

$$Cb = Round((1 << (BitDepth_C – 8)) * (224 * E'_{PB} + 128)) \quad (E\text{-}2)$$

$$Cr = Round((1 << (BitDepth_C – 8)) * (224 * E'_{PR} + 128)) \quad (E\text{-}3)$$

  – Otherwise, if matrix_coefficients is equal to 0 or 8, the following equations apply.

$$R = (1 << (BitDepth_Y – 8)) * (219 * E'_R + 16) \quad (E\text{-}4)$$

$$G = (1 << (BitDepth_Y – 8)) * (219 * E'_G + 16) \quad (E\text{-}5)$$

$$B = (1 << (BitDepth_Y – 8)) * (219 * E'_B + 16) \quad (E\text{-}6)$$

  – Otherwise, if matrix_coefficients is equal to 2, the interpretation of the matrix_coefficients syntax element is unknown or is determined by the application.

  – Otherwise (matrix_coefficients is not equal to 0, 1, 2, 4, 5, 6, 7, or 8), the interpretation of the matrix_coefficients syntax element is reserved for future definition by ITU-T | ISO/IEC.

– Otherwise (video_full_range_flag is equal to 1), the following equations apply.

  – If matrix_coefficients is equal to 1, 4, 5, 6, or 7, the following equations apply.

$$Y = Round(((1 << BitDepth_Y) – 1) * E'_Y) \quad (E\text{-}7)$$

$$Cb = Round(((1 << BitDepth_C) – 1) * E'_{PB} + (1 << (BitDepth_C – 1))) \quad (E\text{-}8)$$

$$Cr = Round(((1 << BitDepth_C) – 1) * E'_{PR} + (1 << (BitDepth_C – 1))) \quad (E\text{-}9)$$

  – Otherwise, if matrix_coefficients is equal to 0 or 8, the following equations apply.

$$R = ((1 << BitDepth_Y) – 1) * E'_R \quad (E\text{-}10)$$

$$G = ((1 << BitDepth_Y) – 1) * E'_G \quad (E\text{-}11)$$

$$B = ((1 << BitDepth_Y) – 1) * E'_B \quad (E\text{-}12)$$

- Otherwise, if matrix_coefficients is equal to 2, the interpretation of the matrix_coefficients syntax element is unknown or is determined by the application.
- Otherwise (matrix_coefficients is not equal to 0, 1, 2, 4, 5, 6, 7, or 8), the interpretation of the matrix_coefficients syntax element is reserved for future definition by ITU-T | ISO/IEC.

– If matrix_coefficients is not equal to 0 or 8, the following equations apply.

$$E'_Y = K_R * E'_R + (1 - K_R - K_B) * E'_G + K_B * E'_B \quad \text{(E-13)}$$

$$E'_{PB} = 0.5 * (E'_B - E'_Y) \div (1 - K_B) \quad \text{(E-14)}$$

$$E'_{PR} = 0.5 * (E'_R - E'_Y) \div (1 - K_R) \quad \text{(E-15)}$$

NOTE 3 – Then E'_Y is analogue with values in the range of 0 to 1, E'_{PB} and E'_{PR} are analogue with values in the range of -0.5 to 0.5, and white is equivalently given by $E'_Y = 1$, $E'_{PB} = 0$, $E'_{PR} = 0$.

– Otherwise, if matrix_coefficients is equal to 0, the following equations apply.

$$Y = \text{Round}(G) \quad \text{(E-16)}$$

$$Cb = \text{Round}(B) \quad \text{(E-17)}$$

$$Cr = \text{Round}(R) \quad \text{(E-18)}$$

– Otherwise (matrix_coefficients is equal to 8), the following applies.
  – If BitDepth_C is equal to BitDepth_Y, the following equations apply.

$$Y = \text{Round}(0.5 * G + 0.25 * (R + B)) \quad \text{(E-19)}$$

$$Cb = \text{Round}(0.5 * G - 0.25 (R + B)) \quad \text{(E-20)}$$

$$Cr = \text{Round}(0.5 * (R - B)) \quad \text{(E-21)}$$

NOTE 4 – For purposes of the YCgCo nomenclature used in Table E.5, Cb and Cr of Equations E-20 and E-21 may be referred to as Cg and Co, respectively. The inverse conversion for the above four equations should be computed as.

$$t = Y - Cb \quad \text{(E-22)}$$

$$G = Y + Cb \quad \text{(E-23)}$$

$$B = t - Cr \quad \text{(E-24)}$$

$$R = t + Cr \quad \text{(E-25)}$$

  – Otherwise (BitDepth_C is not equal to BitDepth_Y), the following equations apply.

$$Cr = \text{Round}(R) - \text{Round}(B) \quad \text{(E-26)}$$

$$t = \text{Round}(B) + (Cr >> 1) \quad \text{(E-27)}$$

$$Cb = \text{Round}(G) - t \quad \text{(E-28)}$$

$$Y = t + (Cb >> 1) \quad \text{(E-29)}$$

NOTE 5 – For purposes of the YCgCo nomenclature used in Table E.5, Cb and Cr of Equations E-28 and E-26 may be referred to as Cg and Co, respectively. The inverse conversion for the above four equations should be computed as.

$$t = Y - (Cb >> 1) \quad \text{(E-30)}$$

$$G = t + Cb \quad \text{(E-31)}$$

$$B = t - (Cr >> 1) \quad \text{(E-32)}$$

$$R = B + Cr \quad \text{(E-33)}$$

Table E.5 – Matrix coefficients

Value	Matrix	Informative remark
0	GBR	Typically referred to as RGB; see Equations E-16 to E-18
1	$K_R = 0.2126$; $K_B = 0.0722$	ITU-R Recommendation BT.709-5 and Society of Motion Picture and Television Engineers RP177 (1993)
2	Unspecified	Image characteristics are unknown or are determined by the application.
3	Reserved	For future use by ITU-T / ISO/IEC
4	$K_R = 0.30$; $K_B = 0.11$	United States Federal Communications Commission Title 47 Code of Federal Regulations (2003) 73.682 (a) (20)
5	$K_R = 0.299$; $K_B = 0.114$	ITU-R Recommendation BT.470-6 System B, G
6	$K_R = 0.299$; $K_B = 0.114$	Society of Motion Picture and Television Engineers 170M (1999)
7	$K_R = 0.212$; $K_B = 0.087$	Society of Motion Picture and Television Engineers 240M (1999)
8	YCgCo	See Equations E-19 to E-33
9	$K_R = 0.2627$; $K_B = 0.0593$	Rec. ITU-R BT.2020-2 non-constant luminance system See Equations E-28 to E-30
10	$K_R = 0.2627$; $K_B = 0.0593$	Rec. ITU-R BT.2020-2 constant luminance system See Equations E-49 to E-58
11-255	Reserved	For future use by ISO/IEC

chroma_loc_info_present_flag equal to 1 specifies that chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field are present. chroma_loc_info_present_flag equal to 0 specifies that chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field are not present.

chroma_sample_loc_type_top_field and **chroma_sample_loc_type_bottom_field** specify the location of chroma samples for the top field and the bottom field as shown in Figure E-1. The value of chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field shall be in the range of 0 to 5, inclusive. When the chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field are not present, the values of chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field shall be inferred to be equal to 0.

NOTE 6 – When coding progressive source material, chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field should have the same value.

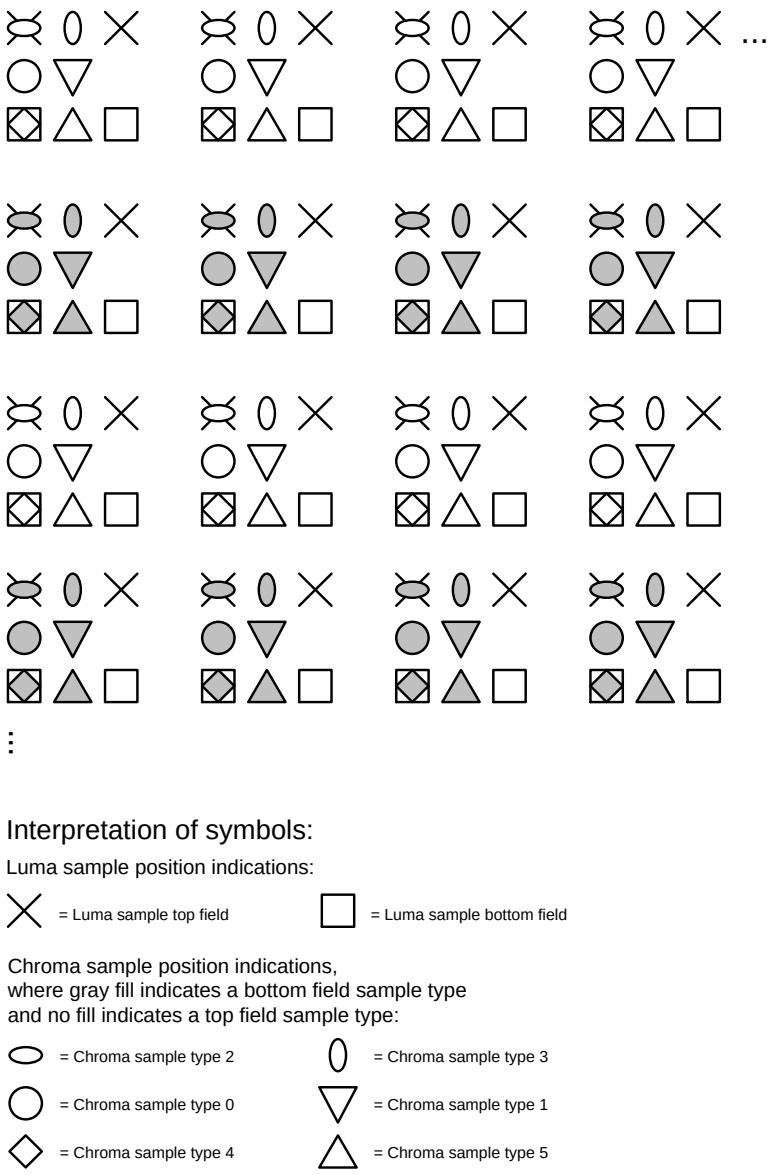

Figure E-1 – Location of chroma samples for top and bottom fields as a function of chroma_sample_loc_type_top_field and chroma_sample_loc_type_bottom_field

timing_info_present_flag equal to 1 specifies that num_units_in_tick, time_scale and fixed_pic_rate_flag are present in the bitstream. timing_info_present_flag equal to 0 specifies that num_units_in_tick, time_scale and fixed_pic_rate_flag are not present in the bitstream.

num_units_in_tick is the number of time units of a clock operating at the frequency time_scale Hz that corresponds to one increment (called a clock tick) of a clock tick counter. num_units_in_tick shall be greater than 0. A clock tick is the minimum interval of time that can be represented in the coded data. For example, when the clock frequency of a video signal is 60 000 ÷ 1001 Hz, time_scale may be equal to 60 000 and num_units_in_tick may be equal to 1001. See Equation C-1.

time_scale is the number of time units that pass in one second. For example, a time coordinate system that measures time using a 27 MHz clock has a time_scale of 27 000 000. time_scale shall be greater than 0.

fixed_pic_rate_flag equal to 1 indicates that the temporal distance between the HRD output times of any two consecutive pictures in output order is constrained as follows. fixed_pic_rate_flag equal to 0 indicates that no such constraints apply to the temporal distance between the HRD output times of any two consecutive pictures in output order.

For each picture n where n indicates the n-th picture (in output order) that is output and picture n is not the last picture in the bitstream (in output order) that is output, the value of $\Delta t_{fi,dpb}(n)$ is specified by

$$\Delta t_{fi,dpb}(n) = \Delta t_{o,dpb}(n) \div \text{DeltaTfiDivisor} \quad \text{(E-34)}$$

where $\Delta t_{o,dpb}(n)$ is specified in Equation C-13 and DeltaTfiDivisor is specified by Table E.6 based on the value of pic_struct_present_flag, field_pic_flag, and pic_struct for the coded video sequence containing picture n. Entries marked "-" in Table E.6 indicate a lack of dependence of DeltaTfiDivisor on the corresponding syntax element.

When fixed_pic_rate_flag is equal to 1 for a coded video sequence containing picture n, the value computed for $\Delta t_{fi,dpb}(n)$ shall be equal to t_c as specified in Equation C-1 (using the value of t_c for the coded video sequence containing picture n) when either or both of the following conditions are true for the following picture n_n that is specified for use in Equation C-13.

– picture n_n is in the same coded video sequence as picture n.

– picture n_n is in a different coded video sequence and fixed_pic_rate_flag is equal to 1 in the coded video sequence containing picture n_n and the value of num_units_in_tick ÷ time_scale is the same for both coded video sequences.

Table E.6 – Divisor for computation of $\Delta t_{fi,dpb}(n)$

pic_struct_present_flag	field_pic_flag	pic_struct	DeltaTfiDivisor
0	1	-	1
1	-	1	1
1	-	2	1
0	0	-	2
1	-	0	2
1	-	3	2
1	-	4	2
1	-	5	3
1	-	6	3
1	-	7	4
1	-	8	6

nal_hrd_parameters_present_flag equal to 1 specifies that NAL HRD parameters (pertaining to Type II bitstream conformance) are present. nal_hrd_parameters_present_flag equal to 0 specifies that NAL HRD parameters are not present.

NOTE 7 – When nal_hrd_parameters_present_flag is equal to 0, the conformance of the bitstream cannot be verified without provision of the NAL HRD parameters, including the NAL sequence HRD parameter information and all buffering period and picture timing SEI messages, by some means not specified in this International Standard.

When nal_hrd_parameters_present_flag is equal to 1, NAL HRD parameters (subclauses E.1.2 and E.2.2) immediately follow the flag.

The variable NalHrdBpPresentFlag is derived as follows.

– If any of the following is true, the value of NalHrdBpPresentFlag shall be set equal to 1.
  – nal_hrd_parameters_present_flag is present in the bitstream and is equal to 1
  – the need for the presence of buffering periods for NAL HRD operation to be present in the bitstream in buffering period SEI messages is determined by the application, by some means not specified in this International Standard.

- Otherwise, the value of NalHrdBpPresentFlag shall be set equal to 0.

vcl_hrd_parameters_present_flag equal to 1 specifies that VCL HRD parameters (pertaining to all bitstream conformance) are present. vcl_hrd_parameters_present_flag equal to 0 specifies that VCL HRD parameters are not present.

NOTE 8 – When vcl_hrd_parameters_present_flag is equal to 0, the conformance of the bitstream cannot be verified without provision of the VCL HRD parameters and all buffering period and picture timing SEI messages, by some means not specified in this International Standard.

When vcl_hrd_parameters_present_flag is equal to 1, VCL HRD parameters (subclauses E.1.2 and E.2.2) immediately follow the flag.

The variable VclHrdBpPresentFlag is derived as follows.

– If any of the following is true, the value of VclHrdBpPresentFlag shall be set equal to 1.

  - vcl_hrd_parameters_present_flag is present in the bitstream and is equal to 1
  - the need for the presence of buffering periods for VCL HRD operation to be present in the bitstream in buffering period SEI messages is determined by the application, by some means not specified in this International Standard.
- Otherwise, the value of VclHrdBpPresentFlag shall be set equal to 0.

The variable CpbDpbDelaysPresentFlag is derived as follows.

- If any of the following is true, the value of CpbDpbDelaysPresentFlag shall be set equal to 1.
  - nal_hrd_parameters_present_flag is present in the bitstream and is equal to 1
  - vcl_hrd_parameters_present_flag is present in the bitstream and is equal to 1
  - the need for the presence of CPB and DPB output delays to be present in the bitstream in picture timing SEI messages is determined by the application, by some means not specified in this International Standard.
- Otherwise, the value of CpbDpbDelaysPresentFlag shall be set equal to 0.

low_delay_hrd_flag specifies the HRD operational mode as specified in Annex C. When fixed_pic_rate_flag is equal to 1, low_delay_hrd_flag shall be equal to 0.

NOTE 9 – When low_delay_hrd_flag is equal to 1, "big pictures" that violate the nominal CPB removal times due to the number of bits used by an access unit are permitted. It is expected, but not required, that such "big pictures" occur only occasionally.

pic_struct_present_flag equal to 1 specifies that picture timing SEI messages (subclause D.2.2) are present that include the pic_struct syntax element. pic_struct_present_flag equal to 0 specifies that the pic_struct syntax element is not present in picture timing SEI messages. When pic_struct_present_flag is not present, its value shall be inferred to be equal to 0.

bitstream_restriction_flag equal to 1, specifies that the following coded video sequence bitstream restriction parameters are present. bitstream_restriction_flag equal to 0, specifies that the following coded video sequence bitstream restriction parameters are not present.

motion_vectors_over_pic_boundaries_flag equal to 0 indicates that no sample outside the picture boundaries and no sample at a fractional sample position whose value is derived using one or more samples outside the picture boundaries is used to inter predict any sample. motion_vectors_over_pic_boundaries_flag equal to 1 indicates that one or more samples outside picture boundaries may be used in inter prediction. When the motion_vectors_over_pic_boundaries_flag syntax element is not present, motion_vectors_over_pic_boundaries_flag value shall be inferred to be equal to 1.

max_bytes_per_pic_denom indicates a number of bytes not exceeded by the sum of the sizes of the VCL NAL units associated with any coded picture in the coded video sequence.

The number of bytes that represent a picture in the NAL unit stream is specified for this purpose as the total number of bytes of VCL NAL unit data (i.e., the total of the NumBytesInNALunit variables for the VCL NAL units) for the picture. The value of max_bytes_per_pic_denom shall be in the range of 0 to 16, inclusive.

Depending on max_bytes_per_pic_denom the following applies.

- If max_bytes_per_pic_denom is equal to 0, no limits are indicated.
- Otherwise (max_bytes_per_pic_denom is not equal to 0), no coded picture shall be represented in the coded video sequence by more than the following number of bytes.

$$(\text{PicSizeInMbs} * \text{RawMbBits}) \div (8 * \text{max\_bytes\_per\_pic\_denom}) \quad \text{(E-35)}$$

When the max_bytes_per_pic_denom syntax element is not present, the value of max_bytes_per_pic_denom shall be inferred to be equal to 2.

max_bits_per_mb_denom indicates the maximum number of coded bits of macroblock_layer( ) data for any macroblock in any picture of the coded video sequence. The value of max_bits_per_mb_denom shall be in the range of 0 to 16, inclusive.

Depending on max_bits_per_mb_denom the following applies.

- If max_bits_per_mb_denom is equal to 0, no limit is specified.
- Otherwise (max_bits_per_mb_denom is not equal to 0), no coded macroblock_layer( ) shall be represented in the bitstream by more than the following number of bits.

$$(128 + \text{RawMbBits}) \div \text{max\_bits\_per\_mb\_denom} \quad \text{(E-36)}$$

Depending on entropy_coding_mode_flag, the bits of macroblock_layer( ) data are counted as follows.

– If entropy_coding_mode_flag is equal to 0, the number of bits of macroblock_layer( ) data is given by the number of bits in the macroblock_layer( ) syntax structure for a macroblock.

– Otherwise (entropy_coding_mode_flag is equal to 1), the number of bits of macroblock_layer( ) data for a macroblock is given by the number of times read_bits(1) is called in subclauses 9.3.3.2.2 and 9.3.3.2.3 when parsing the macroblock_layer( ) associated with the macroblock.

When the max_bits_per_mb_denom is not present, the value of max_bits_per_mb_denom shall be inferred to be equal to 1.

log2_max_mv_length_horizontal and **log2_max_mv_length_vertical** indicate the maximum absolute value of a decoded horizontal and vertical motion vector component, respectively, in ¼ luma sample units, for all pictures in the coded video sequence. A value of n asserts that no value of a motion vector component shall exceed the range from -2^n to 2^n-1, inclusive, in units of ¼ luma sample displacement. The value of log2_max_mv_length_horizontal shall be in the range of 0 to 16, inclusive. The value of log2_max_mv_length_vertical shall be in the range of 0 to 16, inclusive. When log2_max_mv_length_horizontal is not present, the values of log2_max_mv_length_horizontal and log2_max_mv_length_vertical shall be inferred to be equal to 16.

NOTE 10 – The maximum absolute value of a decoded vertical or horizontal motion vector component is also constrained by profile and level limits as specified in Annex A.

num_reorder_pics indicates the maximum number of frames, complementary field pairs, or non-paired fields that precede any frame, complementary field pair, or non-paired field in the coded video sequence in decoding order and follow it in output order. The value of num_reorder_pics shall be in the range of 0 to max_dec_pic_buffering, inclusive. When the num_reorder_pics syntax element is not present, the value of num_reorder_pics value shall be inferred to be equal to max_dec_pic_buffering.

max_dec_pic_buffering specifies the required size of the HRD decoded picture buffer (DPB) in units of picture storage buffers. The coded video sequence shall not require a decoded picture buffer with size of more than Max(1, max_dec_pic_buffering) picture storage buffers to enable the output of decoded pictures at the output times specified by dpb_output_delay of the picture timing SEI messages. The value of max_dec_pic_buffering shall be in the range of num_ref_pics to MaxDpbSize (as specified in subclause A.3.1 or A.3.2), inclusive. When the max_dec_pic_buffering syntax element is not present, the value of max_dec_pic_buffering shall be inferred to be equal to MaxDpbSize.

E.3.2 HRD parameters semantics

cpb_cnt_minus1 plus 1 specifies the number of alternative CPB specifications in the bitstream. The value of cpb_cnt_minus1 shall be in the range of 0 to 31, inclusive. When low_delay_hrd_flag is equal to 1, cpb_cnt_minus1 shall be equal to 0. When cpb_cnt_minus1 is not present, it shall be inferred to be equal to 0.

bit_rate_scale (together with bit_rate_value_minus1[SchedSelIdx]) specifies the maximum input bit rate of the SchedSelIdx-th CPB.

cpb_size_scale (together with cpb_size_value_minus1[SchedSelIdx]) specifies the CPB size of the SchedSelIdx-th CPB.

bit_rate_value_minus1[SchedSelIdx **]** (together with bit_rate_scale) specifies the maximum input bit rate for the SchedSelIdx-th CPB. bit_rate_value_minus1[SchedSelIdx] shall be in the range of 0 to 2^{32} - 2, inclusive. For any SchedSelIdx > 0, bit_rate_value_minus1[SchedSelIdx] shall be greater than bit_rate_value_minus1[SchedSelIdx - 1]. The bit rate in bits per second is given by

$$BitRate[SchedSelIdx] = (bit\_rate\_value\_minus1[SchedSelIdx] + 1) * 2^{(6 + bit\_rate\_scale)} \quad \text{(E-37)}$$

When the bit_rate_value_minus1[SchedSelIdx] syntax element is not present, the value of BitRate[SchedSelIdx] shall be inferred as follows.

– If profile_idc is equal to 66, 77, or 88, BitRate[SchedSelIdx] shall be inferred to be equal to 1000 * MaxBR for VCL HRD parameters and to be equal to 1200 * MaxBR for NAL HRD parameters, where MaxBR is specified in subclause A.3.1.

– Otherwise, BitRate[SchedSelIdx] shall be inferred to be equal to cpbBrVclFactor * MaxBR for VCL HRD parameters and to be equal to cpbBrNalFactor * MaxBR for NAL HRD parameters, where cpbBrVclFactor, cpbBrNalFactor, and MaxBR are specified in subclause A.3.3.

cpb_size_value_minus1[SchedSelIdx **]** is used together with cpb_size_scale to specify the SchedSelIdx-th CPB size. cpb_size_value_minus1[SchedSelIdx] shall be in the range of 0 to 2^{32} - 2, inclusive. For any SchedSelIdx greater than 0, cpb_size_value_minus1[SchedSelIdx] shall be less than or equal to cpb_size_value_minus1[SchedSelIdx -1].

The CPB size in bits is given by

$$CpbSize[\ SchedSelIdx\ ] = (\ cpb\_size\_value\_minus1[\ SchedSelIdx\ ] + 1\ ) * 2^{(4 + cpb\_size\_scale)} \quad \text{(E-38)}$$

When the cpb_size_value_minus1[SchedSelIdx] syntax element is not present, the value of CpbSize[SchedSelIdx] shall be inferred as follows.

– If profile_idc is equal to 66, 77, or 88, CpbSize[SchedSelIdx] shall be inferred to be equal to 1000 * MaxCPB for VCL HRD parameters and to be equal to 1200 * MaxCPB for NAL HRD parameters, where MaxCPB is specified in subclause A.3.1.

– Otherwise, CpbSize[SchedSelIdx] shall be inferred to be equal to cpbBrVclFactor * MaxCPB for VCL HRD parameters and to be equal to cpbBrNalFactor * MaxCPB for NAL HRD parameters, where cpbBrVclFactor, cpbBrNalFactor, and MaxCPB are specified in subclause A.3.3.

cbr_flag[SchedSelIdx **]** equal to 0 specifies that to decode this bitstream by the HRD using the SchedSelIdx-th CPB specification, the hypothetical stream delivery scheduler (HSS) operates in an intermittent bit rate mode. cbr_flag[SchedSelIdx] equal to 1 specifies that the HSS operates in a constant bit rate (CBR) mode. When the cbr_flag[SchedSelIdx] syntax element is not present, the value of cbr_flag shall be inferred to be equal to 0.

initial_cpb_removal_delay_length_minus1 specifies the length in bits of the initial_cpb_removal_delay[SchedSelIdx] and initial_cpb_removal_delay_offset[SchedSelIdx] syntax elements of the buffering period SEI message. The length of initial_cpb_removal_delay[SchedSelIdx] and of initial_cpb_removal_delay_offset[SchedSelIdx] is initial_cpb_removal_delay_length_minus1 + 1. When the initial_cpb_removal_delay_length_minus1 syntax element is present in more than one hrd_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the initial_cpb_removal_delay_length_minus1 parameters shall be equal in both hrd_parameters( ) syntax structures. When the initial_cpb_removal_delay_length_minus1 syntax element is not present, it shall be inferred to be equal to 23.

cpb_removal_delay_length_minus1 specifies the length in bits of the cpb_removal_delay syntax element. The length of the cpb_removal_delay syntax element of the picture timing SEI message is cpb_removal_delay_length_minus1 + 1. When the cpb_removal_delay_length_minus1 syntax element is present in more than one hrd_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the cpb_removal_delay_length_minus1 parameters shall be equal in both hrd_parameters( ) syntax structures. When the cpb_removal_delay_length_minus1 syntax element is not present, it shall be inferred to be equal to 23.

dpb_output_delay_length_minus1 specifies the length in bits of the dpb_output_delay syntax element. The length of the dpb_output_delay syntax element of the picture timing SEI message is dpb_output_delay_length_minus1 + 1. When the dpb_output_delay_length_minus1 syntax element is present in more than one hrd_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the dpb_output_delay_length_minus1 parameters shall be equal in both hrd_parameters( ) syntax structures. When the dpb_output_delay_length_minus1 syntax element is not present, it shall be inferred to be equal to 23.

time_offset_length greater than 0 specifies the length in bits of the time_offset syntax element. time_offset_length equal to 0 specifies that the time_offset syntax element is not present. When the time_offset_length syntax element is present in more than one hrd_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the time_offset_length parameters shall be equal in both hrd_parameters( ) syntax structures. When the time_offset_length syntax element is not present, it shall be inferred to be equal to 24.

