INTERNATIONAL STANDARD

ISO/IEC 23094-1

First edition
2020-10

Information technology — General video coding —

Part 1: Essential video coding

Technologies de l'information — Codage vidéo général —

Partie 1: Codage vidéo essentiel

Reference number
ISO/IEC 23094-1:2020(E)

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

Published in Switzerland

Contents

Page

Foreword

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

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 document 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 and IEC 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) or the IEC list of patent declarations received (see http://patents.iec.ch).

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, *Information technology*, 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.

Introduction

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he/she is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from the patent database available at www.iso.org/patents.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — General video coding —

Part 1:
Essential video coding

1 Scope

This document specifies a video coding technology known as essential video coding (EVC), which contains syntax format, semantics and an associated decoding process. The decoding process is designed to guarantee that all EVC decoders conform to a specified combination of capabilities known as the profile, level and toolset. Any decoding process that produces identical cropped decoded output pictures to those produced by the described process is considered to be in conformance with the requirements of this document.

This document is designed to cover a wide range of application including but not limited to digital storage media, television broadcasting and real-time communications.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 11664-1, *Colorimetry — Part 1: CIE standard colorimetric observers*

3 Terms and definitions

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

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

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

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

3.1
access unit
set of *NAL units* (3.53) that are associated with each other according to a specified classification rule, are consecutive in *decoding order* (3.28), and contain exactly one *coded picture* (3.16)

3.2
adaptive loop filter
ALF
filtering process that is applied as part of the *decoding process* (3.29) and is controlled by *parameters* (3.58) conveyed in an *APS* (3.4)

3.3
ALF APS
APS (3.4) that controls the *ALF* (3.2) process

3.4
adaptation parameter set
APS
syntax structure (3.87) containing *syntax elements* (3.86) that apply to zero or more *slices* (3.81) or *pictures* (3.60) as determined by zero or more *syntax elements* (3.86) found in *slice headers* (3.82)

3.5
bi-predictive slice
B slice
slice (3.81) that is decoded using *intra prediction* (3.43) or using *inter prediction* (3.40) with at most two *motion vectors* (3.52) and *reference indices* (3.74) to predict the sample values of each *block* (3.11)

3.6
bin
one bit of a *bin string* (3.7)

3.7
bin string
intermediate binary representation of values of *syntax elements* (3.86) from the *binarization* (3.8) of the *syntax element* (3.86)

3.8
binarization
set of *bin strings* (3.7) for all possible values of a *syntax element* (3.86)

3.9
binarization process
unique mapping process of all possible values of a *syntax elements* (3.86) from the *binarization* (3.8) of the *syntax element* (3.86)

3.10
bitstream
sequence of bits, in the form of a *NAL unit stream* (3.54) or a *raw bitstream* (3.71), that forms the representation of *coded pictures* (3.16) and associated data forming one or more *coded video sequences* (3.19)

3.11
block
MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients* (3.93)

3.12
buffering period
set of *access units* (3.1) starting with an *access unit* (3.1) that contains a buffering period suplemental enhancement information (SEI) message and containing all subsequent *access units* (3.1) in *decoding order* (3.28) up to but not including the next *access unit* (3.1) (when present) that contains a buffering period SEI message

3.13
byte
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

3.14
byte-aligned
position in a *bitstream* (3.10) where the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream* (3.10)

Note 1 to entry: A bit or *byte* (3.13) or *syntax element* (3.86) is said to be byte-aligned when the position at which it appears in a *bitstream* (3.10) is byte-aligned.

3.15
chroma
sample array or single sample representing one of the two colour difference signals related to the primary colours, represented by the symbols Cb and Cr

Note 1 to entry: 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.

3.16
coded picture
coded representation (3.18) of a *picture* (3.60) containing all *CTUs* (3.22) of the *picture* (3.60)

3.17
coded picture buffer
CPB
first-in first-out buffer containing *access units* (3.1) in *decoding order* (3.28)

Note 1 to entry: Specified in the *hypothetical reference decoder* (3.36) in Annex C.

3.18
coded representation
data element as represented in its coded form

3.19
coded video sequence
CVS
sequence of *access units* (3.1) that consists, in *decoding order* (3.28), of an *IDR access unit* (3.38), followed by zero or more *access units* (3.1) that are not *IDR access units* (3.38), including all subsequent *access units* (3.1) up to but not including any subsequent *access unit* (3.1) that is an *IDR access unit* (3.38)

3.20
coding block
CB
MxN *block* (3.11) of samples for some values of M and N such that the division of a *CTB* (3.21) into coding blocks is a *partitioning* (3.59)

3.21
coding tree block
CTB
NxN *block* (3.11) of samples for some value of N such that the division of a *component* (3.24) into coding tree blocks is a *partitioning* (3.59)

3.22
coding tree unit
CTU
CTB (3.21) of *luma* (3.51) samples, two corresponding *CTBs* (3.21) of *chroma* (3.15) samples of a *picture* (3.60) that has three sample arrays, or a *CTB* of samples of a monochrome *picture* (3.60) or a *picture*

(3.60) that is coded using three separate colour planes and *syntax structures* (3.87) used to code the samples

3.23
coding unit
CU
coding block (3.20) of *luma* (3.51) samples, two corresponding *coding blocks* (3.20) of *chroma* (3.15) samples of a *picture* (3.60) that has three sample arrays, or a *coding block* (3.20) of samples of a monochrome *picture* (3.60) or a *picture* (3.60) that is coded using three separate colour planes and *syntax structures* (3.87) used to code the samples

3.24
component
array or single sample from one of the three arrays (*luma* (3.51) and two *chroma* (3.15)) that compose a *picture* (3.60) 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* (3.60) in monochrome format

3.25
decoded picture
derived by decoding a *coded picture* (3.16)

3.26
decoded picture buffer
DPB
buffer holding *decoded pictures* (3.25) for reference, output reordering, or output delay

Note 1 to entry: Specified for the *hypothetical reference decoder* (3.36) in Annex C.

3.27
decoder
embodiment of a *decoding process* (3.29)

3.28
decoding order
order in which *syntax elements* (3.86) are processed by the *decoding process* (3.29)

3.29
decoding process
process specified that reads a *bitstream* (3.10) and derives *decoded pictures* (3.25) from it

3.30
dynamic range adjustment
DRA
mapping process that is applied to *decoded picture* (3.25) prior to cropping and output as part of the *decoding process* (3.29) and is controlled by parameters conveyed in an *APS* (3.4)

3.31
DRA APS
APS (3.4) that controls the *DRA* (3.30) process

3.32
decoder under test
DUT
decoder (3.27) that is tested for conformance to this document

Note 1 to entry: A *decoder* (3.27) is tested by operating the *hypothetical stream scheduler* (3.37) to deliver a conforming *bitstream* (3.10) to the *decoder* (3.27) and to the *hypothetical reference decoder* (3.36) and comparing the values and timing of the output of the two *decoders* (3.27).

3.33
encoder
embodiment of an *encoding process* (3.34)

3.34
encoding process
process that produces a *bitstream* (3.10) conforming to this document

3.35
flag
variable or single-bit *syntax element* (3.86) that can take one of the two possible values: 0 and 1

3.36
hypothetical reference decoder
HRD
hypothetical *decoder* (3.27) model that specifies constraints on the variability of conforming *NAL unit streams* (3.54) or conforming *raw bitstreams* (3.10) produced by an encoding process

3.37
hypothetical stream scheduler
HSS
hypothetical delivery mechanism for the timing and data flow of the input of a *bitstream* (3.10) into the *hypothetical reference decoder* (3.36)

Note 1 to entry: The HSS is used for checking the conformance of a *bitstream* (3.10) or a *decoder* (3.36).

3.38
IDR access unit
access unit in which the *coded picture* is an *IDR picture*

3.39
IDR picture
coded picture (3.16) for which each *VCL NAL unit* (3.96) has NalUnitType equal to IDR_NUT

3.40
inter prediction
prediction (3.63) derived in a manner that is dependent on data elements (e.g., sample values or motion vectors) of one or more *reference pictures* (3.75)

Note 1 to entry: A *prediction* (3.63) from a *reference picture* (3.75) that is the current *picture* (3.60) itself is also inter prediction.

3.41
intra block copy
IBC
prediction (3.63) derived in a manner that is dependent on data elements (e.g., sample values or block vectors) of the same decoded *slice* (3.81) without referring to a *reference picture* (3.75)

3.42
intra coding
coding of a *coding block* (3.20), *slice* (3.81), or *picture* (3.60) that uses *intra prediction* (3.43)

3.43
intra prediction
prediction (3.63) derived from only data elements (e.g., sample values) of the same decoded *slice* (3.81) without referring to a *reference picture* (3.75)

3.44
intra slice
I slice
slice (3.81) that is decoded using *intra prediction* (3.43) only

3.45
level
defined set of constraints on the values that may be taken by the *syntax elements* (3.86) and variables of this document, or the value of a *transform coefficient* (3.93) prior to scaling

Note 1 to entry: The same set of levels is defined for all *profiles* (3.67), with most aspects of the definition of each level being in common across different *profiles* (3.67). Individual implementations can, within the specified constraints, support a different level for each supported *profile* (3.67).

3.46
list 0 motion vector
motion vector (3.52) associated with a *reference index* (3.74) pointing into a *reference picture list 0* (3.77)

3.47
list 0 prediction
inter prediction (3.40) of the content of a *slice* (3.81) using a *reference index* (3.74) pointing into a *reference picture list 0* (3.77)

3.48
list 1 motion vector
motion vector (3.52) associated with a *reference index* (3.74) pointing into a *reference picture list 1* (3.78)

3.49
list 1 prediction
inter prediction (3.40) of the content of a *slice* (3.81) using a *reference index* (3.74) pointing into a *reference picture list 1* (3.78)

3.50
long-term reference picture
LTRP
picture (3.60) that is marked as "used for long-term reference"

3.51
luma
sample array or single sample is representing the monochrome signal related to the primary colours, represented by the symbol or subscript Y or L

Note 1 to entry: 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.

3.52
motion vector
two-dimensional vector used for *inter prediction* (3.40) that provides an offset from the coordinates in the *decoded picture* (3.25) to the coordinates in a *reference picture* (3.75)

3.53
NAL unit
syntax structure (3.87) containing an indication of the type of data to follow and *bytes* (3.13) containing that data in the form of an *RBSP* (3.72) interspersed as necessary

3.54
NAL unit stream
sequence of *NAL units* (3.53)

3.55
non-IDR picture
coded picture (3.16) that is not an *IDR picture* (3.39)

3.56
non-VCL NAL unit
NAL unit (3.53) that is not a *VCL NAL unit* (3.96)

3.57
output order
order in which the *decoded pictures* (3.25) are output from the *decoded picture buffer* (3.26) [for the *decoded pictures* (3.25) that are to be output from the *decoded picture buffer* (3.26)]

3.58
parameter
syntax element (3.86) of an *SPS* (3.79), *PPS* (3.61) or *APS* (3.4)

3.59
partitioning
division of a set into subsets such that each element of the set is in exactly one of the subsets

3.60
picture
array of *luma* (3.51) samples in monochrome format or an array of *luma* (3.51) samples and two corresponding arrays of *chroma* (3.15) samples in 4:2:0, 4:2:2, and 4:4:4 colour format

Note 1 to entry: A picture can be either a frame or a field. However, in one *CVS* (3.19), either all pictures are frames or all pictures are fields.

3.61
picture parameter set
PPS
syntax structure (3.87) containing *syntax elements* (3.86) that apply to zero or more entire *coded pictures* (3.16) as determined by a *syntax element* (3.86) found in each *slice header* (3.82)

3.62
picture order count
POC
variable that is associated with each *picture* (3.60), uniquely identifies the associated *picture* (3.60) among all *pictures* (3.60) in the *CVS* (3.19), and, when the associated *picture* (3.60) is to be output from the *decoded picture buffer* (3.26), indicates the position of the associated *picture* (3.60) in *output order* (3.57) relative to the *output order* (3.57) positions of the other *pictures* (3.60) in the same *CVS* (3.19) that are to be output from the *decoded picture buffer* (3.26)

3.63
prediction
embodiment of the *prediction process* (3.63)

3.64
prediction process
use of a *predictor* (3.66) to provide an estimate of the data element [e.g., sample value or *motion vector* (3.52)] currently being decoded

3.65
predictive slice
P slice
slice (3.81) that is decoded using *intra prediction* (3.43) or using *inter prediction* (3.40) with at most one *motion vector* (3.52) and *reference index* (3.74) to predict the sample values of each *block* (3.11)

3.66
predictor
combination of specified values or previously decoded data elements [e.g., sample value or *motion vector* (3.52)] used in the *decoding process* (3.29) of subsequent data elements

3.67
profile
specified subset of the syntax of this document

3.68
quantization parameter
QP
variable used by the *decoding process* (3.29) for scaling of *transform coefficient* (3.93) levels

3.69
random access
act of starting the decoding process for a *bitstream* (3.10) at a point other than the beginning of the stream

3.70
raster scan
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

3.71
raw bitstream
encapsulation of a *NAL unit stream* (3.54) containing *NAL unit* (3.53) length field and *NAL units* (3.53)

Note 1 to entry: Specified in Annex A.

3.72
raw byte sequence payload
RBSP
syntax structure (3.87) containing an integer number of *bytes* (3.13) that is encapsulated in a *NAL unit* (3.53) and that is either empty or has the form of a *string of data bits* (3.85) containing *syntax elements* (3.86) followed by an *RBSP stop bit* (3.73) and zero or more subsequent bits equal to 0

3.73
raw byte sequence payload stop bit
RBSP stop bit
bit equal to 1 present within a *raw byte sequence payload* (3.72) after a *string of data bits* (3.85), for which the location of the end within an *RBSP* (3.72) can be identified by searching from the end of the *RBSP* (3.72) for the RBSP stop bit, which is the last non-zero bit in the *RBSP* (3.72)

3.74
reference index
index into a *reference picture list* (3.76)

3.75
reference picture
picture (3.60) that is a *short-term reference picture* (3.80) or *long-term reference picture* (3.50)

Note 1 to entry: A reference picture contains samples that can be used for *inter prediction* (3.40) in the *decoding process* (3.29) of subsequent *pictures* (3.60) in *decoding order* (3.28).

3.76
reference picture list
list of *reference pictures* (3.80) that is used for *inter prediction* (3.40) of a *P slice* (3.65) or *B slice* (3.5)

Note 1 to entry: For the *decoding process* (3.29) of a *P slice* (3.65), there is one reference picture list – *reference picture list 0* (3.77). For the *decoding process* (3.28) of a *B slice* (3.5), there are two reference picture lists – *reference picture list 0* (3.77) and *reference picture list 1* (3.78).

3.77
reference picture list 0
reference picture list (3.76) used for *inter prediction* of a *P slice* (3.65) or the first *reference picture list* (3.76) used for *inter prediction* (3.40) of a *B slice* (3.5)

3.78
reference picture list 1
second *reference picture list* (3.76) used for *inter prediction* (3.40) of a *B slice* (3.5)

3.79
sequence parameter set
SPS
syntax structure (3.87) containing *syntax elements* (3.86) that apply to zero or more entire *CVSs* as determined by the content of a *syntax element* (3.86) found in the *PPS* (3.61) referred to by a *syntax element* (3.86) found in each *slice header* (3.82)

3.80
short-term reference picture
STRP
picture (3.60) that is marked as "used for short-term reference"

3.81
slice
integer number of *tiles* (3.88) of a *picture* (3.60) in the *tile scan* (3.91) of the *picture* (3.60) and that are exclusively contained in a single *NAL unit* (3.53)

3.82
slice header
part of a coded *slice* (3.81) containing the data elements pertaining to the first or all *tiles* (3.88) represented in the *slice* (3.81)

3.83
source
term used to describe the video material or some of its attributes before *encoding process* (3.34)

3.84
split unit
SU
MxN *block* (3.11) of samples for some values and M and N such that the division of a *component* (3.24) into four quad *blocks* (3.11), or three horizontal *blocks* (3.11) or two horizontal *blocks* (3.11), or three vertical *blocks* (3.11) or two vertical *blocks* (3.11)

3.85
string of data bits
SODB
sequence of some number of bits representing *syntax elements* (3.86) present within a *raw byte sequence payload* (3.72) prior to the *raw byte sequence payload stop bit* (3.73), 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

3.86
syntax element
element of data represented in the *bitstream* (3.10)

3.87
syntax structure
zero or more *syntax elements* (3.86) present together in the *bitstream* (3.10) in a specified order

3.88
tile
rectangular region of *CTUs* (3.22) within a particular *tile column* (3.89) and a particular *tile row* (3.90) in a *picture* (3.60)

3.89
tile column
rectangular region of *CTUs* (3.22) having a height equal to the height of the *picture* (3.60) and width specified by *syntax elements* (3.86) in the *PPS* (3.61)

3.90
tile row
rectangular region of *CTUs* (3.22) having a height specified by *syntax elements* (3.86) in the *PPS* (3.61) and a width equal to the width of the *picture* (3.60)

3.91
tile scan
specific sequential ordering of *CTUs* (3.22) *partitioning* (3.59) a *picture* (3.60) in which the *CTUs* (3.22) are ordered consecutively in *CTU* (3.22) *raster scan* (3.70) in a *tile* (3.88) whereas *tiles* (3.88) in a *picture* (3.60) are ordered consecutively in a *raster scan* (3.70) of the *tiles* (3.88) of the *picture* (3.60)

3.92
transform block
TB
rectangular MxN *block* (3.11) of samples resulting from a transform in the *decoding process* (3.29)

3.93
transform coefficient
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* (3.29)

3.94
transform unit
TU
transform block (3.94) of *luma* (3.51) samples and two corresponding *transform blocks* (3.94) of *chroma* (3.15) samples of a *picture* (3.60)

3.95
tree
tree is a finite set of nodes with a unique root node

3.96
VCL NAL unit
collectively, coded *slice* (3.81) *NAL units* (3.53) and the subset of *NAL units* (3.53) that have reserved values of NalUnitType that are classified as VCL NAL units in this document

4 Abbreviated terms

ATS	adaptive transform selection
B	bi-predictive
CABAC	context-based adaptive binary arithmetic coding
CBR	constant bit rate
CIE	International Commission on Illumination (Commission Internationale de l'Eclairage)
DMVR	decoder-side motion vector refinement
EG	exponential-Golomb
EGk	k-th order exponential-Golomb
FCC	Federal Communications Commission (of the United States)
FIR	finite impulse response

FL	fixed length
GOP	group of pictures
HMVP	history-based motion vector prediction
I	intra
IDR	instantaneous decoding refresh
LSB	least significant bit
LPS	least probable symbol
MAC	multiplexed analogue components
MMVD	merge with motion vector difference
MPS	most probable symbol
MSB	most significant bit
MVP	motion vector prediction
NAL	network abstraction layer
NTSC	National Television System Committee (of the United States)
P	predictive
PAL	phase alternating line
RGB	same as GBR
RPL	reference picture list
SAR	sample aspect ratio
SECAM	sequential colour with memory (séquentiel couleur avec mémoire)
SEI	supplemental enhancement information
SMPTE	Society of Motion Picture and Television Engineers
TB	truncated binary
TR	truncated rice
U	unary
UCS	universal coded character set
UTF	UCS transmission format
VBR	variable bit rate
VCL	video coding layer
VUI	video usability information

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

+	addition
−	subtraction (as a two-argument operator) or negation (as a unary prefix operator)
*	multiplication, including matrix multiplication
×	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
÷	division in mathematical equations where no truncation or rounding is intended
$\frac{x}{y}$	division in mathematical equations where no truncation or rounding is intended
$\sum_{i=x}^{y} f(i)$	summation of f(i) with i taking all integer values from x up to and including y
x % y	modulus. Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0

5.3 Logical operators

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

>	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

&	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

=	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

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

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

Asin(x) 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 (2)

Atan(x) 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 (3)

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

$\text{Ceil}(x)$ smallest integer greater than or equal to x (5)

$\text{Clip1}_Y(x) = \text{Clip3}(0, (1 << \text{BitDepth}_Y) - 1, x)$ (6)

$\text{Clip1}_C(x) = \text{Clip3}(0, (1 << \text{BitDepth}_C) - 1, x)$ (7)

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

$\text{Cos}(x)$ trigonometric cosine function operating on an argument x in units of radians (9)

$\text{Floor}(x)$ largest integer less than or equal to x (10)

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

$\text{Ln}(x)$ natural logarithm of x (the base-e logarithm, where e is the natural logarithm base constant 2.718 281 828...) (12)

$\text{Log2}(x)$ base-2 logarithm of x (13)

$\text{Log10}(x)$ base-10 logarithm of x (14)

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

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

$\text{Round}(x) = \text{Sign}(x) * \text{Floor}(\text{Abs}(x) + 0.5)$ (17)

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

$\text{Sin}(x)$ trigonometric sine function operating on an argument x in units of radians (19)

$\text{Sqrt}(x) = \sqrt{x}$ (20)

$\text{Swap}(x, y) = (y, x)$ (21)

$\text{Tan}(x)$ trigonometric tangent function operating on an argument x in units of radians (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 1 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

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

Table 1 — Operation precedence from highest (top of the table) to lowest (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", "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 letters and without any underscore characters. Variables with names starting with an upper case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables with names 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 with names starting with a lower case letter are only used within the subclause in which they are derived.

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

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in subclause 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 subclause 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 subclause specifies the relationship between the network abstraction layer (NAL) unit stream and raw bitstream, 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 raw bitstream file storage 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 raw bitstream file storage 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 raw bitstream file storage

format are outside the scope of this document. The raw bitstream file storage format is specified in Annex B.

6.2 Source, decoded and output picture formats

This subclause 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 2, 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 2 — 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.

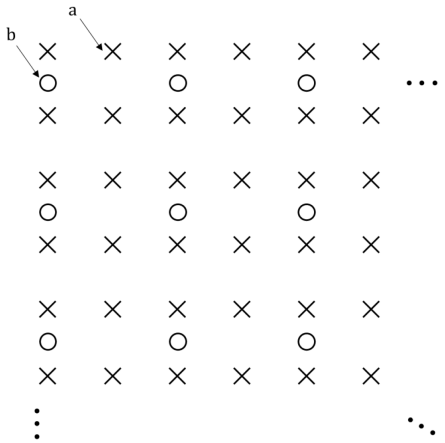
In 4:2:0 sampling, each of the two chroma arrays has half the height and half the width of the luma array.

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 1. Alternative chroma sample relative locations may be indicated in video usability information (see Annex E).

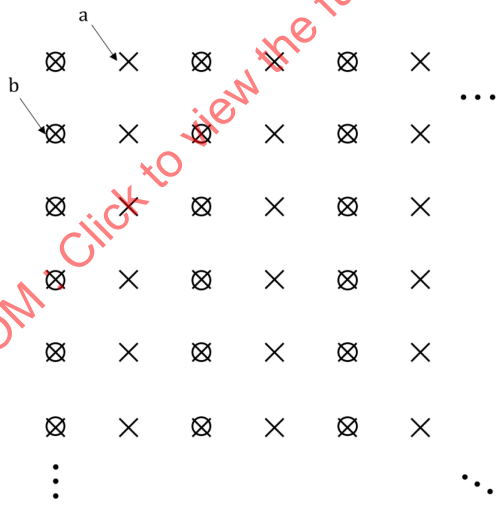


a Location of luma sample.

b Location of chroma sample.

Figure 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 2.

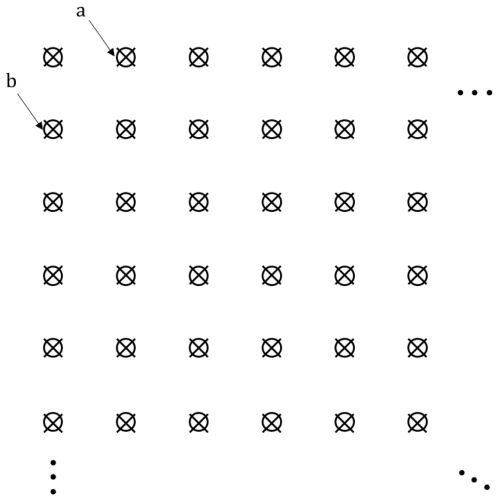


a Location of luma sample.

b Location of chroma sample.

Figure 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 3.



a Location of luma sample.

b Location of chroma sample.

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

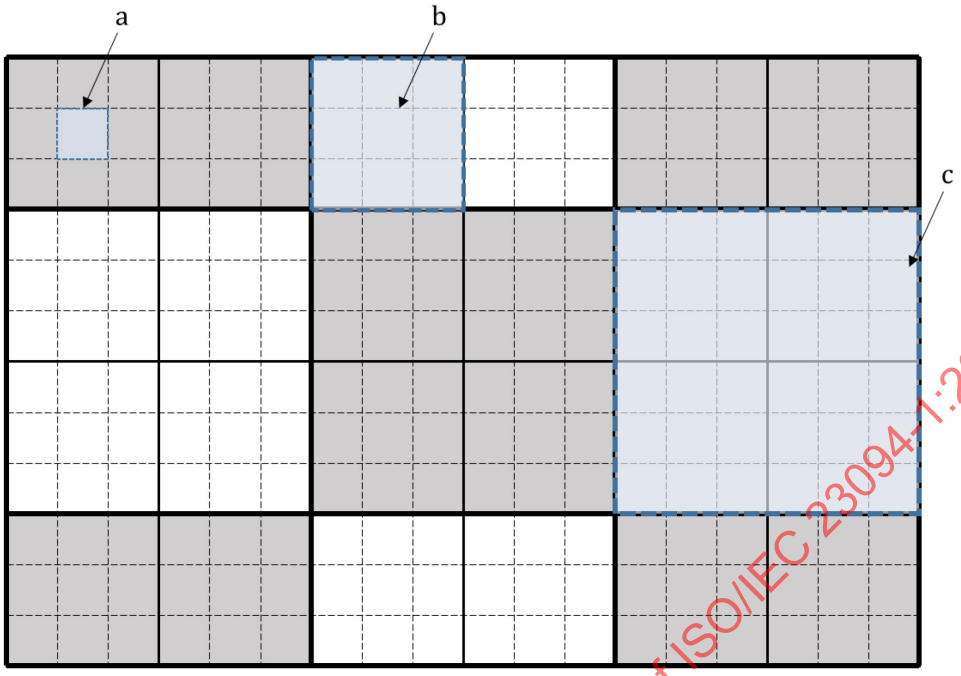
6.3 Partitioning of pictures, slices, tiles, and CTUs

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 4.



Key

a CTU
b tile
c slice

Figure 4 — Picture with 18 by 12 luma CTUs that is partitioned into 24 tiles and 9 slices

6.3.2 Spatial or component-wise partitionings

The following divisions of processing elements 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; and

— the division of each CTU into CTBs.

6.4 Availability processes

6.4.1 Derivation process for neighbouring block availability

Input to this process is 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 one or more of the following conditions are true, availableN is set equal to FALSE.

  — The neighbouring block is contained in a different tile than the current block.

  — xNbY is less than 0.

  — yNbY is less than 0.

  — xNbY is greater than or equal to pic_width_in_luma_samples.

  — yNbY is greater than or equal to pic_height_in_luma_samples.

  — IsCoded[xNbY][yNbY] is equal to FALSE.

— Otherwise, availableN is set equal to TRUE.

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, and

— a variable nCbW specifying the width 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).

— The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked the left luma location (xNbL, yNbL) as input, and the output is assigned to the coding block availability flag, availableL.

— The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the right luma location (xNbR, yNbR) as input, 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 (23)

availLR can be equal to LR_00, LR_10, LR_01, or 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 2 when left neighbouring block is not available but the right block is available; LR_11 denotes the availabilty, availLR, equal to 3 when both left and right neighbouring block are available. When sps_suco_flag is equal to 0, availLR is always smaller than 2 (availLR is equal to LR_10 or availLR is equal to LR_00).

6.4.3 Derivation process for neighbouring block motion vector candidate availability

Input to this process is 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 one or more of the following conditions are true, availableN is set equal to FALSE.

  — The neighbouring block is contained in a different tile than the current block.

  — xNbY is less than 0.

  — yNbY is less than 0.

  — xNbY is greater than or equal to pic_width_in_luma_samples.

  — yNbY is greater than or equal to pic_height_in_luma_samples.

  — IsCoded[xNbY][yNbY] is equal to FALSE.

  — The neighbouring block is coded in intra or intra block copy mode.

— Otherwise, availableN is set equal to TRUE.

6.4.4 Derivation process for ALF neighbouring block availability

Input to this process is 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 one or more of the following conditions are true, availableN is set equal to FALSE.

  — xNbY is less than 0.

  — yNbY is less than 0.

— xNbY is greater than or equal to pic_width_in_luma_samples.

— yNbY is greater than or equal to pic_height_in_luma_samples.

— IsCoded[xNbY][yNbY] is equal to FALSE.

— The neighbouring block is coded in intra or intra block copy mode.

— 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                                    (24)
    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                                     (25)
    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 ]                                          (26)
```

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 ]                                    (27)
```

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++ )                                    (28)
        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++ )                      (29)
    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++ )                              (30)
            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++ )                      (31)
        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                                    (32)
        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 1D array initialization process

Inputs to this process are:

— a variable blkWidth specifying the width of block, and

— a variable blkHeight specifying the height of block.

Output of this process is the array zigZagScan[sPos].

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
zigZagScan[ pos ] = 0
pos++
for( line = 1; line < ( blkWidth + blkHeight − 1 ); line++ ) {
    if( line % 2 ) {
        x = Min( line, blkWidth − 1 )
        y = Max( 0, line − ( blkWidth − 1 ) )
        while( x >= 0 && y < blkHeight ) {
            zigZagScan[ pos ] = y * blkWidth + x
            pos++
            x− −
            y++
        }
    }                                                                            (33)
    else{
        y = Min( line, −1 )
        x = Max( 0, line − ( blkHeight − 1 ) )
        while( y >= 0 && x < blkWidth ) {
            zigZagScan[ pos ] = y * blkWidth + x
```

```
            pos++
            x++
            y– –
        }
    }
}
```

6.5.3 Inverse scan order 1D array initialization process

Inputs to this process are:

— a variable blkWidth specifying the width of block, and

— a variable blkHeight specifying the height of block.

Output of this process is the array inverseScan[rPos].

The array index rPos specifies the raster scan position ranging from 0 to (blkWidth * blkHeight) – 1. Depending on the value of blkWidth and blkHeight, the array inverseScan is derived as follows:

— The variable forwardScan is derived by invoking zig-zag scan order 1D array initialization process as specified in subclause 6.5.2 with input parameters blkWidth and blkHeight.

— The output variable inverseScan is derived as follows:

```
for( pos = 0; pos < blkWidth * blkHeight; pos++ ) {
    inverseScan[ forwardScan[ pos ] ] = pos                    (34)
}
```

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.

NOTE An actual decoder must 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.

Table 3 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.

Table 3 — Examples of the syntax specification format

	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 in this document 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_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 raw bitstream file storage 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.

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 subclause 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 subclause 9.2

— 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 subclause 9.2.

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

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++)	
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(8)
level_idc	u(8)
toolset_idc_h	u(32)
toolset_idc_l	u(32)
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_minus5	ue(v)
log2_min_cb_size_minus2	ue(v)
log2_diff_ctu_max_14_cb_size	ue(v)
log2_diff_ctu_max_tt_cb_size	ue(v)
log2_diff_min_cb_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_admvp_flag	u(1)
if(sps_admvp_flag) {	
sps_affine_flag	u(1)
sps_amvr_flag	u(1)
sps_dmvr_flag	u(1)
sps_mmvd_flag	u(1)

sps_hmvp_flag	u(1)
}	
sps_eipd_flag	u(1)
if(sps_eipd_flag) {	
sps_ibc_flag	u(1)
if(sps_ibc_flag)	
log2_max_ibc_cand_size_minus2	ue(v)
}	
sps_cm_init_flag	u(1)
if(sps_cm_init_flag)	
sps_adcc_flag	u(1)
sps_iqt_flag	u(1)
if(sps_iqt_flag)	
sps_ats_flag	u(1)
sps_addb_flag	u(1)
sps_alf_flag	u(1)
sps_htdf_flag	u(1)
sps_rpl_flag	u(1)
sps_pocs_flag	u(1)
sps_dquant_flag	u(1)
sps_dra_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)
}	
if(!sps_rpl_flag)	
max_num_tid0_ref_pics	ue(v)
else {	
sps_max_dec_pic_buffering_minus1	ue(v)
long_term_ref_pics_flag	u(1)
rpl1_same_as_rpl0_flag	u(1)
for(i = 0; i < !rpl1_same_as_rpl0_flag ? 2 : 1; i++) {	
num_ref_pic_lists_in_sps[i]	ue(v)
for(j = 0; j < num_ref_pic_lists_in_sps[i]; j++)	
ref_pic_list_struct(i, j, 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)

}	
if(ChromaArrayType != 0) {	
chroma_qp_table_present_flag	u(1)
if(chroma_qp_table_present_flag)	
same_qp_table_for_chroma	u(1)
global_offset_flag	u(1)
for(i = 0; i < same_qp_table_for_chroma ? 1 : 2; i++) {	
num_points_in_qp_table_minus1[i]	ue(v)
for(j = 0; j <= num_points_in_qp_table_minus1[i]; j++)	
delta_qp_in_val_minus1[i][j]	u(6)
delta_qp_out_val[i][j]	se(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)
additional_lt_poc_lsb_len	ue(v)
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_len_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)
pic_dra_enabled_flag	u(1)
if(pic_dra_enabled_flag)	
pic_dra_aps_id	u(5)
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)
aps_params_type	u(3)
if(aps_params_type = = ALF_APS)	
alf_data( )	
else if(aps_params_type = = DRA_APS)	
dra_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	
do {	
payload_type_byte	u(8)
payloadType += payload_type_byte	
} while(payload_type_byte = = 0xFF)	
payloadSize = 0	
do {	
payload_size_byte	u(8)
payloadSize += payload_size_byte	
} while(payload_size_byte = = 0xFF)	
sei_payload(payloadType, payloadSize)	
}	

7.3.4 Slice header syntax

slice_header( ) {	Descriptor
slice_pic_parameter_set_id	ue(v)
if(!single_tile_in_pic_flag) {	
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_NUT)	
no_output_of_prior_pics_flag	u(1)
if(sps_mmvd_flag && (slice_type == B || slice_type == P))	
mmvd_group_enable_flag	u(1)
if(sps_alf_flag) {	
slice_alf_enabled_flag	u(1)
if(slice_alf_enabled_flag) {	
slice_alf_luma_aps_id	u(5)
slice_alf_map_flag	u(1)
slice_alf_chroma_idc	u(2)
if((ChromaArrayType == 1 || ChromaArrayType == 2) && slice_alf_chroma_idc > 0)	
slice_alf_chroma_aps_id	u(5)
}	
if(ChromaArrayType == 3) {	
if(!slice_alf_enabled_flag)	
slice_alf_chroma_idc	u(2)
if(sliceChromaAlfEnabledFlag) {	
slice_alf_chroma_aps_id	u(5)
slice_alf_chroma_map_flag	u(1)
}	
if(sliceChroma2AlfEnabledFlag) {	
slice_alf_chroma2_aps_id	u(5)
slice_alf_chroma2_map_flag	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(num_ref_pic_lists_in_sps[i] > 0 && (i == 0 || (i == 1 && rpl1_idx_present_flag)))	
ref_pic_list_sps_flag[i]	u(1)
if(ref_pic_list_sps_flag[i]) {	
if(num_ref_pic_lists_in_sps[i] > 1 &&	
(i == 0 || (i == 1 && rpl1_idx_present_flag)))	
ref_pic_list_idx[i]	u(v)
} else	
ref_pic_list_struct(i, num_ref_pic_lists_in_sps[i], 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(sps_admvp_flag) {	
temporal_mvp_assigned_flag	u(1)
if(temporal_mvp_assigned_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( )	
}	

7.3.5 Adaptive loop filter data syntax

alf_data( ) {	**Descriptor**
alf_luma_filter_signal_flag	u(1)
alf_chroma_filter_signal_flag	u(1)
if(alf_luma_filter_signal_flag) {	
alf_luma_num_filters_signalled_minus1	ue(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]	u(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_flag[i]	u(1)
}	
if(alf_luma_fixed_filter_usage_pattern > 0) {	
for(i = 0; i < NumAlfFilters; i++) {	
if(alf_luma_fixed_filter_usage_flag[i])	
alf_luma_fixed_filter_set_idx[i]	u(4)
}	
}	
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 < LumaMaxGolombIdx; i++)	
alf_luma_eg_order_increase_flag[i]	u(1)
if(alf_luma_coeff_delta_flag) {	
for(i = 0; i < NumSignalledFilter; i++)	
alf_luma_coeff_flag[i]	u(1)
}	

for(i = 0; i < NumSignalledFilter; 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_flag[i][j]	u(1)
}	
}	
}	
}	
if(alf_chroma_filter_signal_flag) {	
alf_chroma_min_eg_order_minus1	ue(v)
for(i = 0; i < ChromaMaxGolombIdx; 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_flag[j]	u(1)
}	
}	
}	

7.3.6 DRA data syntax

dra_data( ) {	**Descriptor**
dra_descriptor1	u(4)
dra_descriptor2	u(4)
dra_number_ranges_minus1	ue(v)
dra_equal_ranges_flag	u(1)
dra_global_offset	u(10)
if(dra_equal_ranges_flag)	
dra_delta_range[0]	u(10)
else	
for(j = 0; j <= dra_number_ranges_minus1; j++)	
dra_delta_range[j]	u(10)
for(j = 0; j <= dra_number_ranges_minus1; j++)	
dra_scale_value[j]	u(v)
dra_cb_scale_value	u(v)
dra_cr_scale_value	u(v)
dra_table_idx	ue(v)
}	

7.3.7 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)
}	
}	

7.3.8 Slice data syntax

7.3.8.1 General slice data syntax

slice_data( ) {	Descriptor
for(i = 0; i < NumTilesInSlice; i++) {	
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( )	
}	
}	

7.3.8.2 Coding tree unit syntax

coding_tree_unit( ) {	Descriptor
xCtb = (CtbAddrInRs % PicWidthInCtbsY) << CtbLog2SizeY	
yCtb = (CtbAddrInRs / PicWidthInCtbsY) << CtbLog2SizeY	
tileIndex = TileIdToIdx[TileId[CtbAddrRsToTs[CtbAddrInRs]]]	
firstCtbAddrRs = CtbAddrTsToRs[FirstCtbAddrTs[tileIndex]]	
xFirstCtb = (firstCtbAddrRs % PicWidthInCtbsY) << CtbLog2SizeY	
if(xCtb == xFirstCtb)	
NumHmvpCand = 0	
if(slice_alf_enabled_flag && slice_alf_map_flag)	
alf_ctb_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY]	ae(v)
if(sliceChromaAlfEnabledFlag && slice_alf_chroma_map_flag)	
alf_ctb_chroma_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY]	ae(v)
if(sliceChroma2AlfEnabledFlag && slice_alf_chroma2_map_flag)	
alf_ctb_chroma2_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY]	ae(v)
split_unit(xCtb, yCtb, CtbLog2SizeY, CtbLog2SizeY, 0, 0, 0, PRED_MODE_NO_CONSTRAINT)	
}	

7.3.8.3 Split unit syntax

split_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth, splitUnitOrder, cuQpDeltaCode, predModeConstraintCurrent) {	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 != 2) {	
if(log2CbWidth > MaxTbLog2SizeY || log2CbHeight > MaxTbLog2SizeY)	
cuQpDeltaCode = 2	

else	
cuQpDeltaCode = 1	
isCuQpDeltaCoded = 0	
}	
else if((log2CbWidth + log2CbHeight == cuQpDeltaArea + 1 && btt_split_type[x0][y0] = = 1) | | (log2CbWidth + log2CbHeight == cuQpDeltaArea && cuQpDeltaCode != 2)) {	
cuQpDeltaCode = 2	
isCuQpDeltaCoded = 0	
}	
}	
}	
}	
if(sps_suco_flag && allowSplitUnitCodingOrder)	
split_unit_coding_order_flag[x0][y0]	ae(v)
if(needSignalPredModeConstraintTypeFlag)	
pred_mode_constraint_type_flag[x0][y0]	ae(v)
if(split_cu_flag[x0][y0]) {	
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 + 1, 0, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
if(x1 < pic_width_in_luma_samples)	
split_unit(x1, y0, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 0, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
if(y1 < pic_height_in_luma_samples)	
split_unit(x0, y1, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 0, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
split_unit(x1, y1, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 0, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
}	
else {	
split_unit(x1, y0, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 1, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
split_unit(x0, y0, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 1, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
split_unit(x1, y1, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 1, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
split_unit(x0, y1, log2CbWidth − 1, log2CbHeight − 1, ctDepth + 1, 1, cuQpDeltaCode, PRED_MODE_NO_CONSTRAINT)	
}	
}	
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, cuQpDeltaCode, predModeConstraint)	
if(x1 < pic_width_in_luma_samples)	
split_unit(x1, y0, log2CbWidth − 1, log2CbHeight, ctDepth + 1, 0, cuQpDeltaCode, predModeConstraint)	
}	
else {	
split_unit(x1, y0, log2CbWidth − 1, log2CbHeight, ctDepth + 1, 1, cuQpDeltaCode, predModeConstraint)	
split_unit(x0, y0, log2CbWidth − 1, log2CbHeight, ctDepth + 1, 1, cuQpDeltaCode, predModeConstraint)	
}	
}	
else if(SplitMode[x0][y0] == SPLIT_BT_HOR) {	
y1 = y0 + (1 << (log2CbHeight − 1))	
split_unit(x0, y0, log2CbWidth, log2CbHeight − 1, ctDepth + 1, splitUnitOrder, cuQpDeltaCode, predModeConstraint)	
if(y1 < pic_height_in_luma_samples)	
split_unit(x0, y1, log2CbWidth, log2CbHeight − 1, ctDepth + 1, splitUnitOrder, cuQpDeltaCode, predModeConstraint)	
}	
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, cuQpDeltaCode, predModeConstraint)	
split_unit(x1, y0, log2CbWidth − 1, log2CbHeight, ctDepth + 1, 0, cuQpDeltaCode, predModeConstraint)	
split_unit(x2, y0, log2CbWidth − 2, log2CbHeight, ctDepth + 2, 0, cuQpDeltaCode, predModeConstraint)	
}	
else {	
split_unit(x2, y0, log2CbWidth − 2, log2CbHeight, ctDepth + 2, 1, cuQpDeltaCode, predModeConstraint)	
split_unit(x1, y0, log2CbWidth − 1, log2CbHeight, ctDepth + 1, 1, cuQpDeltaCode, predModeConstraint)	
split_unit(x0, y0, log2CbWidth − 2, log2CbHeight, ctDepth + 2, 1, cuQpDeltaCode, predModeConstraint)	
}	
}	
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, cuQpDeltaCode, predModeConstraint)	
split_unit(x0, y1, log2CbWidth, log2CbHeight − 1, ctDepth + 1, splitUnitOrder, cuQpDeltaCode, predModeConstraint)	

split_unit(x0, y2, log2CbWidth, log2CbHeight − 2, ctDepth + 2, splitUnitOrder, cuQpDeltaCode, predModeConstraint)	
}	
else {	
treeType = predModeConstraint == PRED_MODE_CONSTRAINT_INTRA_IBC ? DUAL_TREE_LUMA : SINGLE_TREE	
if(predModeConstraint == PRED_MODE_NO_CONSTRAINT && (slice_type == I || (sps_admvp_flag && log2CbWidth == 2 && log2CbHeight == 2)))	
predModeConstraint = PRED_MODE_CONSTRAINT_INTRA_IBC	
coding_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth, cuQpDeltaCode)	
}	
if (isTreeSplitPoint)	
coding_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth, cuQpDeltaCode, DUAL_TREE_CHROMA, PRED_MODE_CONSTRAINT_INTRA_IBC)	
}	

7.3.8.4 Coding unit syntax

coding_unit(x0, y0, log2CbWidth, log2CbHeight, ctDepth, cuQpDeltaCode, treeType predModeConstraint) {	**Descriptor**
if(predModeConstraint != PRED_MODE_CONSTRAINT_INTRA_IBC)	
cu_skip_flag[x0][y0]	ae(v)
if(cu_skip_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 {	
if(!sps_admvp_flag) {	
mvp_idx_l0[x0][y0]	ae(v)
if(slice_type == B)	
mvp_idx_l1[x0][y0]	ae(v)
} else	
merge_idx[x0][y0]	ae(v)
}	
}	

}	
else {	
if(predModeConstraint == PRED_MODE_NO_CONSTRAINT)	
pred_mode_flag[x0][y0]	ae(v)
if(isIbcAllowed)	
ibc_flag[x0][y0]	ae(v)
if(CuPredMode == MODE_INTRA) {	
if(treeType != DUAL_TREE_CHROMA) {	
if(!sps_eipd_flag)	
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(treeType != DUAL_TREE_LUMA && sps_eipd_flag == 1 && ChromaArrayType != 0)	
intra_chroma_pred_mode[x0][y0]	ae(v)
}	
else { /* (CuPredMode != MODE_INTRA) */	
if(CuPredMode == MODE_IBC) {	
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_admvp_flag == 0)	
direct_mode_flag[x0][y0]	ae(v)
else if(sps_admvp_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	
merge_idx[x0][y0]	ae(v)
}	
}	
}	
if(direct_mode_flag[x0][y0] = = 0 && merge_mode_flag[x0][y0] = = 0) {	
if(slice_type = = B)	
inter_pred_idc[x0][y0]	ae(v)
if(sps_admvp_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_admvp_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_flag[x0][y0]	ae(v)
vertexNum = 1+ affine_flag[x0][y0] + affine_mode_flag[x0][y0]	
if(inter_pred_idc[x0][y0] != PRED_L1 || inter_pred_idc[x0][y0] = = PRED_BI) {	
ref_idx_l0[x0][y0]	ae(v)
affine_mvp_flag_l0[x0][y0]	ae(v)
affine_mvd_flag_l0[x0][y0]	ae(v)
for(vertex = 0; vertex < vertexNum; vertex++) {	
if (affine_mvd_flag_l0[x0][y0]) {	
abs_mvd_l0[x0][y0][vertex][0]	ae(v)
if(abs_mvd_l0[x0][y0][vertex][0])	
mvd_l0_sign_flag[x0][y0][vertex][0]	ae(v)
abs_mvd_l0[x0][y0][vertex][1]	ae(v)
if(abs_mvd_l0[x0][y0][vertex][1])	
mvd_l0_sign_flag[x0][y0][vertex][1]	ae(v)
}	
}	
}	
if(inter_pred_idc[x0][y0] != PRED_L0 || inter_pred_idc[x0][y0] = = PRED_BI) {	
ref_idx_l1[x0][y0]	ae(v)
affine_mvp_flag_l1[x0][y0]	ae(v)
affine_mvd_flag_l1[x0][y0]	ae(v)
for(vertex = 0; vertex < vertexNum; vertex++) {	
if(affine_mvd_flag_l1[x0][y0]) {	
abs_mvd_l1[x0][y0][vertex][0]	ae(v)
if(abs_mvd_l1[x0][y0][vertex][0])	
mvd_l1_sign_flag[x0][y0][vertex][0]	ae(v)
abs_mvd_l1[x0][y0][vertex][1]	ae(v)
if(abs_mvd_l1[x0][y0][vertex][1])	
mvd_l1_sign_flag[x0][y0][vertex][1]	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 != MODE_INTRA && merge_mode_flag[x0][y0] == 0 && treeType == SINGLE_TREE)	
cbf_all[x0][y0]	ae(v)
if(cbf_all[x0][y0]) {	
isSplit = log2CbWidth > MaxTbLog2SizeY || log2CbHeight > MaxTbLog2SizeY	
log2TbWidth = log2CbWidth > MaxTbLog2SizeY ? MaxTbLog2SizeY : log2CbWidth	
log2TbHeight = log2CbHeight > MaxTbLog2SizeY ? MaxTbLog2SizeY : log2CbHeight	
transform_unit(x0, y0, log2TbWidth, log2TbHeight, isSplit, cuQpDeltaCode, treeType, CuPredMode)	
if(log2CbWidth > MaxTbLog2SizeY)	
transform_unit(x0 + (1 << log2TbWidth), y0, log2TbWidth, log2TbHeight, isSplit, cuQpDeltaCode, treeType, CuPredMode)	
if(log2CbHeight > MaxTbLog2SizeY)	
transform_unit(x0, y0 + (1 << log2TbHeight), log2TbWidth, log2TbHeight, isSplit, cuQpDeltaCode, treeType, CuPredMode)	
if(log2CbWidth > MaxTbLog2SizeY && log2CbHeight > MaxTbLog2SizeY)	
transform_unit(x0 + (1 << logTbWidth), y0 + (1 << log2TbHeight), log2TbWidth, log2TbHeight, isSplit, cuQpDeltaCode, treeType, CuPredMode)	

}	
}	
}	

7.3.8.5 Transform unit syntax

transform_unit(x0, y0, log2TbWidth, log2TbHeight, isSplit, cuQpDeltaCode, treeType, CuPredMode) {	**Descriptor**
if(treeType != DUAL_TREE_LUMA && ChromaArrayType != 0) {	
cbf_cb	ae(v)
cbf_cr	ae(v)
}	
if((isSplit || CuPredMode = = MODE_INTRA || cbf_cb || cbf_cr) && treeType != DUAL_TREE_CHROMA)	
cbf_luma	ae(v)
if(cu_qp_delta_enabled_flag && (((!sps_dquant_flag || (cuQpDeltaCode = = 1 && isCuQpDeltaCoded = = 0)) && (cbf_luma || cbf_cb || cbf_cr)) || (cuQpDeltaCode = = 2 && isCuQpDeltaCoded = = 0))) {	
cu_qp_delta_abs	ae(v)
if(cu_qp_delta_abs > 0)	
cu_qp_delta_sign_flag	
}	
if(CuPredMode = = MODE_INTRA && sps_ats_flag && 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 = = MODE_INTER && 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]) {	

TrafoLog2Width = log2TbWidth − (ats_cu_inter_quad_flag[x0][y0] ? 2 : 1)	
TrafoLog2Height = log2TbHeight	
TrafoX0 = ats_cu_inter_pos_flag[x0][y0] ? 0 : (1 << log2TbWidth) − (1 << TrafoLog2Width)	
TrafoY0 = 0	
} else {	
TrafoLog2Width = log2TbWidth	
TrafoLog2Height = log2TbHeight − (ats_cu_inter_quad_flag[x0][y0] ? 2 : 1)	
TrafoX0 = 0	
TrafoY0 = ats_cu_inter_pos_flag[x0][y0] ? 0 : (1 << log2TbHeight) − (1 << TrafoLog2Height)	
}	
} else {	
TrafoX0 = 0	
TrafoY0 = 0	
TrafoLog2Width = log2TbWidth	
TrafoLog2Height = log2TbHeight	
}	
if(cbf_luma)	
residual_coding(x0 + TrafoX0, y0 + TrafoY0, TrafoLog2Width, TrafoLog2Height, 0)	
if(cbf_cb)	
residual_coding(x0 + TrafoX0, y0 + TrafoY0, TrafoLog2Width − SubWidthC + 1, TrafoLog2Height − SubHeightC + 1, 1)	
if(cbf_cr)	
residual_coding(x0 + TrafoX0, y0 + TrafoY0, TrafoLog2Width − SubWidthC + 1, TrafoLog2Height − SubHeightC + 1, 2)	
}	

7.3.8.6 Residual coding syntax

residual_coding(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	**Descriptor**
if(sps_adcc_flag == 0)	
residual_coding_rle(x0, y0, log2TbWidth, log2TbHeight, cIdx)	
else	
residual_coding_adv(x0, y0, log2TbWidth, log2TbHeight, cIdx)	
}	

7.3.8.7 Run-length residual coding syntax

residual_coding_rle(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	Descriptor
ScanPos = 0	
PrevLevel = 6	
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) * (1 − 2 * coeff_sign_flag)	
blkPos = ScanOrder[log2TbWidth][log2TbHeight][ScanPos]	
xC = blkPos & ((1 << log2TbWidth) − 1)	
yC = blkPos >> log2TbWidth	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = Level	
if(ScanPos < ((1 << (log2TbWidth + log2TbHeight)) − 1))	
coeff_last_flag	ae(v)
PrevLevel = coeff_abs_level_minus1 + 1	
ScanPos++	
} while(coeff_last_flag = = 0 && ScanPos < (1 << (log2TbWidth + log2TbHeight)))	
}	

7.3.8.8 Advanced residual coding syntax

residual_coding_adv(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 = LastSignificantCoeffX + LastSignificantCoeffY * (1 << log2TbWidth)	
scanPosLast = InvScanOrder[log2TbWidth][log2TbHeight][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][log2TbHeight][iPos]	
xC = blkPos & ((1 << log2TbWidth) − 1)	
yC = blkPos >> log2TbWidth	
if(iPos != scanPosLast)	
sig_coeff_flag[xC][yC]	ae(v)

else	
sig_coeff_flag[xC][yC] = 1	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = sig_coeff_flag[xC][yC]	
if(TransCoeffLevel[x0][y0][cIdx][xC][yC]) {	
blkPosArray[numNZ] = blkPos	
numNZ++	
}	
}	
if(numNZ > 0) {	
lastGreaterAScanPos = −1	
numC1Flag = Min(numNZ, 8)	
for(n = 0; n < numC1Flag; n++) {	
blkPos = blkPosArray[n]	
xC = blkPos & ((1 << log2TbWidth) − 1)	
yC = blkPos >> log2TbWidth	
coeff_abs_level_greaterA_flag[n]	ae(v)
TransCoeffLevel[x0][y0][cIdx][xC][yC] += 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]	
xC = blkPos & ((1 << log2TbWidth) − 1)	
yC = blkPos >> log2TbWidth	
coeff_abs_level_greaterB_flag[lastGreaterAScanPos]	ae(v)
TransCoeffLevel[x0][y0][cIdx][xC][yC] += 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]	
xC = blkPos & ((1 << log2TbWidth) − 1)	
yC = blkPos >> log2TbWidth	
baseLevel = (n < 8) ? (2 + countFirstBCoef) : 1	
if(TransCoeffLevel[x0][y0][cIdx][xC][yC] >= baseLevel) {	
coeff_abs_level_remaining[n]	ae(v)
TransCoeffLevel[x0][y0][cIdx][xC][yC] = baseLevel + coeff_abs_level_remaining	

}	
if(TransCoeffLevel[x0][y0][cIdx][xC][yC] >= 2)	
countFirstBCoef = 0	
}	
}	
coeff_signs_group	ae(v)
coeff_signs_group = coeff_signs_group << (32 – numNZ)	
for(n = 0; n < numNZ; n++) {	
blkPos = blkPosArray[n]	
xC = blkPos & ((1 << log2TbWidth) – 1)	
yC = blkPos >> log2TbWidth	
signVal = coeff_signs_group >> 31	
coeff_signs_group = coeff_signs_group << 1	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = signVal > 0 ? –TransCoeffLevel[x0][y0][cIdx][xC][yC] : TransCoeffLevel[x0][y0][cIdx][xC][yC]	
}	
}	
}	
}	

7.4 Semantics

7.4.1 General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this subclause. 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 the 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 raw bitstream file storage format. Other methods of demarcation may be specified outside of this document.

NOTE 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 raw bitstream is identical except that each NAL unit is preceded by a NAL unit length field in the raw bitstream file storage 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.

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 contain 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 4.

NOTE 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 4. The value of nal_unit_type_plus1 shall not be equal to 0. The variable NalUnitType is derived as follows:

$$NalUnitType = nal\_unit\_type\_plus1 - 1 \quad (35)$$

Table 4 — 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

NalUnitType	Name of NalUnitType	Content of NAL unit and RBSP syntax structure	NAL unit type class
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:

$$\text{TemporalId} = \text{nuh\_temporal\_id} \tag{36}$$

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, or APS_NUT, TemporalId can be greater than or equal to the TemporalId of the containing access unit, as all PPSs or APSs can 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 can be greater than or equal to the TemporalId of the containing access unit, as an SEI NAL unit can 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 subclause 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 subclauses 7.3.1, 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 document 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 subclause 7.4.2.3.3.

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

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

This subclause 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 as specified in Clauses 2 through 9.

An access unit consists of one coded picture, 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 of the previous coded picture 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_NONVCL29..RSV_NONVCL55 (when present),

— NAL units with NalUnitType in the range of UNSPEC_NUT56..UNSPEC_NUT62 (when present).

NOTE The first NAL unit preceding firstVclNalUnitInAu and succeeding the last VCL NAL unit of the previous coded picture 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 unit of the previous coded picture 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_NONVCL29..RSV_NONVCL55, or NAL units with NalUnitType in the range of UNSPEC_NUT56..UNSPEC_NUT62 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 FD_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 must 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).

toolset_idc_h indicates set of constraints to which the CVS conforms as specified in Annex A.

If a particular bit in syntax element toolset_idc_h is equal to 0, the corresponding, as per Annex A, SPS flag shall be equal to 0.

Otherwise, when the bit value in syntax element toolset_idc_h is equal to 1, the corresponding, as per Annex A, SPS flag value is not constrained.

toolset_idc_l indicates set of constraints to which the CVS conforms as specified in Annex A.

If a particular bit in syntax element toolset_idc_l is equal to 1, the corresponding, as per Annex A, SPS flag shall be equal to 1.

Otherwise, when the bit value in syntax element toolset_idc_l is equal to 0, the corresponding, as per Annex A, SPS flag value is not constrained.

The value of toolset_idc_l shall satisfy the following constraint:

— The value of (toolset_idc_l | toolset_idc_h) shall be equal to toolset_idc_h.

chroma_format_idc specifies the chroma sampling relative to the luma sampling as specified in subclause 6.2. The value of chroma_format_idc shall be in the range of 0 to 3, inclusive.

Depending on the value of chroma_format_idc, the value of the variables SubWidthC, and SubHeightC are assigned as specified in subclause 6.2 and the variable 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 Max(MinCbSizeY, 8).

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 Max(MinCbSizeY, 8).

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:

$$BitDepth_Y = 8 + bit\_depth\_luma\_minus8 \quad (37)$$

$$QpBdOffset_Y = 6 * bit\_depth\_luma\_minus8 \quad (38)$$

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:

$$BitDepth_C = 8 + bit\_depth\_chroma\_minus8 \quad (39)$$

$$QpBdOffset_C = 6 * bit\_depth\_chroma\_minus8 \quad (40)$$

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

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

log2_ctu_size_minus5 plus 5 specifies the luma coding tree block size of each CTU. When log2_ctu_size_minus5 is not present, the value of log2_ctu_size_minus5 is inferred to be equal to 1. The value of log2_ctu_size_minus5 shall be in the range of 0 to 2, inclusive.

The variables CtbLog2SizeY, CtbSizeY, PicWidthInCtbsY, PicHeightInCtbsY, PicSizeInCtbsY, PicSizeInSamplesY, PicWidthInSamplesC, and PicHeightInSamplesC are derived as follows:

$$CtbLog2SizeY = log2\_ctu\_size\_minus5 + 5 \quad (41)$$

$$CtbSizeY = 1 \ll CtbLog2SizeY \quad (42)$$

$$MaxCbLog2Size11Ratio = CtbLog2SizeY \quad (43)$$

$$MaxCbLog2Size12Ratio = MaxCbLog2Size11Ratio \quad (44)$$

$$PicWidthInCtbsY = Ceil(pic\_width\_in\_luma\_samples \div CtbSizeY) \quad (45)$$

PicHeightInCtbsY = Ceil(pic_height_in_luma_samples ÷ CtbSizeY) (46)

PicSizeInCtbsY = PicWidthInCtbsY * PicHeightInCtbsY (47)

PicSizeInSamplesY = pic_width_in_luma_samples * pic_height_in_luma_samples (48)

PicWidthInSamplesC = pic_width_in_luma_samples / SubWidthC (49)

PicHeightInSamplesC = pic_height_in_luma_samples / SubHeightC (50)

The variables MaxTbLog2SizeY, MinTbLog2SizeY, MaxTbSizeY, and MinTbSizeY are derived as follows:

MaxTbLog2SizeY = 6 (51)

MinTbLog2SizeY = 2 (52)

MaxTbSizeY = 1 << MaxTbLog2SizeY (53)

MinTbSizeY = 1 << MinTbLog2SizeY (54)

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 (55)

CtbHeightC = CtbSizeY / SubHeightC (56)

log2_min_cb_size_minus2 plus 2 specifies the minimum coding block size. When log2_min_cb_size_minus2 is not present, it is inferred to be equal to 0. The value of log2_min_cb_size_minus2 shall be in the range of 0 to CtbLog2SizeY – 2, inclusive.

The variables MinCbLog2SizeY, MinCbSizeY, PicWidthInMinCbsY, PicHeightInMinCbsY, PicSizeInMinCbsY, MinCbLog2Size11Ratio, MinCbLog2Size12Ratio, and MinCbLog2Size14Ratio are derived as follows:

MinCbLog2SizeY = 2 + log2_min_cb_size_minus2 (57)

MinCbSizeY = 1 << MinCbLog2SizeY (58)

PicWidthInMinCbsY = pic_width_in_luma_samples / MinCbSizeY (59)

PicHeightInMinCbsY = pic_height_in_luma_samples / MinCbSizeY (60)

PicSizeInMinCbsY = PicWidthInMinCbsY * PicHeightInMinCbsY (61)

MinCbLog2Size11Ratio = MinCbLog2SizeY (62)

MinCbLog2Size12Ratio = MinCbLog2Size11Ratio + 1 (63)

MinCbLog2Size14Ratio = MinCbLog2Size12Ratio + 1 (64)

log2_diff_ctu_max_14_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 or greater than 1:4 or 4:1. When log2_diff_ctu_max_14_cb_size is not present, it is inferred to be equal to 0. The value of log2_diff_ctu_max_14_cb_size shall be in the range of 0 to CtbLog2SizeY – MinCbLog2Size14Ratio + 1, inclusive.

The variable MaxCbLog2Size14Ratio is derived as follows:

$$MaxCbLog2Size14Ratio = Min(CtbLog2SizeY - log2\_diff\_ctu\_max\_14\_cb\_size, MaxTbLog2SizeY) \quad (65)$$

log2_diff_ctu_max_tt_cb_size specifies the difference between the luma coding tree block size and the maximum coding block size allowing ternary tree split. When log2_diff_ctu_max_tt_cb_size is not present, it is inferred to be equal to 0. The value of log2_diff_ctu_max_tt_cb_size shall be in the range of 0 to CtbLog2SizeY – MinCbLog2Size14Ratio, inclusive.

The variable MaxTtLog2Size is derived as follows:

$$MaxTtLog2Size = Min(CtbLog2SizeY - log2\_diff\_ctu\_max\_tt\_cb\_size, MaxTbLog2SizeY) \quad (66)$$

log2_diff_min_cb_min_tt_cb_size_minus2 plus 2 specifies the difference between the minimum coding block size and the minimum coding block size allowing ternary tree split. When log2_diff_min_cb_min_tt_cb_size_minus2 is not present, it is inferred to be equal to 0. The value of log2_diff_min_cb_min_tt_cb_size_minus2 shall be in the range of 0 to MaxTtLog2Size – MinCbLog2SizeY – 1, inclusive.
The variable MinTtLog2Size is derived as follows:

$$MinTtLog2Size = MinCbLog2SizeY + 2 + log2\_diff\_min\_cb\_min\_tt\_cb\_size\_minus2 \quad (67)$$

sps_suco_flag equal to 1 specifies that the split unit coding ordering is used. sps_suco_flag equal to 0 specifies that the 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 CtbLog2SizeY – MinCbLog2SizeY. The value of log2_diff_ctu_size_max_suco_cb_size shall be in the range of 0 to CtbLog2SizeY – MinCbLog2SizeY, inclusive.

The variable MaxSucoLog2Size is derived as follows:

$$MaxSucoLog2Size = Min(CtbLog2SizeY - log2\_diff\_ctu\_size\_max\_suco\_cb\_size, 6) \quad (68)$$

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 value of log2_diff_max_suco_min_suco_cb_size shall be in the range of 0 to MaxSucoLog2Size – MinCbLog2SizeY, inclusive.

The variable MinSucoLog2Size is derived as follows:

$$MinSucoLog2Size = Max(MaxSucoLog2Size - log2\_diff\_max\_suco\_min\_suco\_cb\_size, Max(4, MinCbLog2SizeY)) \quad (69)$$

sps_admvp_flag equal to 1 specifies that the advanced motion vector prediction, signalling and interpolation are used. sps_admvp_flag equal to 0 specifies that the advanced motion vector prediction, signalling and interpolation are not used.

sps_affine_flag equal to 1 specifies that the affine model based motion compensation can be used for inter prediction. sps_affine_flag equal to 0 specifies that the syntax shall be constrained such that no affine model based motion compensation is used in the CVS, and affine_flag, affine_merge_idx and affine_mode_flag are not present in the coding unit syntax of the CVS. When sps_affine_flag is not present, it is inferred to be equal to 0.

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

sps_dmvr_flag equal to 1 specifies that decoder-side motion vector refinement can be used. sps_dmvr_flag equal to 0 specifies that decoder-side motion vector refinement is not used. When sps_dmvr_flag is not presented, it is inferred to be equal to 0.

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

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

sps_ibc_flag equal to 1 specifies that the intra block copy can be used. sps_ibc_flag equal to 0 specifies that the intra block copy is not used. When sps_ibc_flag is not present, it is inferred to be equal to 0.

log2_max_ibc_cand_size_minus2 plus 2 specifies the maximum block size of the intra block copy mode as follows:

$$log2MaxIbcCandSize = 2 + log2\_max\_ibc\_cand\_size\_minus2 \quad (70)$$

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

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

sps_adcc_flag equal to 1 specifies that the advanced residual coding specified in subclause 7.3.8.8 is used. sps_adcc_flag equal to 0 specifies that the run-length residual coding specified in subclause 7.3.8.7 is used. When sps_adcc_flag is not presented, it is inferred to be equal to 0.

For log2TbWidth ranging from 1 to MaxTbLog2SizeY, inclusive, and log2TbHeight ranging from 1 to MaxTbLog2SizeY, inclusive, 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 position 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.

For log2TbWidth ranging from 1 to MaxTbLog2SizeY, inclusive, and log2TbHeight ranging from 1 to MaxTbLog2SizeY, inclusive, the array ScanOrder is derived by invoking subclause 6.5.2 with input parameters blkWidth equal to (1 << log2TbWidth) and blkHeight equal to (1 << log2TbHeight) and the array InvScanOrder is derived by invoking subclause 6.5.3 with input parameters blkWidth equal to (1 << log2TbWidth) and blkHeight equal to (1 << log2TbHeight).

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

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_addb_flag equal to 1 specifies that the advanced deblocking filter specified in subclause 8.8.3 can be applied. sps_addb_flag equal to 0 specifies that the deblocking filter specified in subclause 8.8.2 can be applied.

sps_alf_flag equal to 1 specifies that the adaptive loop filter can be applied. sps_alf_flag equal to 0 specifies that the adaptive loop filter is not applied.

sps_htdf_flag equal to 1 specifies that the hadamard transform domain filter can be applied. sps_htdf_flag equal to 0 specifies that the hadamard transform domain filter is not applied.

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 that the improved delta qp signalling processes is used. sps_dquant_flag equal to 0 specifies that the improved delta qp signalling process is not used. sps_dquant_flag shall be equal to 0 when both sps_btt_flag and sps_admvp_flag are equal to 0.

sps_dra_flag equal to 1 specifies that the dynamic range adjustment mapping on output samples can be used. sps_dra_flag equal to 0 specifies that dynamic range adjustment mapping on output samples 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:

$$MaxPicOrderCntLsb = 2^{(log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4)} \quad (71)$$

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:

$$SubGopLength = 2^{(log2\_sub\_gop\_length)} \quad (72)$$

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:

$$RefPicGapLength = 2^{(log2\_ref\_pic\_gap\_length)} \quad (73)$$

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

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 For each value of listIdx (equal to 0 or 1), a decoder must 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 can 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.

picture_crop_left_offset, **picture_crop_right_offset**, **picture_crop_top_offset** and **picture_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 picture_crop_left_offset, picture_crop_right_offset, picture_crop_top_offset and picture_crop_bottom_offset are inferred to be equal to 0.The conformance cropping window contains the luma samples with horizontal picture coordinates from SubWidthC * picture_crop_left_offset to picture_width_in_luma_samples − (SubWidthC * picture_crop_right_offset + 1) and vertical picture coordinates from SubHeightC * picture_crop_top_offset to picture_height_in_luma_samples − (SubHeightC * picture_crop_bottom_offset + 1), inclusive.

The value of SubWidthC * (picture_crop_left_offset + picture_crop_right_offset) shall be less than picture_width_in_luma_samples, and the value of SubHeightC * (picture_crop_top_offset + picture_crop_bottom_offset) shall be less than picture_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.

chroma_qp_table_present_flag equal to 1 specifies that user defined chroma QP mapping tables ChromaQpTable are signalled. chroma_qp_table_present_flag equal to 0 specifies that user defined chroma QP mapping tables are not signalled and variable ChromaQpTable are initialized as follows:

— If ChromaArrayType is equal to 1, the following applies:

  — If sps_iqt_flag is equal to 0, the variables ChromaQpTable[m][qPi] with m being equal to 0 and 1, and qPi being in the range of −QpBdOffset$_C$ to 57 are set equal to the value of Qp$_C$ as specified in Table 5 based on the index qPi.

  — Otherwise (sps_iqt_flag is equal to 1), the variables ChromaQpTable[m][qPi] with m equal to 0 and 1, and qPi being in the range of −QpBdOffset$_C$ to 57 set equal to the value of Qp$_C$ as specified in Table 6 based on the index qPi.

— Otherwise, ChromaQpTable[m][qPi] with m being equal to 0 and 1, and qPi being in the range of −QpBdOffset$_C$ to 57 are set equal to the value of qPi.

Table 5 — 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 6 — 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
Qp$_C$	= qPi	29	30	31	32	33	34	35	36	37	37	38	39	40	40	= qPi − 3

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

same_qp_table_for_chroma equal to 1 specifies that only one chroma QP mapping table is signalled and applies to both Cb and Cr components. same_qp_table_for_chroma equal to 0 specifies that two chroma QP mapping tables for Cb and Cr respectively are signalled in the SPS.

global_offset_flag equal to 1 specifies that the first pivot point in the chroma QP mapping tables have an input coordinate of at least 16 and output coordinate of at least 16. global_offset_flag equal to 0 specifies that there are no restrictions on the first pivot point in the chroma QP mapping tables.

num_points_in_qp_table_minus1[i] plus 1 specifies the number of points used to describe the chroma QP mapping table. The value of num_points_in_qp_table_minus1[i] shall be in the range of 0 to 57 + QpBdOffset$_C$ − (global_offset_flag = = 1 ? 16 : 0), inclusive.

delta_qp_in_val_minus1[i][j] plus 1 specifies a delta value used to derive the input coordinate of the j-th pivot point of the i-th chroma QP mapping table.

delta_qp_out_val[i][j] specifies a delta value used to derive the output coordinate of the j-th pivot point of the i-th chroma QP mapping table.

The i-th chroma QP mapping tables ChromaQpTable[i] for i = 0..(same_qp_table_for_chroma ? 0 : 1) are derived as follows:

```
startQp = ( global_offset_flag  = =  1 ) ? 16 : −QpBdOffsetC
qpInVal[ i ][ 0 ] = startQP + delta_qp_in_val_minus1[ i ][ 0 ]
qpOutVal[ i ][ 0 ] = startQP + delta_qp_in_val_minus1[ i ][ 0 ] + delta_qp_out_val[ i ][ 0 ]
for( j = 1; j <= num_points_in_qp_table_minus1[ i ]; j++ ) {
    qpInVal[ i ][ j ] = qpInVal[ i ][ j − 1 ] + delta_qp_in_val_minus1[ i ][ j ] + 1
    qpOutVal[ i ][ j ] = qpOutVal[ i ][ j − 1 ] + ( delta_qp_in_val_minus1[ i ][ j ] + 1 −
        delta_qp_out_val[ i ][ j ]                                                          (74)
}
ChromaQpTable[ i ][ qpInVal[ i ][ 0 ] ] = qpOutVal[ i ][ 0 ]
for( k = qpInVal[ i ][ 0 ] − 1; k >= −QpBdOffsetC; k− − )
    ChromaQpTable[ i ][ k ] = Clip3( −QpBdOffsetC, 57, ChromaQpTable[ i ][ k + 1 ] − 1 )

for( j = 0 ; j < num_points_in_qp_table_minus1[ i ]; j++ ) {
    sh = ( delta_qp_in_val_minus1[ i ][ j + 1 ] + 1 ) >> 1
    for( k = qpInVal[ i ][ j ] + 1, m = 1; k <= qpInval[ i ][ j + 1 ]; k++, m++ )
        ChromaQpTable[ i ][ k ] = ChromaQpTable[ i ][ qpInVal[ i ][ j ] ] +
            ( delta_qp_out_val[ i ][ j + 1 ] * m + sh ) / ( delta_qp_in_val_minus1[ i ][ j + 1 ] + 1 )
}

for( k = qpInVal[ i ][ num_points_in_qp_table_minus1[ i ] ] + 1; k <= 57; k++ )
    ChromaQpTable[ i ][ k ] = Clip3( −QpBdOffsetC, 57, ChromaQpTable[ i ][ k − 1 ] + 1 )
```

When same_qp_table_for_chroma is equal to 1, ChromaQpTable[1][k] is set equal to ChromaQpTable[0][k] for k = −QpBdOffsetC..57.

The values of qpInVal[i][j] and qpOutval[i][j] shall be in the range of −QpBdOffsetC to 57, inclusive, for i = 0..(same_qp_table_for_chroma ? 0 : 1) and j = 0..num_points_in_qp_table_minus1[i].

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:

$$MaxLtPicOrderCntLsb = 2^{(\ log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 + additional\_lt\_poc\_lsb\_len\ )} \quad (75)$$

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 long_term_ref_pics_flag is equal to 0, 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.

When sps_rlp_flag is equal to 0, rpl1_idx_present_flag shall be equal to 0.

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.

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_column_width_minus1 shall be in the range of 0 to PicWidthInCtbsY – num_tile_columns_minus1 – 1, inclusive. The sum of (tile_column_width_minus1[i] + 1) for i ranging from 0 to num_tile_columns_minus1 – 1 is less than PicWidthInCtbsY.

tile_row_height_minus1[i] plus 1 specifies the height of the i-th tile row in units of CTBs. tile_row_height_minus1 shall be in the range of 0 to PicHeightInCtbsY – num_tile_rows_minus1 – 1, inclusive. The sum of (tile_row_height_minus1[i] + 1) for i ranging from 0 to num_tile_rows_minus1 – 1 is less than PicHeightInCtbsY.

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)) – 1 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 subclause 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.

pic_dra_enabled_flag equal to 1 specifies that DRA is enabled for all decoded picture referring to the PPS. pic_dra_enabled_flag equal to 0 specifies that DRA is not enabled for all decoded pictures referring to the PPS. When not present, pic_dra_enabled_flag is inferred to be equal to 0. When sps_dra_flag is equal to 0, pic_dra_enabled_flag shall be equal to 0.

pic_dra_aps_id specifies the adaptation_parameter_set_id of the DRA APS that is enabled for decoded pictures referring to the PPS.

The TemporalId of the APS NAL unit having aps_params_type equal to DRA_APS and adaptation_parameter_set_id equal to pic_dra_aps_id shall be less than or equal to the TemporalId of the picture associated with the PPS.

When multiple APSs of DRA_APS type with the same value of adaptation_parameter_set_id are referred to by two or more pictures within a CVS, the multiple APSs of type DRA_APS with the same value of adaptation_parameter_set_id shall have the same content.

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_abs and cu_qp_delta_sign_flag 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_abs and cu_qp_delta_sign_flag are 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:

$$cuQpDeltaArea = log2\_cu\_qp\_delta\_area\_minus6 + 6 \quad (76)$$

The value of log2_cu_qp_delta_area_minus6 shall be in the range of 0 to 2 * CtbLog2SizeY – 6, inclusive.

7.4.3.3 APS RBSP semantics

Each APS RBSP shall be available to the decoding process prior to it being referenced, included in at least one access unit with TemporalId less than or equal to the TemporalId of the coded slice NAL unit that refers it or provided through external means.

For all APS NAL units with aps_param_type equal to ALF_APS or DRA_APS, it is the requirement of bitstream conformance that number of APS NAL units within an access unit shall not exceed 32 for each of the APS types.

adaptation_parameter_set_id provides an identifier for the APS for reference by other syntax elements.

APS NAL units with different values of aps_params_type use separate values spaces for adaptation_parameter_set_id.

When aps_params_type is equal to ALF_APS or DRA_APS, the value of adaptation_parameter_set_id shall be in the range of 0 to 31, inclusive.

All APS NAL units with aps_param_type equal to ALF_APS and a particular value of adaptation_parameter_set_id within an access unit, shall have the same content.

All APS NAL units with aps_param_type equal to DRA_APS and a particular value of adaptation_parameter_set_id within a CVS, shall have the same content.

NOTE An APS NAL unit (with a particular value of adaptation_parameter_set_id and a particular value of aps_params_type) can be shared across pictures, and different slices within a picture can refer to different ALF APSs.

aps_params_type specifies the type of APS parameters carried in the APS as specified in Table 7.

Table 7 — APS parameters type codes and types of APS parameters

aps_params_type	Name of aps_params_type	Type of APS parameters
0	ALF_APS	ALF parameters
1	DRA_APS	DRA parameters
2..7	Reserved	Reserved

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. Bitstreams conforming to this version of this document shall have aps_extension_flag value be equal to 0.

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 document. Decoders conforming to this version of this document 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.

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

Let the variable BinCountsInPicNalUnits be the number of times that the parsing process function DecodeBin( ), specified in subclause 9.3.4.3, is invoked to decode the contents of all VCL NAL units of a coded picture.

Let the variable RawMinCuBits be derived as follows:

$$RawMinCuBits = MinCbSizeY * MinCbSizeY * (BitDepth_Y + ((ChromaArrayType != 0) ? 2 * BitDepth_C / (SubWidthC * SubHeightC) : 0)) \quad (77)$$

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

NOTE The constraint on the maximum number of bins resulting from decoding the contents of the coded slice segment NAL units can be met by inserting a number of cabac_zero_word syntax elements to increase the value of NumBytesInVclNalUnits.

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.

payload_type_byte is a byte of the payload type of an SEI message.

payload_size_byte is a 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. When not present, the value of single_tile_in_slice_flag is inferred to be equal to 1.

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

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 )                    (78)
        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++ ) {
    if( currTileIdx / ( num_tile_columns_minus1 + 1 ) >
                        tileIdx / ( num_tile_columns_minus1 + 1 ) )                    (79)
        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:

$$NumTilesInSlice = num\_remaining\_tiles\_in\_slice\_minus1 + 2 \quad (80)$$

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   (81)

liceTileIdx[ i ] = TileIdToIdx[ sliceTileId[ i ] ]                                                        (82)
```

slice_type specifies the coding type of the slice according to Table 8.

Table 8 — 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. When ChromaArrayType is in the range from 0 to 2, inclusive, slice_alf_enabled_flag equal to 0 specifies that adaptive loop filter is disabled for all colour components in a slice. When ChromaArrayType is equal to 3, slice_alf_enabled_flag equal to 0 specifies that adaptive loop

filter is disabled for Y components in a slice. When not present, the value of slice_alf_enabled_flag is inferred to be equal to 0.

slice_alf_chroma_idc equal to 0 specifies that the ALF is not applied to Cb and Cr colour components in a slice. slice_alf_chroma_idc equal to 1 indicates that the adaptive loop filter can be applied to the Cb colour component, slice_alf_chroma_idc equal to 2 indicates that the adaptive loop filter can be applied to the Cr colour component, and slice_alf_chroma_idc equal to 3 indicates that the adaptive loop filter can be applied to Cb and Cr colour components. When alf_chroma_idc is not present, it is inferred to be equal to 0.

When slice_alf_chroma_idc is equal to 0, the variables sliceChromaAlfEnabledFlag and sliceChroma2AlfEnabledFlag are set equal to 0.

If slice_alf_chroma_idc is equal to 1 or 3, the variable sliceChromaAlfEnabledFlag is set equal to 1. Otherwise, the variables sliceChromaAlfEnabledFlag is set equal to 0.

If slice_alf_chroma_idc is equal to 2 or 3, the variable sliceChroma2AlfEnabledFlag is set equal to 1. Otherwise, the variables sliceChroma2AlfEnabledFlag is set equal to 0.

When ChromaArrayType is equal to 0, it is requirement of bitstream conformance that the value of slice_alf_chroma_idc shall be equal to 0.

slice_alf_luma_aps_id specifies the adaptation_parameter_set_id of the ALF APS to be used for filtering of luma component of the current slice. The TemporalId of the ALF APS NAL unit having adaptation_parameter_set_id equal to slice_alf_luma_aps_id shall be less than or equal to the TemporalId of the coded slice NAL unit. The value of alf_luma_filter_signal_flag of the APS NAL unit having aps_params_type equal to ALF_APS and adaptation_parameter_set_id equal to slice_alf_luma_aps_id shall be equal to 1.

slice_alf_chroma_aps_id specifies the adaptation_parameter_set_id of the ALF APS to be used in the current slice for filtering of Cb chroma component, when ChromaArrayType is in the range from 1 to 3, inclusive, and Cr chroma components when ChromaArrayType is in the range from 1 to 2, inclusive. The TemporalId of the ALF APS NAL unit having adaptation_parameter_set_id equal to slice_alf_chroma_aps_id shall be less than or equal to the TemporalId of the coded slice NAL unit.

When ChromaArrayType is equal to 3, the value of alf_luma_filter_signal_flag of the APS NAL unit having aps_params_type equal to ALF_APS and adaptation_parameter_set_id equal to slice_alf_chroma_aps_id shall be equal to 1.

When ChromaArrayType is equal to 1 or 2, the value of alf_chroma_filter_signal_flag of the APS NAL unit having aps_params_type equal to ALF_APS and adaptation_parameter_set_id equal to slice_alf_chroma_aps_id shall be equal to 1.

slice_alf_chroma2_aps_id specifies the adaptation_parameter_set_id of the ALF APS to be used for filtering of chroma Cr component of the current slice, when ChromaArrayType is equal to 3. The TemporalId of the ALF APS NAL unit having adaptation_parameter_set_id equal to slice_alf_chroma2_aps_id shall be less than or equal to the TemporalId of the coded slice NAL unit.

The value of alf_luma_filter_signal_flag of the APS NAL unit having aps_params_type equal to ALF_APS and adaptation_parameter_set_id equal to slice_alf_chroma2_aps_id shall be equal to 1.

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

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

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

slice_alf_chroma2_map_flag equal to 1 specifies that adaptive loop filter applicability map for Cr component is signalled. slice_alf_chroma2_map_flag equal to 0 specifies that adaptive loop filter applicability map for Cr component is not signalled. When not present, the value of slice_alf_chroma2_map_flag 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:

$$SliceRplsIdx[i] = ref\_pic\_list\_sps\_flag[i] \; ? \; ref\_pic\_list\_idx[i] : num\_ref\_pic\_lists\_in\_sps[i] \quad (83)$$

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:

$$FullPocLsbLt[i][SliceRplsIdx[i]][j] = additional\_poc\_lsb\_val[i][j] * MaxPicOrderCntLsb + poc\_lsb\_lt[i][SliceRplsIdx[i]][j] \quad (84)$$

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:

$$NumRefIdxActive[i] = num\_ref\_idx\_active\_minus1[i] + 1 \quad (85)$$

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

col_pic_list_idx specifies the reference picture list for currently coded slice 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_source_mvp_list_idx specifies the reference picture list in collocated location 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.

col_pic_ref_idx specifies the reference index used for temporal motion vector prediction. col_pic_ref_idx refers to a picture in RefPicList[col_pic_list_idx] of the current picture, and the value of col_pic_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.

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 Table 34 and Table 35 for deblocking filtering operations the slice. From this value, the offset that shall be applied when addressing these tables shall be computed as:

$$FilterOffsetA = slice\_alpha\_offset \quad (86)$$

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 Table 34 for deblocking filtering operations within the slice. From this value, the offset that is applied when addressing Table 34 of the deblocking filter shall be computed as:

$$\text{FilterOffsetB} = \text{slice\_beta\_offset} \tag{87}$$

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, Qp_Y 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 Qp_Y used for deriving Qp_{Cb} and Qp_{Cr}, 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. 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:

$$\text{firstByte}[\,k\,] = \sum_{n=1}^{k} (\text{entry\_point\_offset\_minus\_1}[n-1] + 1) \tag{88}$$

$$\text{lastByte}[\,k\,] = \text{firstByte}[\,k\,] + \text{entry\_point\_offset\_minus1}[\,k\,] \tag{89}$$

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.

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

alf_chroma_filter_signal_flag equal to 1 specifies that chroma filter is signalled. alf_chroma_filter_signal_flag equal to 0 specifies that chroma filter is not signalled. When alf_chroma_filter_signal_flag is not present, it is inferred to be equal to 0. When ChromaArrayType is equal to 0 or 3, it is requirement of bitstream conformance that the value of alf_chroma_filter_signal_flag shall be equal to 0.

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

alf_luma_num_filters_signalled_minus1 plus 1 specifies the number of 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 variable NumSignalledFilter is set equal to alf_luma_num_filters_signalled_minus1 + 1.

alf_luma_type_flag specifies a type of luma filter. When alf_luma_type_flag is equal to 0, the value LumaMaxGolombIdx 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 LumaMaxGolombIdx is set equal to 3 and the variable NumAlfCoefs is set equal to 13.

The array coefPosMap[ ] specified for follows:

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

coefPosMap[ ] = { 0, 0, 1, 0, 0, 2, 3, 4, 0, 0, 5, 6, 7 } (90)

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

coefPosMap[ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 } (91)

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, inclusive. When alf_luma_coeff_delta_idx[i] is not present, it is inferred to be equal to 0.

The codeword length of alf_luma_coeff_delta_idx[i] is Ceil(Log2(alf_luma_num_filters_signalled_minus1 + 1)) bits.

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_flag[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_flag[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_flag[i] syntax elements. The value i shall be in the range from 0 to NumAlfFilters − 1, inclusive. The order k of the exp-Golomb code used to decode the value of alf_luma_fixed_filter_usage_pattern is equal to 0.

alf_luma_fixed_filter_usage_flag[i] equal to 1 specifies i-th filter uses fixed filter. alf_luma_fixed_filter_usage_flag[i] equal to 0 specifies i-th filter does not use fixed filter. When it is not present, alf_luma_fixed_filter_usage_flag[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_flag[i] is inferred to be equal to 1 if alf_luma_fixed_filter_usage_pattern is equal to 1.

alf_luma_fixed_filter_set_idx specifies a fixed filter set index. The value of alf_luma_fixed_filter_set_idx shall be in a range of 0 to 15, inclusive.

alf_luma_coeff_delta_flag equal to 1 indicates that alf_luma_coeff_delta_prediction_flag is not signalled and syntax elements alf_luma_coeff_flag[i] are signalled. alf_luma_coeff_delta_flag equal to 0 indicates that alf_luma_coeff_delta_prediction_flag may be signalled and syntax elements alf_luma_coeff_flag[i] are not 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.

The variable kMin is set equal to alf_luma_min_eg_order_minus1 + 1.

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.

With i being in range from 0 to LumaMaxGolombIdx – 1, inclusive, 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] = kMin + alf_luma_eg_order_increase_flag[i] (92)

kMin = expGoOrderY[i] (93)

alf_luma_coeff_flag[i] equal to 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 to 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 } (94)

k = expGoOrderY[golombOrderIdxY[j]] (95)

alf_luma_coeff_delta_sign_flag[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_flag[i][j] is equal to 0, the corresponding luma filter coefficient has a negative value.

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

When alf_luma_coeff_delta_sign_flag[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] * (2 * alf_luma_coeff_delta_sign_flag[i][j] – 1) (96)

When alf_luma_coeff_delta_prediction_flag is equal to 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] (97)

The luma filter coefficients $AlfCoeff_L$[adaptation_parameter_set_id] with elements $AlfCoeff_L$[adaptation_parameter_set_id][filtIdx][j], with filtIdx = 0..NumAlfFilters – 1 and coefficient position index j = 0..11 are derived as follows:

— If alf_luma_fixed_filter_usage_flag[filtIdx] is equal to 1, the following applies:

outCoef[filtIdx][j] =
AlfFixFiltCoeff[AlfClassToFiltMap[filtIdx][alf_luma_fixed_filter_set_idx[filtidx]]][j] (98)

— Otherwise (alf_luma_fixed_filter_usage_flag[filtIdx] is equal to 0), the following applies:

outCoef[filtIdx][j] = 0 (99)

— For every j with coefPosMap[j] larger than 0, outCoef[j] is updated as follows:

outCoef[filtIdx][j] += filterCoefficients[alf_luma_coeff_delta_idx[filtIdx]][coefPosMap[j] – 1] (100)

— The luma filter coefficients $AlfCoeff_L$[adaptation_parameter_set_id][filtIdx][j] is derived as follows:

$AlfCoeff_L$[adaptation_parameter_set_id][filtIdx][j] = outCoef[j] (101)

The fixed filter coefficients AlfFixFiltCoeff[i][j] with i = 0..63, j = 0..11 and the class to filter mapping AlfClassToFiltMap[m][n] with m = 0..24 and n = 0..15 are defined as follows:

AlfFixFiltCoeff[i][j] = (102)

```
{
  {   0,   2,   7, -12,  -4, -11,  -2,  31,  -9,   6,  -4,  30 },
  { -26,   4,  17,  22,  -7,  19,  40,  47,  49, -28,  35,  48 },
  { -24,  -8,  30,  64, -13,  18,  18,  27,  80,   0,  31,  19 },
  {  -4, -14,  44, 100,  -7,   6,  -4,   8,  90,  26,  26, -12 },
  { -17,  -9,  23,  -3, -15,  20,  53,  48,  16, -25,  42,  66 },
  { -12,  -2,   1, -19,  -5,   8,  66,  80,  -2, -25,  20,  78 },
  {   2,   8, -23, -14,  -3, -23,  64,  86,  35, -17,  -4,  79 },
  {  12,   4, -39,  -7,   1, -20,  78,  13,  -8,  11, -42,  98 },
  {   0,   3,  -4,   0,   2,  -7,   6,   0,   0,   3,  -8,  11 },
  {   4,  -7, -25, -19,  -9,   8,  86,  65, -14,  -7,  -7,  97 },
  {   3,   3,   2, -30,   6, -34,  43,  71, -10,   4, -23,  77 },
  {  12,  -3, -34, -14,  -5, -14,  88,  28, -12,   8, -34, 112 },
  {  -1,   6,   8, -29,   7, -27,  15,  60,  -4,   6, -21,  39 },
  {   8,  -1,  -7, -22,   5, -41,  63,  40, -13,   7, -28, 105 },
  {   1,   3,  -5,  -1,   1, -10,  12,  -1,   0,   3,  -9,  19 },
  {  10,  -1, -23, -14,  -3, -27,  78,  24, -14,   8, -28, 102 },
  {   0,   0,  -1,   0,   0,  -1,   1,   0,   0,   0,   0,   1 },
  {   7,   3, -19,  -7,   2, -27,  51,   8,  -6,   7, -24,  64 },
  {  11, -10, -22, -22, -11, -12,  87,  49, -20,   4, -16, 108 },
  {  17,  -2, -69,  -4,  -4,  22, 106,  31,  -7,  13, -63, 121 },
  {   1,   4,  -1,  -7,   5, -26,  24,   0,   1,   3, -18,  51 },
  {   3,   5, -10,  -2,   4, -17,  17,   1,  -2,   6, -16,  27 },
  {   9,   2, -23,  -5,   6, -45,  90, -22,   1,   7, -39, 121 },
  {   4,   5, -15,  -2,   4, -22,  34,  -2,  -2,   7, -22,  48 },
  {   6,   8, -22,  -3,   4, -32,  57,  -3,  -4,  11, -43, 102 },
  {   2,   5, -11,   1, -12, -46,  64, -32,   7,   4, -31,  85 },
  {   5,   5, -12,  -8,   6, -48,  74, -13,  -1,   7, -41, 129 },
  {   0,   1,  -1,   0,   1,  -3,   2,   0,   0,   1,  -3,   4 },
  {  -1,   3,  16, -42,   6, -16,   2, 105,   6,   6, -31,  43 },
  {   7,   8, -27,  -4,  -4, -23,  46,  79,  64,  -8, -13,  68 },
  {  -3,  12,  14, -34,  14,  -6, -24, 179,  56,   2, -48,  15 },
  {   8,   0, -16, -25,  -1, -29,  68,  84,   3,  -3, -18,  94 },
  {  -3,  -1,  22, -32,   2, -20,   5,  89,   0,   9, -18,  40 },
  {  14,   6, -51,  22, -10, -22,  36,  75, 106,  -4, -11,  56 },
  {   1,  38, -59,  14,   8, -44, -18, 156,  80,  -1, -42,  29 },
  {  -1,   2,   4,  -9,   3, -13,   7,  17,  -4,   2,  -6,  17 },
  {  11,  -2, -15, -36,   2, -32,  67,  89, -19,  -1, -14, 103 },
  {  -1,  10,   3, -28,   7, -27,   7, 117,  34,   1, -35,  51 },
  {   3,   3,   4, -18,   6, -40,  36,  18,  -8,   7, -25,  86 },
  {  -1,   3,   9, -18,   5, -26,  12,  37, -11,   3,  -7,  32 },
  {   0,  17, -38,  -9, -28, -17,  25,  48, 103,   2,  40,  69 },
  {   6,   4, -11, -20,   5, -32,  51,  77,  17,   0, -25,  84 },
  {   0,  -5,  28, -24,  -1, -22,  18,  -9,  17,  -1, -12, 107 },
  { -10,  -4,  17, -30, -29,  31,  40,  49,  44, -26,  67,  67 },
  { -30, -12,  39,  15, -21,  32,  29,  26,  71,  20,  43,  28 },
  {   6,  -7,  -7, -34, -21,  15,  53,  60,  12, -26,  45,  89 },
  {  -1,  -5,  59, -58,  -8, -30,   2,  17,  34,  -7,  25, 111 },
  {   7,   1,  -7, -20,  -9, -22,  48,  27,  -4,  -6,   0, 107 },
  {  -2,  22,  29, -70,  -4, -28,   2,  19,  94, -40,  14, 110 },
```

```
  {  13,   0, -22, -27, -11, -15,  66,  44,  -7,  -5, -10, 121 },
  {  10,   6, -22, -14,  -2, -33,  68,  15,  -9,   5, -35, 135 },
  {   2,  11,   4, -32,  -3, -20,  23,  18,  17,  -1, -28,  88 },
  {   0,   3,  -2,  -1,   3, -16,  16,  -3,   0,   2, -12,  35 },
  {   1,   6,  -6,  -3,  10, -51,  70, -31,   5,   6, -42, 125 },
  {   5,  -7,  61, -71, -36,  -6,  -2,  15,  57,  18,  14, 108 },
  {   9,   1,  35, -70, -73,  28,  13,   1,  96,  40,  36,  80 },
  {  11,  -7,  33, -72, -78,  48,  33,  37,  35,   7,  85,  76 },
  {   4,  15,   1, -26, -24, -19,  32,  29,  -8,  -6,  21, 125 },
  {  11,   8,  14, -57, -63,  21,  34,  51,   7,  -3,  69,  89 },
  {   7,  16,  -7, -31, -38,  -5,  41,  44, -11, -10,  45, 109 },
  {   5,  16,  16, -46, -55,   3,  22,  32,  13,   0,  48, 107 },
  {   2,  10,  -3, -14,  -9, -28,  39,  15, -10,  -5,  -1, 123 },
  {   3,  11,  11, -27, -17, -24,  18,  22,   2,   4,   3, 100 },
  {   0,   1,   7,  -9,   3, -20,  16,   3,  -2,   0,  -9,  61 },
}
```

AlfClassToFiltMap[m][n] = (103)

```
{
  { 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 $AlfCoeff_L[\ adaptation\_parameter\_set\_id\ ][\ filtIdx\ ][\ 12\ ]$ for filtIdx = 0..NumAlfFilters − 1 are derived as follows:

$$AlfCoeff_L[\ adaptation\_parameter\_set\_id\ ][\ filtIdx\ ][\ 12\ ] = 512 - \Sigma_k\ (\ AlfCoeff_L[\ adaptation\_parameter\_set\_id\ ][\ filtIdx\ ][\ k\ ]\ <<\ 1\ ),\ \text{with}\ k = 0..NumAlfCoefs - 2 \quad (104)$$

The values of $AlfCoeff_L[\ adaptation\_parameter\_set\_id\ ][\ filtIdx\ ][\ j\ ]$ with filtIdx = 0..NumAlfFilters − 1, j = 0..11 shall be in the range of -2^9 to $2^9 - 1$, inclusive and the values of $AlfCoeff_L[\ adaptation\_parameter\_set\_id\ ][\ filtIdx\ ][\ 12\ ]$ shall be in the range of -2^{10} to $2^{10} - 1$, 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 variable kMin is set equal to alf_chroma_min_eg_order_minus1 + 1 and the value ChromaMaxGolombIdx is set equal to 2.

With i being in range from 0 to ChromaMaxGolombIdx – 1, inclusive, 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] = kMin + alf_chroma_eg_order_increase_flag[i] (105)

kMin = expGoOrderY[i] (106)

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 to 0.

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

golombOrderIdxC[ ] = { 0, 0, 1, 0, 0, 1 } (107)

k = expGoOrderC[golombOrderIdxC[j]] (108)

alf_chroma_coeff_sign_flag[j] specifies the sign of the j-th chroma filter coefficient as follows:

— If alf_chroma_coeff_sign_flag[j] is equal to 0, the corresponding chroma filter coefficient has a negative value.

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

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

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

$AlfCoeff_C$[adaptation_parameter_set_id][j] = alf_chroma_coeff_abs[j] *
(2 * alf_chroma_coeff_sign_flag[j] – 1) (109)

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

$AlfCoeff_C$[adaptation_parameter_set_id][6] =
512 – Σ_k ($AlfCoeff_C$[adaptation_parameter_set_id][k] << 1), with k = 0..5 (110)

The values of $AlfCoeff_C$[adaptation_parameter_set_id][j] with altIdx = 0..alf_chroma_num_alt_filters_minus1, j = 0..5 shall be in the range of -2^9 to $2^9 - 1$, inclusive and the values of $AlfCoeff_C$[adaptation_parameter_set_id][6] shall be in the range of -2^{10} to $2^{10} - 1$, inclusive.

7.4.7 DRA data semantics

dra_descriptor1 specifies the number of bits used to represent the integer part of the DRA scale parameters signalling. The value of dra_descriptor1 shall be in the range of 0 to 15, inclusive. In the current version of the specification value of syntax element dra_descriptor1 is restricted to 4; other values are reserved for future use.

dra_descriptor2 specifies the number of bits used to represent the fractional part of the DRA scale parameters signalling and the reconstruction process. The value of dra_descriptor2 shall be in the range of 0 to 15, inclusive. In the current version of the specification value of syntax element dra_descriptor2 is restricted to 9; other values are reserved for future use.

The variable numBitsDraScale is derived as follows:

numBitsDraScale = dra_descriptor1 + dra_descriptor2 (111)

It is requirement of bitstream conformance that the value of numBitsDraScale shall be greater than 0.

dra_number_ranges_minus1 plus 1 specifies the number of ranges signalled to describe the DRA table. The value of dra_number_ranges_minus1 shall be in the range of 0 to 31, inclusive.

dra_equal_ranges_flag equal to 1 specifies that the DRA table is derived using equal-sized ranges, with size specified by the syntax element dra_delta_range[0]. dra_equal_ranges_flag equal to 0 specifies that the DRA table is derived using dra_number_ranges, with the size of each of the ranges specified by the syntax element dra_delta_range[j].

dra_global_offset specifies that the starting codeword position utilized to derive the DRA table, InDraRange[0]. The value of dra_global_offset shall be in the range of 1 to Min(1023, (1 << $BitDepth_Y$) – 1), inclusive.

dra_delta_range[j] specifies the size of the j-th range in codewords which is utilized to derive the DRA table, for j in the range of 0 to dra_number_ranges_minus1, inclusive. The value of dra_delta_range[j] shall be in the range of 1 to Min(1023, (1 << $BitDepth_Y$) – 1), inclusive.

The variable InDraRange[0] is initialized as follows:

InDraRange[0] = dra_global_offset << Max(0, $BitDepth_Y$ – 10) (112)

The variable InDraRange[j] for j in the range of 1 to dra_number_ranges_minus1 + 1, inclusive, are derived as follows:

deltaRange = (dra_equal_ranges_flag = = 1) ? dra_delta_range[0] : dra_delta_range[j – 1] (113)

InDraRange[j] = InDraRange[j – 1] + (deltaRange << Max(0, $BitDepth_Y$ – 10)) (114)

It is a requirement of the bitstream conformance that InDraRange[j] shall be in the range 0 to (1 << $BitDepth_Y$) – 1, inclusive.

dra_scale_value[j] specifies the DRA scale value associated with j-th range of the DRA table. The number of bits used to signal dra_scale_value[j] is equal to numBitsDraScale.

dra_cb_scale_value specifies the scale value for chroma samples of the Cb component utilized to derive the DRA table. The number of bits used to signal dra_cb_scale_value is equal to numBitsDraScale.

dra_cr_scale_value specifies the scale value for chroma samples of the Cr component utilized to derive the DRA table. The number of bits used to signal dra_cr_scale_value is equal to numBitsDraScale.

The values of dra_scale_value[j] for j in the range of 0 to dra_number_ranges_minus1, inclusive, dra_cb_scale_value and dra_cr_scale_value shall not be equal to 0. In the current version of the specification, the value of syntax elements dra_scale_value[j], dra_cb_scale_value and dra_cr_scale_value shall be less than 4 << dra_descriptor2. Other values are reserved for future use. When ChromaArrayType is equal to 0, it is requirements of the bitstream conforming to this document, that values dra_cb_scale_value and dra_cr_scale_value shall be equal to 1.

dra_table_idx specifies the mode the chroma scale derivation process, and when it is applicable, dra_table_idx is less than or equal to 57, the access entry of the ChromaQpTable utilized to derived the chroma scale values. The value of dra_table_idx shall be in the range of 0 to 58, inclusive. When ChromaArrayType is equal to 0, it is requirements of the bitstream conforming to this document, that values dra_table_idx shall be equal to 58.

If dra_table_idx is equal to 58, the variable DraJoinedScaleFlag is set equal to 0, otherwise it is set equal to 1.

The variable numOutRangesL is set equal to dra_number_ranges_minus1 + 1.

The variable OutRangesL[adaptation_parameter_set_id][0] is set equal to 0, the variable OutRangesL[i], for i in the range of 0 to numOutRangesL, inclusive, are derived as follows:

$$outDelta = dra\_scale\_value[i - 1] * (InDraRange[i] - InDraRange[i - 1]) \quad (115)$$

$$OutRangesL[adaptation\_parameter\_set\_id][i] = OutRangesL[adaptation\_parameter\_set\_id][i - 1] + outDelta \quad (116)$$

The variables denoting DRA scale and offset values, InvLumaScales[adaptation_parameter_set_id][i], and DraOffsets[adaptation_parameter_set_id][i] respectively, for i in the range of 1 to dra_number_ranges_minus1, inclusive, are derived as follows:

$$invScalePrec = 18 \quad (117)$$

$$invScale = ((1 << invScalePrec) + (dra\_scale\_val[i] >> 1)) / dra\_scale\_val[i] \quad (118)$$

$$InvLumaScales[adaptation\_parameter\_set\_id][i] = invScale \quad (119)$$

$$diffVal = OutRangesL[i + 1] * invScale \quad (120)$$

$$DraOffsets[adaptation\_parameter\_set\_id][i] = ((InDraRange[i +1] << invScalePrec) - diffVal + (1<< (dra\_descriptor2 - 1))) >> dra\_descriptor2 \quad (121)$$

The variable OutRangesL[i], for i in the range of 0 to numOutRangesL, inclusive, are modified as follows:

$$OutRangesL[adaptation\_parameter\_set\_id][i] = (OutRangesL[adaptation\_parameter\_set\_id][i] + (1 << (dra\_descriptor2 - 1))) >> dra\_descriptor2 \quad (122)$$

When ChromaArrayType is not equal to 0, the process of deriving output chroma DRA parameters in subclause 8.9.7 is invoked with array of luma scale values dra_scale_val[ ], array of luma range values OutRangesL[ ], and chroma index cIdx as inputs, and array of chroma scale values OutScalesC[ ][ ], array of chroma offset values OutOffsetsC[ ][ ] and array of chroma range values OutRangesC[ ][ ] as outputs.

7.4.8 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 subclause 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. When not present, the value of num_ltrp_entries[listIdx][rplsIdx] is inferred to be equal to 0. The value of num_strp_entries[listIdx][rplsIdx] shall be in the range of 0 to sps_max_dec_pic_buffering_minus1, inclusive.

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:

```
NumEntriesInList[ listIdx ][ rplsIdx ] =
num_strp_entries[ listIdx ][ rplsIdx ] +num_ltrp_entries[ listIdx ][ rplsIdx ]          (123)
```

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.

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+ + ) {
    if( !lt_ref_pic_flag[ i ][ RplsIdx[ i ] ][ j ] )                                    (124)
        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.

7.4.9 Slice data semantics

7.4.9.1 General slice data semantics

end_of_tile_one_bit shall be equal to 1.

7.4.9.2 Coding tree unit semantics

The CTU is the root node of the coding structure.

The array IsCoded[x][y], with x = x0..x0 + (1 << CtbLog2SizeY), and y = y0..y0 + (1 << CtbLog2SizeY), specifying whether the sample at (x, y) is coded is initialized as follows:

IsCoded[x][y] = FALSE (125)

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.

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

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

alf_ctb_chroma2_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY] equal to 1 specifies that the adaptive loop filter is applied to the coding tree block of the Cr chroma component of the coding tree unit at location (xCtb, yCtb). alf_ctb_chroma2_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY] equal to 0 specifies that the adaptive loop filter is not applied to the coding tree block of the Cr chroma component of the coding tree unit at location (xCtb, yCtb).

When alf_ctb_chroma2_flag[xCtb >> CtbLog2SizeY][yCtb >> CtbLog2SizeY] is not present, it is inferred to be equal to sliceChroma2AlfEnabledFlag.

7.4.9.3 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:

— 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 2.
  - 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.
  - If predModeConstraintCurrent is equal to PRED_MODE_CONSTRAINT_INTER and log2CbWidthTemp is equal to 2 and log2CbHeight is equal to 2.
- 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 2.
  - 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.
  - If predModeConstraintCurrent is equal to PRED_MODE_CONSTRAINT_INTER and log2CbWidth is equal to 2 and log2CbHeightTemp is equal to 2.
- Otherwise, allowSplitBtHor is set equal to TRUE.

— The variable allowSplitTtVer is derived as follows:

- If one or more of the following conditions are true, allowSplitTtVer is set equal to FALSE.

- log2CbWidth is less than log2CbHeight.
- log2CbWidth is greater than MaxTtLog2Size or log2CbWidth is less than MinTtLog2Size.
- log2CbWidth is equal to log2CbHeight and log2CbWidth is larger than MaxCbLog2Size14Ratio.
- log2CbWidth is equal to log2CbHeight and log2CbWidth is smaller than MinCbLog2Size14Ratio.
- If predModeConstraintCurrent is equal to PRED_MODE_CONSTRAINT_INTER and log2CbWidth is equal to 4 and log2CbHeight is equal to 2.

— Otherwise, allowSplitTtVer is set equal to TRUE.

— The variable allowSplitTtHor is derived as follows:

— If one or more of the following conditions are true, allowSplitTtHor is set equal to FALSE.

- log2CbHeight is less than log2CbWidth.
- log2CbHeight is greater than MaxTtLog2Size or log2CbHeight is less than MinTtLog2Size.
- log2CbWidth is equal to log2CbHeight and log2CbHeight is larger than MaxCbLog2Size14Ratio.
- log2CbWidth is equal to log2CbHeight and log2CbHeight is smaller than MinCbLog2Size14Ratio.
- If predModeConstraintCurrent is equal to PRED_MODE_CONSTRAINT_INTER and log2CbWidth is equal to 2 and log2CbHeight is equal to 4.

— 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 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.

— Otherwise, btt_split_type[x0][y0] is set equal to 0.

The array SplitMode[x][y] is derived as follows for x = x0..x0 + (1 << log2CbWidth) − 1 and y = y0..y0 + (1 << log2CbHeight) − 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 smaller than or equal to pic_height_in_luma_samples, the following applies:

— If allowSplitBtVer is equal to TRUE, SplitMode[x][y] is set equal to SPLIT_BT_VER.

— Otherwise, SplitMode[x][y] is set equal to SPLIT_BT_HOR.

— Otherwise, if y0 + (1 << log2CbHeight) is greater than pic_height_in_luma_samples, the following applies:

— If allowSplitBtHor is equal to TRUE, SplitMode[x][y] is set equal to SPLIT_BT_HOR.

— Otherwise, SplitMode[x][y] is set equal to SPLIT_BT_VER.

— Otherwise, 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 + (1 << log2CbWidth) − 1 and y = y0..y0 + (1 << log2CbHeight) − 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.

- log2CbSizeLongerSide 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_cu_flag[x0][y0] is equal to 0.
- SplitMode[x0][y0] is equal to SPLIT_BT_HOR, SPLIT_TT_HOR or NO_SPLIT, and split_cu_flag[x0][y0] is equal to 0.

— Otherwise, allowSplitUnitCodingOrder is set equal to TRUE.

The variable needSignalPredModeConstraintTypeFlag is derived as follows:

— If one of the following conditions is true, needSignalPredModeConstraintTypeFlag is set equal to 0:
  - sps_btt_flag is equal to 0 or sps_admvp_flag is equal to 0.
  - slice_type is equal to I.
  - ChromaArrayType is not equal to 1.
  - predModeConstraintCurrent is not equal to PRED_MODE_NO_CONSTRAINT.
— Otherwise, if one of the following conditions is true, needSignalPredModeConstraintTypeFlag is set equal to 1:
  - cbWidth * cbHeight is equal to 64 and SplitMode[x0][y0] is equal to SPLIT_BT_HOR or SPLIT_BT_VER.
  - cbWidth * cbHeight is equal to 128 and SplitMode[x0][y0] is equal to SPLIT_TT_HOR or SPLIT_TT_VER.
— Otherwise, needSignalPredModeConstraintTypeFlag is set equal to 0.

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.

pred_mode_constraint_type_flag[x0][y0] specifies type of restriction on prediction mode for all coding units inside the current split unit. When pred_mode_constraint_type_flag[x0][y0] is not present, it is inferred to be equal to 0.

The variable predModeConstraint is derived as follows:

— If sps_btt_flag is equal to 0 or sps_admvp_flag is equal to 0, predModeConstraint is set equal to PRED_MODE_NO_CONSTRAINT.

— Otherwise, if needSignalPredModeConstraintTypeFlag is equal to 1, the variable predModeConstraint is derived as follows:

predModeConstraint = pred_mode_constraint_type_flag[x0][y0] ?
PRED_MODE_CONSTRAINT_INTRA_IBC : PRED_MODE_CONSTRAINT_INTER (126)

— Otherwise, if predModeConstraintCurrent is not equal to PRED_MODE_NO_CONSTRAINT the variable predModeConstraint is set equal to predModeConstraintCurrent.

— Otherwise, if one of the following conditions is true, predModeConstraint is set equal to PRED_MODE_CONSTRAINT_INTRA_IBC:

  — ChromaArrayType is equal to 1 and cbWidth * cbHeight is equal to 64 and SplitMode[x0][y0] is not equal to NO_SPLIT.

  — ChromaArrayType is equal to 1 and cbWidth * cbHeight is equal to 128 and SplitMode[x0][y0] is equal to SPLIT_TT_HOR or SPLIT_TT_VER.

  — ChromaArrayType is equal to 2 and cbWidth * cbHeight is equal to 32 and SplitMode[x0][y0] is equal to SPLIT_BT_HOR or SPLIT_BT_VER.

  — ChromaArrayType is equal to 2 and cbWidth * cbHeight is equal to 64 and SplitMode[x0][y0] is equal to SPLIT_TT_HOR or SPLIT_TT_VER.

— Otherwise, predModeConstraint is set equal to PRED_MODE_NO_CONSTRAINT.

The variable isTreeSplitPoint is set equal to 1 if the following two conditions both are true, and is set equal to 0 otherwise:

— predModeConstraintCurrent is equal to PRED_MODE_NO_CONSTRAINT.

— predModeConstraint is equal to PRED_MODE_CONSTRAINT_INTRA_IBC.

7.4.9.4 Coding unit semantics

The luma coding block width cbWidth and height cbHeight are derived as follows:

cbWidth = (1 << log2CbWidth) (127)

cbHeight = (1 << log2CbHeight) (128)

The arrays CbPosX and CbPosY specify horizontal and vertical coordinates of top-left sample of luma block corresponding to considered coding unit. The arrays CbWidth and CbHeight specify width and height of luma block corresponding to considered coding unit. The array indices x and y specify the location (x, y) relative to the top-left luma sample of the picture.

The arrays CbPosX, CbPosY, CbWidth, and CbHeight are derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

CbPosX[x][y] = x0 (129)

CbPosY[x][y] = y0 (130)

CbWidth[x][y] = cbWidth (131)

CbHeight[x][y] = cbHeight (132)

The array LumaPredMode[x][y] specify prediction mode of luma block covers location (x, y). The array indices x, y specify the location (x, y) relative to the top-left luma sample of the picture. LumaPredMode[x][y] can be one of followings: MODE_INTRA, MODE_INTER, and MODE_IBC.

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.

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 inter prediction direction (list 0, list 1, or bi-prediction direction) of the merging candidate determined by mmvd_merge_idx[x0][y0] is used at MMVD mode, i.e. specifies whether the merging candidate is modified. 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 9. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When mmvd_distance_idx[x0][y0] is not present, it is inferred to be equal to 0.

Table 9 — 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 the index used to derive MmvdSign[x0][y0] as specified in Table 10. 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 10 — 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:

$$MmvdOffset[x0][y0][0] = MmvdDistance[x0][y0] * MmvdSign[x0][y0][0] \quad (133)$$

$$MmvdOffset[x0][y0][1] = MmvdDistance[x0][y0] * MmvdSign[x0][y0][1] \quad (134)$$

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. affine_flag[x0][y0] equal to 0 specifies that the coding unit is not predicted by affine model based motion compensation. When affine_flag[x0][y0] is not present, it is inferred to be equal to 0.

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

merge_idx[x0][y0] specifies the merging candidate index of the merging candidate list where x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When merge_idx[x0][y0] is not present, it is inferred to be equal to 0.

pred_mode_flag[x0][y0] when present specifies that the current coding unit is coded in inter or intra block copy prediction mode if pred_mode_flag[x0][y0] is equal to 0. pred_mode_flag[x0][y0] is equal to 1 specifies that the current coding unit is coded in intra prediction mode.

The variable isIbcAllowed is set equal to 1 if all of the following conditions are met, and is set equal to 0 otherwise:

— sps_ibc_flag is equal to 1.

— log2CbWidth is less than or equal to log2MaxIbcCandSize and log2CbHeight is less than or equal to log2MaxIbcCandSize.

— treeType is not equal to DUAL_TREE_CHROMA.

— predModeConstraint is not equal to PRED_MODE_CONSTRAINT_INTER.

— predModeConstraint is not equal to PRED_MODE_NO_CONSTRAINT or pred_mode_flag[x0][y0] is equal to 1.

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 is derived as follows:

— If ibc_flag[x0][y0] is equal to 1, the variable CuPredMode is set equal to MODE_IBC.

— Otherwise, if predModeConstraint is equal to PRED_MODE_CONSTRAINT_INTER or cu_skip_flag[x0][y0] is equal to 1, CuPredMode is set equal to MODE_INTER.

— Otherwise, if predModeConstraint is equal to PRED_MODE_CONSTRAINT_INTRA, CuPredMode is set equal to MODE_INTRA.

— Otherwise, CuPredMode is derived as follows:

CuPredMode = pred_mode_flag[x0][y0] = = 0 ? MODE_INTER : MODE_INTRA (135)

If treeType is not equal to DUAL_TREE_CHROMA, LumaPredMode[x][y] is set equal to CuPredMode for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbWidth − 1.

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

affine_mode_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. affine_mode_flag[x0][y0] equal to 0 specifies that 4-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit.

MotionModelIdc[x][y] represents motion model of a coding unit as illustrated in Table 11. The array indices x, y specify the luma sample location (x, y) relative to the top-left luma sample of the picture.

The variable MotionModelIdc[x][y] is derived as follows for x = x0..x0 + (1 << log2CbWidth) − 1 and y = y0..y0 + (1 << log2CbHeight) − 1:

— If merge_mode_flag[x0][y0] is equal to 1, the following applies:

MotionModelIdc[x][y] = affine_flag[x0][y0] (136)

— Otherwise (merge_mode_flag[x0][y0] is equal to 0), the following applies:

MotionModelIdc[x][y] = affine_flag[x0][y0] + affine_mode_flag[x0][y0] (137)

Table 11 — Interpretation of MotionModelIdc[x0][y0]

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

affine_mvp_flag_l0[x0][y0] specifies the affine motion vector predictor flag 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 affine_mvp_flag_l0[x0][y0] is not present, it is inferred to be equal to 0.

affine_mvp_flag_l1[x0][y0] has the same semantics as affine_mvp_flag_l0, with l0 and list 0 replaced by l1 and list 1, respectively.

affine_mvd_flag_l0[x0][y0] equal to 1 indicates that the motion vector difference of list 0 of current affine coding uint is presented in the bitstream, 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 affine_mvd_flag_l0[x0][y0] is not present, it is inferred to be equal to 0.

When affine_mvd_flag_l0 is equal to 0, the following applies:

MvdCpL0[x0][y0][0][0] = 0 (138)

MvdCpL0[x0][y0][0][1] = 0 (139)

MvdCpL0[x0][y0][1][0] = 0 (140)

MvdCpL0[x0][y0][1][1] = 0 (141)

MvdCpL0[x0][y0][2][0] = 0 (142)

MvdCpL0[x0][y0][2][1] = 0 (143)

affine_mvd_flag_l1[x0][y0] has the same semantics as affine_mvd_flag_l0, with l0, L0 and list 0 replaced by l1, L1 and list 1, respectively.

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.

direct_mode_flag[x0][y0] specifies whether the inter prediction parameters for the current coding unit are inferred from the temporal collocated block. 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.

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.

inter_pred_idc[x0][y0] specifies whether list 0, list 1, or bi-prediction is used for the current coding unit according to Table 12. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When inter_pred_idc[x0][y0] is not present, it is inferred to be equal to PRED_L0.

Table 12 — Name association to inter prediction mode

inter_pred_idc	Name of inter_pred_idc	
	((1 << log2CbWidth) + (1 << log2CbHeight)) > 12 || !sps_admvp_flag	((1 << log2CbWidth) + (1 << log2CbHeight)) <= 12 && sps_admvp_flag
0	PRED_BI	PRED_L0
1	PRED_L0	PRED_L1
2	PRED_L1	na

ref_idx_l0[x0][y0] specifies the list 0 reference picture index for the current coding unit. The array indices x0 and 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.

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

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.

mvd_l0_sign_flag[x0][y0][vertex][compIdx] has the same semantics as mvd_l0_sign_flag[x0][y0][compIdx].

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 with l0 and list 0 replaced by l1 and list 1, respectively.

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

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.

mvd_l1_sign_flag[x0][y0][vertex][compIdx] has the same semantics as mvd_l1_sign_flag[x0][y0][compIdx].

If MotionModelIdc[x0][y0] is equal to 0, the following applies:

— The variable motion vector difference of list X MvdLX[x0][y0][compIdx], with X being replaced by 0 or 1, specifies the difference between a list X motion vector component and its prediction, is derived as follows for compIdx = 0..1:

$$MvdLX[x0][y0][compIdx] = abs\_mvd\_lX[x0][y0][compIdx] * (1 - 2 * mvd\_lX\_sign\_flag[x0][y0][compIdx]) \quad (144)$$

The value of MvdLX[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 MvdLX[x0][y0][compIdx], with X being replaced by 0 or 1, for compIdx = 0..1 are modified as follows:

$$MvdLX[x0][y0][compIdx] = MvdLX[x0][y0][compIdx] << amvr\_idx[x0][y0] \quad (145)$$

Otherwise (MotionModelIdc[x0][y0] is not equal to 0), the following applies:

— The variable motion vector difference of list X MvdCpLX[x0][y0][vertex][compIdx], with X being replaced by 0 or 1, specifies the difference between a list X motion vector component and its prediction, is derived as follows for vertex = 0..vertexNum – 1 and compIdx = 0..1:

$$MvdCpLX[x0][y0][vertex][compIdx] = abs\_mvd\_lX[x0][y0][vertex][compIdx] * (1 - 2 * mvd\_lX\_sign\_flag[x0][y0][vertex][compIdx]) \quad (146)$$

The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, the array index vertex specifies the control point index. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

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[x0][y0] equal to 1 specifies that the transform_unit( ) syntax structure is present for the current coding unit. cbf_all[x0][y0] equal to 0 specifies that transform_unit( ) syntax structure is not present for the current coding unit. The array indices x0 and 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 cbf_all[x0][y0] is not present, its value is inferred to be equal to 1.

7.4.9.5 Transform unit semantics

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

— The variable allowAtsInterVerHalf is derived as follows:

  — If all of the following conditions are true, the variable allowAtsInterVerHalf is set equal to TRUE.

    — log2CbWidth is greater than or equal to (MinTbLog2SizeY + 1).

    — log2CbWidth is less than or equal to MaxTbLog2SizeY.

    — log2CbHeight is less than or equal to MaxTbLog2SizeY.

  — Otherwise, the variable allowAtsInterVerHalf is set equal to FALSE.

— The variable allowAtsInterVerQuad is derived as follows:

  — If all of the following conditions are true, the variable allowAtsInterVerQuad is set equal to TRUE.

    — log2CbWidth is greater than or equal to (MinTbLog2SizeY + 2).

    — log2CbWidth is less than or equal to MaxTbLog2SizeY.

    — log2CbHeight is less than or equal to MaxTbLog2SizeY.

  — Otherwise, the variable allowAtsInterVerQuad is set equal to FALSE.

— The variable allowAtsInterHorHalf is derived as follows:

  — If all of the following conditions are true, the variable allowAtsInterHorHalf is set equal to TRUE.

    — log2CbHeight is greater than or equal to (MinTbLog2SizeY + 1).

    — log2CbWidth is less than or equal to MaxTbLog2SizeY.

    — log2CbHeight is less than or equal to MaxTbLog2SizeY.

  — Otherwise, the variable allowAtsInterHorHalf is set equal to FALSE.

— The variable allowAtsInterHorQuad is derived as follows:

  — If all of the following conditions are true, the variable allowAtsInterHorQuad is set equal to TRUE.

    — log2CbHeight is greater than or equal to (MinTbLog2SizeY + 2).

    — log2CbWidth is less than or equal to MaxTbLog2SizeY.

    — log2CbHeight is less than or equal to MaxTbLog2SizeY.

  — Otherwise, the variable 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 0.

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 0.

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 0 if treeType is equal to DUAL_TREE_CHROMA, otherwise it is inferred to be equal to 1.

cu_qp_delta_abs specifies the absolute value of the difference CuQpDelta[x][y] value between the luma quantization parameter of the current coding unit and its prediction. When cu_qp_delta_abs is not present, it is inferred to be equal to 0.

When cu_qp_delta_abs is present, the variable isCuQpDeltaCoded is derived as follows:

$$isCuQpDeltaCoded = 1 \tag{147}$$

cu_qp_delta_sign_flag specifies the sign of CuQpDelta[x][y] as follows:

— If cu_qp_delta_sign_flag is equal to 0, the corresponding cu_qp_delta value has a positive value.

— Otherwise (cu_qp_delta_sign_flag is equal to 1), the corresponding cu_qp_delta value has a negative value.

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

CuQpDelta[x][y] represents the value of the difference between the luma quantization parameter of the current coding unit and its prediction. The array indices x, y specify the luma sample location (x, y) relative to the top-left luma sample of the picture.

When cu_qp_delta_abs is present or isSplit is equal to FALSE, the variable CuQpDelta[x][y] is derived as follows for x = x0..x0 + (1 << log2CbWidth) − 1 and y = y0..y0 + (1 << log2CbHeight) − 1:

$$CuQpDelta[x][y] = cu\_qp\_delta\_abs * (1 - 2 * cu\_qp\_delta\_sign) \tag{148}$$

The value of CuQpDelta[x][y] shall be in the range of −26 to +25, inclusive.

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.

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.

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 width of the transform unit is equal to the width of the current coding unit and the height of the transform unit is smaller than the height of the current coding unit. ats_cu_inter_horizontal_flag[x0][y0] equal to 0 specifies that the height of the transform unit is equal to the height of the current coding unit and the width of the transform unit is smaller than the width of the current coding unit.

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 inferred to be equal to allowAtsInterHorQuad.

— Otherwise (ats_cu_inter_quad_flag[x0][y0] is equal to 0), ats_cu_inter_horizontal_flag[x0][y0] is inferred to be equal to allowAtsInterHorHalf.

ats_cu_inter_pos_flag[x0][y0] equal to 1 specifies that the bottom-right corner of the transform unit is aligned with the bottom-right corner of the coding unit. ats_cu_inter_pos_flag[x0][y0] equal to 0 specifies that the top-left corner of the transform unit is aligned with the top-left corner of the coding unit.

7.4.9.6 Run-length residual coding semantics

The transform coefficient levels are represented by the arrays TransCoeffLevel[x0][y0][cIdx][xC][yC], which are either specified in subclause 7.3.8.7 or inferred as follows. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for Y, 1 for Cb, and 2 for Cr. The array indices xC and yC specify the transform coefficient location (xC, yC) within the current transform block. When the value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is not specified in subclause 7.3.8.7, it is inferred to be equal to 0.

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.

The value of coeff_abs_level_minus1 shall be constrained such that the corresponding value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is in the range of −32768 to 32767, inclusive.

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 (1 << log2CbWidth) * (1 << log2CbHeight) – 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.

7.4.9.7 Advanced residual coding semantics

The transform coefficient levels are represented by the arrays TransCoeffLevel[x0][y0][cIdx][xC][yC], which are either specified in subclause 7.3.8.8 or inferred as follows. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for Y, 1 for Cb, and 2 for Cr. The array indices xC and yC specify the transform coefficient location (xC, yC) within the current transform block. When the value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is not specified in subclause 7.3.8.8, it is inferred to be equal to 0.

last_sig_coeff_x_prefix specifies the prefix of the column position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_x_prefix shall be in the range of 0 to (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:

$$LastSignificantCoeffX = last\_sig\_coeff\_x\_prefix \tag{149}$$

— Otherwise (last_sig_coeff_x_suffix is present), the following applies:

$$LastSignificantCoeffX = (1 << ((last\_sig\_coeff\_x\_prefix >> 1) - 1)) * (2 + (last\_sig\_coeff\_x\_prefix \& 1)) + last\_sig\_coeff\_x\_suffix \tag{150}$$

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:

$$LastSignificantCoeffY = last\_sig\_coeff\_y\_prefix \quad (151)$$

— Otherwise (last_sig_coeff_y_suffix is present), the following applies:

$$LastSignificantCoeffY = (1 << ((last\_sig\_coeff\_y\_prefix >> 1) - 1)) * (2 + (last\_sig\_coeff\_y\_prefix \& 1)) + last\_sig\_coeff\_y\_suffix \quad (152)$$

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 numNZ present in the transform coefficients group and should be in the range $1..2^{16} - 1$, 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.

The value of coeff_abs_level_remaining[n] shall be constrained such that the corresponding value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is in the range of −32768 to 32767, inclusive.

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 subclause 8.2.

2) The processes in subclause 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 subclause 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 subclause 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 subclause 8.3.4 is invoked for derivation of the variables ColPic.

   — The decoding process for reference picture marking in subclause 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 subclauses 8.4, 8.5, 8.6, 8.7 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                    (153)
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:

$$PicOrderCntVal = PicOrderCntMsb + slice\_pic\_order\_cnt\_lsb \quad (154)$$

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

— If the current picture is an IDR picture, the following applies:

$$PicOrderCntVal = 0 \quad (155)$$

$$DocOffset = -1 \quad (156)$$

— Otherwise (the current picture is not an IDR picture), the following ordered steps are applied:

1) Let prevPicOrderCntVal be the PicOrderCntVal of prevTid0Pic.

2) If TemporalId equals 0, then PicOrderCntVal and DocOffset are derived as follows:

$$PicOrderCntVal = prevPicOrderCntVal + SubGopLength \quad (157)$$

$$DocOffset = 0 \quad (158)$$

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:

$$DocOffset = (prevDocOffset + 1) \% SubGopLength \quad (159)$$

— If DocOffset equals 0, the following applies:

prevPicOrderCntVal = prevPicOrderCntVal + SubGopLength (160)

ExpectedTemporalId = 0 (161)

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

ExpectedTemporalId = 1 + Floor(Log2(DocOffset)) (162)

— PicOrderCntVal is derived as follows:

```
while( TemporalId != ExpectedTemporalId ) {
    DocOffset = ( DocOffset + 1 ) % SubGopLength
    if( DocOffset = = 0 )
        ExpectedTemporalId = 0
    else                                                              (163)
        ExpectedTemporalId = 1 + Floor( Log2( DocOffset ) )
}
PocOffset = SubGopLength * ( ( 2 * DocOffset + 1 ) / 2^ExpectedTemporalId − 2 )
PicOrderCntVal = prevPicOrderCntVal + PocOffset
```

NOTE All IDR pictures 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 (164)

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

DiffPicOrderCnt(picA, picB) = PicOrderCnt(picA) − PicOrderCnt(picB) (165)

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 Where X is the current picture and Y and Z are 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 subclause 8.3.3 or in decoding of the slice data.

NOTE For an I slice of a non-IDR picture that it is not the first slice of the picture, RefPicList[0] and RefPicList[1] can 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] can 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 ]                                          (166)
        } 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 1 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 2 The active entries in RefPicList[0] and the active entries in RefPicList[1] collectively refer to all reference pictures that can 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 can be used in inter prediction for one or more pictures that follow the current picture in decoding order.

NOTE 3 There can 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[1] that is equal to "no reference picture" must be ignored. An unintentional picture loss must 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 subclause 8.3.2.2.2 is invoked with i set equal to 0 and startIdx set equal 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 subclause 8.3.2.2.3 is invoked with i set equal to 0 and startIdx set equal 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 subclause 8.3.2.2.3 is invoked with i set equal to 1 and startIdx set equal 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 subclause 8.3.2.2.2 is invoked with i set equal to 1 and startIdx set equal 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

Inputs to this process are:

— a reference picture list identifier i, and

— a start index position startIdx.

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

The variable nextIdx is set equal to startIdx.

The variable 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                                    (167)
            nextTemporalId = Max( TemporalId of picA − 1, 0 )
    }
}
```

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

Inputs to this process are:

— a reference picture list identifier i, and

— a start index position startIdx.

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

The variable nextIdx is set equal to startIdx.

The variable 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                                    (168)
            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 subclause 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_len 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 subclause 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                                                              (169)
            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++                                                                    (170)
        } 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 subclause 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

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, and

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

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

When treeType is not equal to DUAL_TREE_CHROMA, the following steps are performed:

— The derivation process for the luma intra prediction mode as specified in subclause 8.4.2 is invoked with the luma location (xCb, yCb) and the width of the current coding block log2BlkWidth set equal to log2CbWidth as inputs.

— The decoding process for intra prediction as specified in subclause 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 outputs are the predicted samples predSamples[x][y], with x = 0..(1 << log2CbWidth) − 1 and y = 0..(1 << log2CbHeight) − 1.

— Let resSamples be (1 << log2CbWidth)x(1 << log2CbHeight) array of residual samples.

— The decoding process for the residual signal as specified in subclause 8.4.5 is invoked with the luma location (xCb, yCb), the luma location (0, 0), the variable nCbW set equal to (1 << log2CbWidth), the variable nCbH set equal to (1 << log2CbHeight), the variable nTbW set equal to (1 << log2CbWidth), the variable nTbH set equal to (1 << log2CbHeight), the variable cIdx set equal to 0 and the (1 << log2CbWidth)x(1 << log2CbHeight) array resSamples as inputs, the output is a modified version of the (1 << log2CbWidth)x(1 << log2CbHeight) array resSamples.

— The picture construction process prior to post-reconstruction and in-loop filtering for a colour component as specified in subclause 8.7.5 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 << log2CbHeight), the variable cIdx set equal to 0, the (1 << log2CbWidth)x(1 << log2CbHeight) array predSamples set equal to predSamples, and the (1 << log2CbWidth)x(1 << log2CbHeight) array resamples set equal to resSamples as inputs, and the output is a modified reconstructed picture before the post-reconstruction filtering process and in-loop filtering.

— When the value of sps_htdf_flag is equal to 1, the post-reconstruction filter process prior to in-loop filtering for a luma component as specified in subclause 8.7.6.1 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 << log2CbHeight), the variable Qp_Y set equal to Qp'_Y, which is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1, and the output is a modified reconstructed picture before in-loop filtering.

When ChromaArrayType is not equal to 0 and treeType is not equal to DUAL_TREE_LUMA, 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 = = 1 ? 1 : 0).

— The derivation process for the chroma intra prediction mode as specified in subclause 8.4.3 is invoked with the luma location (xCb, yCb) as input, and the output is the variable IntraPredModeC.

— The decoding process for intra prediction as specified in subclause 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 the predicted samples predSamples[x][y], with x = 0..(1 << log2CbWidthC) − 1 and y = 0..(1 << log2CbHeightC) − 1.

— Let resSamples be (1 << log2CbWidthC)x(1 << log2CbHeightC) array of residual samples.

— The decoding process for the residual signal as specified in subclause 8.4.5 is invoked with the luma location (xCb, yCb), the luma location (0, 0), the variable nCbW set equal to (1 << log2CbWidth), the variable nCbH set equal to (1 << log2CbHeight), the variable nTbW set equal to (1 << log2CbWidth), the variable nTbH set equal to (1 << log2CbHeight), the variable cIdx set equal to 1, and the (1 << log2CbWidthC)x(1 << log2CbHeightC) array resSamples as inputs, the output is a modified version of the (1 << log2CbWidthC)x(1 << log2CbHeightC) array resSamples.

— The picture construction process prior to in-loop filtering for a colour component as specified in subclause 8.7.5 is invoked with the luma coding block location (xCb, yCb), the variable nCbW set equal to (1 << log2CbWidthC), the variable nCbH set equal to (1 << log2CbHeightC), the variable cIdx set equal to 1, the (1 << log2CbWidthC)x(1 << log2CbHeightC) array predSamples set equal to predSamples, and the (1 << log2CbWidthC)x(1 << log2CbHeightC) array resamples set equal to resSamples as inputs, and the output is a modified reconstructed picture before in-loop filtering.

— The decoding process for intra prediction as specified in subclause 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 the predicted samples predSamples[x][y], with x = 0..(1 << log2CbWidthC) − 1 and y = 0..(1 << log2CbHeightC) − 1.

— The decoding process for the residual signal as specified in subclause 8.4.5 is invoked with the luma location (xCb, yCb), the luma location (0, 0), the variable nCbW set equal to (1 << log2CbWidth), the variable nCbH set equal to (1 << log2CbHeight), the variable nTbW set equal to (1 << log2CbWidth), the variable nTbH set equal to (1 << log2CbHeight), the variable cIdx set equal to 2, and the (1 << log2CbWidthC)x(1 << log2CbHeightC) array resSamples as inputs, the output is a modified version of the (1 << log2CbWidthC)x(1 << log2CbHeightC) array resSamples.

— The picture construction process prior to in-loop filtering for a colour component as specified in subclause 8.7.5 is invoked with the luma coding block location (xCb, yCb), the variable nCbW set equal to (1 << log2CbWidthC), the variable nCbH set equal to (1 << log2CbHeightC), the variable cIdx set equal to 2, the (1 << log2CbWidthC)x(1 << log2CbHeightC) array predSamples set equal to predSamples, and the (1 << log2CbWidthC)x(1 << log2CbHeightC) array resamples set equal to resSamples 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 to this process are:

— a sample location (xCb, yCb) specifying the top-left sample of the current block relative to the top-left sample of the current picture, and

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

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

If sps_eipd_flag is equal to 0, the following applies:

— Table 13 specifies the value for the intra prediction mode and the associated names.

Table 13 — Specification of intra prediction mode and associated names (when sps_eipd_flag = = 0)

Intra prediction mode (IntraPredModeY[xCb][yCb])	**Associated name**
0	INTRA_DC
1	INTRA_HOR
2	INTRA_VER
3	INTRA_UL
4	INTRA_UR

— IntraPredModeList specified in Table 14 is derived by the intra prediction modes of neighbouring blocks, and IntraPredModeY[xCb][yCb] is derived using IntraPredModeList as follows:

IntraPredModeY[xCb][yCb] = IntraPredModeList[intra_pred_mode[xCb][yCb]] (171)

Table 14 — Derivation of IntraPredModeList from the intra prediction mode of the neighbouring blocks

		IntraPredModeY[xCb][yCb − 1]					
		na	**0**	**1**	**2**	**3**	**4**
IntraPredModeY [xCb − 1][yCb]	**na**	0, 2, 3, 1, 4	0, 2, 1, 3, 4	0, 2, 1, 3, 4	1, 2, 0, 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	1, 2, 0, 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	2, 0, 1, 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	1, 2, 0, 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	1, 2, 0, 3, 4	1, 2, 3, 0, 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 (sps_eipd_flag is equal to 1), the following applies:

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

1) The neighbouring locations (xNbA, yNbA), (xNbB, yNbB) and (xNbC, yNbC) are set equal to (xCb − 1, yCb), (xCb, yCb − 1) and (xCb + (1 << log2BlkWidth), yCb), respectively.

2) For X being replaced by either A, B or C, the variables candIntraPredModeX are derived as follows:

   — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked the neighbouring location (xNbY, yNbY) set equal to (xNbX, yNbX) as input, and the output is assigned to availableX.

   — A variable validX and the candidate intra prediction mode candIntraPredModeX are derived as follows:

      — If the availableX is equal to TRUE and LumaPredMode[xNbX][yNbX] is equal to MODE_INTRA, validX is set equal to TRUE and candIntraPredModeX is set equal to IntraPredModeY[xNbX][yNbX].

      — Otherwise, validX is set equal to FALSE and candIntraPredModeX is set equal to INTRA_DC.

   — When validC is equal to TRUE, the following applies:

      — If both validA and validB are equal to TRUE, the following applies:

         — If candIntraPredModeA is equal to candIntraPredModeB, candIntraPredModeB is set equal to candIntraPredModeC and validC is set equal to FALSE.

         — Otherwise, if candIntraPredModeA or candIntraPredModeB is equal to candIntraPredModeC, validC is set equal to FALSE.

      — Otherwise, if validA is equal to FALSE, candIntraPredModeA is set equal to candIntraPredModeC and validC is set equal to FALSE.

      — Otherwise, if validB is equal to FALSE, candIntraPredModeB is set equal to candIntraPredModeC and validC is set equal to FALSE.

3) candModeList[x] with x = 0, 1 is derived as follows:

candModeList[0] = Min(candIntraPredModeA, candIntraPredModeB) (172)

candModeList[1] = Max(candIntraPredModeA, candIntraPredModeB) (173)

— When candModeList[1] is equal to candModeList[0], the following applies:

candModeList[0] = INTRA_DC (174)

candModeList[1] = (candModeList[1] = = INTRA_DC) ? INTRA_BI : candModeList[1] (175)

4) extCandModeList[x] with x = 0..7 is derived as follows:

— If validC 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:

— extCandModeList[0] is a mode which is not included in candModeList[x] with x = 0, 1 among INTRA_PLN, INTRA_DC, and INTRA_BI. extCandModeList[x] with x = 1..7 is derived as follows:

extCandModeList[1] = INTRA_VER (176)

extCandModeList[2] = INTRA_HOR (177)

extCandModeList[3] = INTRA_DIA_R (i.e., (INTRA_VER + INTRA_HOR) >> 1) (178)

extCandModeList[4] = INTRA_DIA_L (179)

extCandModeList[5] = INTRA_DIA_U (180)

extCandModeList[6] = INTRA_VER + 4 (181)

extCandModeList[7] = INTRA_HOR − 4 (182)

— Otherwise, if candModeList[0] is less than 3 and candModeList[1] are greater than or equal to 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 (183)

extCandModeList[1] = INTRA_DC (184)

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

extCandModeList[0] = (candModeList[0] = = INTRA_BI) ? INTRA_DC : INTRA_BI (185)

extCandModeList[1] = INTRA_PLN (186)

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

extCandModeList[2] = (candModeList[1] = = 32) ? 31 : 32 (187)

extCandModeList[3] = 30 (188)

extCandModeList[4] = 29 (189)

extCandModeList[5] = 28 (190)

extCandModeList[6] = INTRA_HOR (191)

extCandModeList[7] = INTRA_DIA_R (192)

— 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 (193)

extCandModeList[3] = 5 (194)

extCandModeList[4] = 6 (195)

extCandModeList[5] = 7 (196)

extCandModeList[6] = INTRA_VER (197)

extCandModeList[7] = INTRA_DIA_R (198)

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

extCandModeList[2] = candModeList[1] + 2 (199)

extCandModeList[3] = candModeList[1] − 2 (200)

extCandModeList[4] = candModeList[1] + 1 (201)

extCandModeList[5] = candModeList[1] − 1 (202)

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

extCandModeList[6] = candModeList[1] − 5 (203)

extCandModeList[7] = candModeList[1] + 5 (204)

— Otherwise, the following applies:

extCandModeList[6] = (candModeList[1] > 23) ?
candModeList[1] − 5 : candModeList[1] + 5 (205)

extCandModeList[7] = (candModeList[1] > 23) ?
candModeList[1] − 10 : candModeList[1] + 10 (206)

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

extCandModeList[0] = INTRA_BI (207)

extCandModeList[1] = INTRA_DC (208)

— extCandModeList[x] with x = 2..7 is derived as follows:

— 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 (209)

list[1] = (candModeList[0] = = 31) ?
candModeList[0] − 1 : candModeList[0] + 2 (210)

list[2] = (candModeList[1] = = 4) ?
candModeList[1] + 1 : candModeList[1] − 2 (211)

list[3] = (candModeList[1] = = 32 | | candModeList[1] = = 31) ?
candModeList[1] − 1 : candModeList[1] + 2 (212)

list[4] = (candModeList[0] + candModeList[1] + 1) >> 1 (213)

list[5] = (list[4] + candModeList[0] + 1) >> 1 (214)

list[6] = (list[4] + candModeList[1] + 1) >> 1 (215)

list[7] = INTRA_VER (216)

list[8] = INTRA_HOR (217)

list[9] = INTRA_DIA_R (218)

list[10] = INTRA_PLN (219)

list[11] = INTRA_DIA_L (220)

list[12] = INTRA_DIA_U (221)

list[13] = INTRA_VER + 4 (222)

list[14] = INTRA_HOR − 4 (223)

— A variable iCount is set equal to 2.

— For i equal to 0 to 14, inclusive, the following is applied until iCount is greater than 7:

— For j equal to 0 to iCount − 1, inclusive, the following is applied:

— When list[i] is equal to among extCandModeList[j], candModeList[0], and candModeList[1], i is incremented by 1 and j is set equal to 0.

— When j is equal to (iCount − 1), extCandModeList[iCount] is set equal to list[i], i and iCount are incremented by 1 and j is set equal to 0.

— Otherwise (validC is equal to TRUE), 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:

  — extCandModeList[0] is set equal to a mode which is not included in the candModeList among INTRA_PLN, INTRA_DC, and INTRA_BI.

  — If candIntraPredModeC is less than 3, extCandModeList[x] with x = 1..7 is set equal to Formulae 176 to 182.

  — Otherwise, extCandModeList[x] with x = 1..7 is derived as follows:

$$extCandModeList[\ 1\ ] = candIntraPredModeC \tag{224}$$

$$extCandModeList[\ 2\ ] = (\ candIntraPredModeC = = 3\ ||\ candIntraPredModeC = = 4\ )\ ?\ candIntraPredModeC + 1 : candIntraPredModeC - 2 \tag{225}$$

$$extCandModeList[\ 3\ ] = (\ candIntraPredModeC = = 32\ ||\ candIntraPredModeC = = 31\ )\ ?\ candIntraPredModeC - 1 : candIntraPredModeC + 2 \tag{226}$$

  — extCandModeList[x] with x = 4..7 is derived as follows:

    — A list[y] with y = 0..9 is derived as follows:

$$list[\ 0\ ] = INTRA\_VER \tag{227}$$

$$list[\ 1\ ] = INTRA\_HOR \tag{228}$$

$$list[\ 2\ ] = INTRA\_DIA\_R \tag{229}$$

$$list[\ 3\ ] = INTRA\_PLN \tag{230}$$

$$list[\ 4\ ] = INTRA\_DIA\_L \tag{231}$$

$$list[\ 5\ ] = INTRA\_DIA\_U \tag{232}$$

$$list[\ 6\ ] = INTRA\_VER + 4 \tag{233}$$

$$list[\ 7\ ] = INTRA\_HOR - 4 \tag{234}$$

$$list[\ 8\ ] = INTRA\_VER - 4 \tag{235}$$

$$list[\ 9\ ] = INTRA\_HOR + 4 \tag{236}$$

    — A variable iCount is set equal to 4.

    — For i equal to 0 to 9, inclusive, the following is applied until iCount is greater than 7:

      — For j equal to 0 to iCount – 1, inclusive,the following is applied:

        — When list[i] is equal to among extCandModeList[j], candModeList[0], and candModeList[1], i is incremented by 1 and j is set equal to 0.

— When j is equal to (iCount – 1), extCandModeList[iCount] is set equal to list[i], i and iCount are incremented by 1 and j is set equal to 0.

— Otherwise, if candModeList[0] is less than 3 and candModeList[1] are greater than or equal to 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 equal to Formulae 183 to 206.

— Otherwise, extCandModeList[x] with x = 0..7 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 (237)

extCandModeList[1] = INTRA_DC (238)

extCandModeList[2] = candIntraPredModeC (239)

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

extCandModeList[0] = (candModeList[0] = = INTRA_BI) ? INTRA_DC : INTRA_BI (240)

extCandModeList[1] = INTRA_PLN (241)

extCandModeList[2] = candIntraPredModeC (242)

— extCandModeList[x] with x = 3..7 is derived as follows:

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

list[0] = (candIntraPredModeC = = 3 | | candIntraPredModeC = = 4) ? candIntraPredModeC + 1 : candIntraPredModeC – 2 (243)

list[1] = (candIntraPredModeC = = 32 | | candIntraPredModeC = = 31) ? candIntraPredModeC – 1 : candIntraPredModeC + 2 (244)

list[2] = (candModeList[1] = = 3 | | candModeList[1] = = 4) ? candModeList[1] + 1 : candModeList[1] – 2 (245)

list[3] = (candModeList[1] = = 32 | | candModeList[1] = = 31) ? candModeList[1] – 1 : candModeList[1] + 2 (246)

list[4] = (candIntraPredModeC + candModeList[1] + 1) >> 1 (247)

list[5] = (candIntraPredModeC + list[4] + 1) >> 1 (248)

list[6] = (candModeList[1] + list[4] + 1) >> 1 (249)

list[7] = INTRA_VER (250)

list[8] = INTRA_HOR (251)

list[9] = INTRA_DIA_R (252)

list[10] = INTRA_PLN (253)

list[11] = INTRA_DIA_L (254)

list[12] = INTRA_DIA_U (255)

list[13] = INTRA_VER + 4 (256)

list[14] = INTRA_HOR – 4 (257)

- A variable iCount is set equal to 3.
- For i equal to 0 to 14, inclusive, the following is applied until iCount is greater than 7:
  - For j equal to 0 to iCount – 1, inclusive, the following is applied:
    - When list[i] is equal to among extCandModeList[j], candModeList[0], and candModeList[1], i is incremented by 1 and j is set equal to 0.
    - When j is equal to (iCount – 1), extCandModeList[iCount] is set equal to list[i], i and iCount are incremented by 1 and j is set equal to 0.

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

extCandModeList[0] = candIntraPredModeC (258)

extCandModeList[1] = (candIntraPredModeC = = INTRA_BI) ? INTRA_DC : INTRA_BI (259)

  - extCandModeList[x] with x = 2..7 is derived as follows:
    - A list[x] with x = 0..14 is set equal to Formulae 209 to 223.
    - A variable iCount is set equal to 2.
    - For i equal to 0 to 14, inclusive, the following is applied until iCount is greater than 7:
      - For j equal to 0 to iCount – 1, inclusive, the following applies:
        - When list[i] is equal to among extCandModeList[j], candModeList[0], and candModeList[1], i is incremented by 1 and j is set equal to 0.
        - When j is equal to (iCount – 1), extCandModeList[iCount] is set equal to list[i], i and iCount are incremented by 1 and j is set equal to 0.

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

extCandModeList[0] = INTRA_BI (260)

extCandModeList[1] = INTRA_DC (261)

extCandModeList[2] = candIntraPredModeC (262)

— extCandModeList[x] with x = 3..7 is derived as follows:

— 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 (263)

list[1] = (candModeList[0] = = 31) ?
candModeList[0] − 1 : candModeList[0] + 2 (264)

list[2] = (candModeList[1] = = 4) ?
candModeList[1] + 1 : candModeList[1] − 2 (265)

list[3] = (candModeList[1] = = 32 | | candModeList[1] = = 31) ?
candModeList[0] − 1 : candModeList[0] + 2 (266)

list[4] = (candIntraPredModeC = = 3 | | candIntraPredModeC = = 4) ?
candIntraPredModeC + 1 : candIntraPredModeC − 2 (267)

list[5] = (candIntraPredModeC = = 32 | | candIntraPredModeC = = 31) ?
candIntraPredModeC − 1 : candIntraPredModeC + 2 (268)

list[6] = (candIntraPredModeC < candModeList[1]) ?
(candModeList[0] + condIntraPredModeC + 1) >> 1 :
(candModeList[0] + candModeList[1] + 1) >> 1 (269)

list[7] = (candIntraPredModeC < candModeList[0]) ?
(candModeList[0] + candModeList[1] + 1) >> 1 :
(candIntraPredModeC + candModeList[1] + 1) >> 1 (270)

list[8] = INTRA_VER (271)

list[9] = INTRA_HOR (272)

list[10] = INTRA_DIA_R (273)

list[11] = INTRA_PLN (274)

list[12] = INTRA_DIA_L (275)

list[13] = INTRA_DIA_U (276)

list[14] = INTRA_VER + 4 (277)

list[15] = INTRA_HOR − 4 (278)

— A variable iCount is set equal to 3.

— For i equal to 0 to 15, inclusive, the following is applied until iCount is greater than 7:

— For j equal to 0 to iCount – 1, inclusive, the following applies:

— When list[i] is equal to among extCandModeList[j], candModeList[0], and candModeList[1], i is incremented by 1 and j is set equal to 0.

— When j is equal to (iCount – 1), extCandModeList[iCount] is set equal to list[i], i and iCount are incremented by 1 and j is set equal to 0.

5) A list remModeList[x] with x = 0..32 is derived as follows:

— remModeList[x] with x = 0, 1 is set equal to candModeList[y] with y = 0, 1.

— remModeList[x] with x = 2..9 is set equal to extCandModeList[y] with y = 0..7.

— remModeList[x] with x = 10..32 is derived as follows:

— A variable iCount is set equal to 10.

— A defaultModeList[33] is set equal to { 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, the following applies:

— If defaultModeList[i] is equal to candModeList[y] with y = 0, 1 or the extCandModeList[z] with z = 0..7, i is incremented by 1.

— Otherwise, remModeList[iCount] is set equal to defaultModeList[i] and iCount is incremented by 1.

6) 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]].

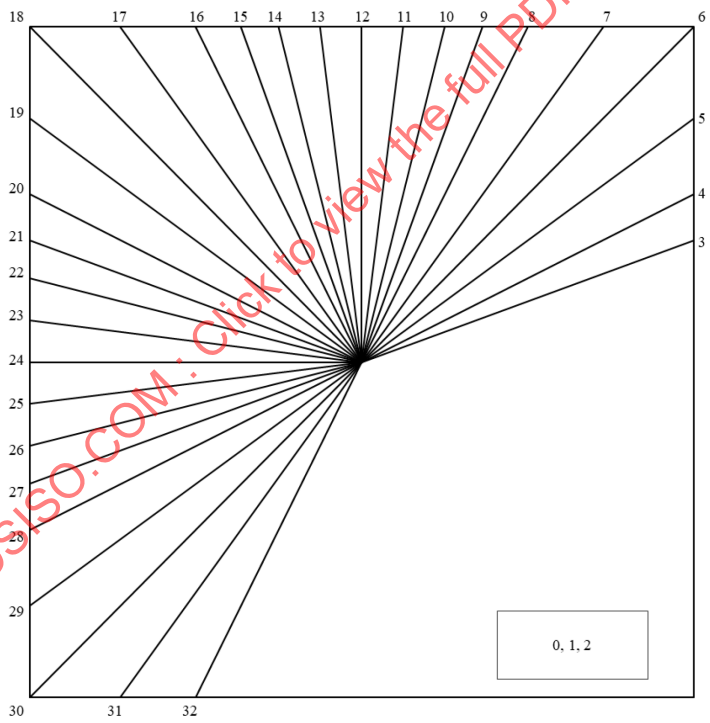
— Otherwise, the IntraPredModeY[xCb][yCb] is set equal to remModeList[intra_luma_pred_rem_mode[x0][y0] + 2 + 8].

— Table 15 specifies the value for the intra prediction mode and the associated names.

Table 15 — Specification of intra prediction mode and associated names (when sps_eipd_flag = = 1)

Intra prediction mode (IntraPredModeY[xCb][yCb])	Associated name
0	INTRA_DC
1	INTRA_PLN
2	INTRA_BI
6	INTRA_DIA_L
12	INTRA_VER
18	INTRA_DIA_R
24	INTRA_HOR
30	INTRA_DIA_U

— IntraPredModeY[xCb][yCb] labelled 0..32 represents directions of predictions as illustrated in Figure 5.



Key

0 INTRA_DC
1 INTRA_PLN
2 INTRA_BI

Figure 5 — Intra prediction mode directions

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.

When LumaPredMode[xCb][yCb] is equal to MODE_IBC, IntraPredModeY[xCb][yCb] is set equal to INTRA_DC.

The chroma intra prediction mode IntraPredModeC is derived as follows:

— If intra_chroma_pred_mode[xCb][yCb] is equal to 0, IntraPredModeC is set 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[xCb][yCb] is equal to or greater than modeIdx, IntraPredModeC is set equal to intra_chroma_pred_mode[xCb][yCb] + 1.

    — Otherwise, IntraPredModeC is set equal to intra_chroma_pred_mode[xCb][yCb].

  — Otherwise, IntraPredModeC is set equal to intra_chroma_pred_mode[xCb][yCb].

Table 16 specifies the value for the chroma intra prediction mode.

Table 16 — 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

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,

— variables log2BlkWidth and log2BlkHeight specifying the width and height of the current coding block,

— a variable predModeIntra specifying the intra prediction mode, and

— 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.

If 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:

$$(xNbCmp, yNbCmp) = (xCbCmp + x, yCbCmp + y) \quad (279)$$

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

$$(xCbY, yCbY) = (cIdx == 0) ? (xCbCmp, yCbCmp) : (xCbCmp * SubWidthC, yCbCmp * SubHeightC) \quad (280)$$

$$(xNbY, yNbY) = (cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp * SubWidthC, yNbCmp * SubHeightC) \quad (281)$$

  — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xNbY, yNbY) as input, 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.

      — LumaPredMode[xNbY][yNbY] 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 as specified in subclause 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 outputs.

Otherwise (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) (282)

— 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) (283)

(xNbY, yNbY) =
(cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp * SubWidthC, yNbCmp * SubHeightC) (284)

— The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xNbY, yNbY) as input, 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.

— LumaCuPredMode[xNbY][yNbY] 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 as specified in subclause 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 outputs.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in subclause 6.4.2 with the luma location (xCbCmp, yCbCmp) and the luma coding block width nCbW as inputs.

Depending on the value of predModeIntra, the following applies:

— If predModeIntra is equal to INTRA_DC, the corresponding intra prediction mode specified in subclause 8.4.4.3 is invoked with the sample array p, the coding block width nCbW, height nCbH and the neigbhoring availability value availLR as inputs, and the outputs are the predicted sample array predSamples.

— Otherwise, if predModeIntra is equal to INTRA_HOR, the corresponding intra prediction mode specified in subclause 8.4.4.4 is invoked with the sample array p, the coding block width nCbW, height nCbH and the neigbhoring availability value availLR as inputs, and the outputs are the predicted sample array predSamples.

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

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

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

— Otherwise, if predModeIntra is equal to INTRA_BI, the corresponding intra prediction mode specified in subclause 8.4.4.8 is invoked with the sample array p, the block size nCbW, nCbH, log2BlkWidth, log2BlkHeight and the neigbhoring availability value availLR as inputs, and the outputs are the predicted sample array predSamples.

— Otherwise, if predModeIntra is equal to INTRA_PLN, the corresponding intra prediction mode specified in subclause 8.4.4.9 is invoked with the sample array p, the block size nCbW, nCbH, log2BlkWidth, log2BlkHeight and the neigbhoring availability value availLR as inputs, and the outputs are the predicted sample array predSamples.

— Otherwise, the corresponding intra prediction mode specified in subclause 8.4.4.10 is invoked with the intra prediction mode predModeIntra, the sample array p, the coding block width nCbW, height nCbH and the neigbhoring availability value availLR as inputs, and the outputs are 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, and

— 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, and

— 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 $BitDepth_C$.

If 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:

  — When 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] is 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] is 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 (sps_eipd_flag is equal to 1), the following applies:

— If 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:

    — When 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] is 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] is 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] is 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 (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:

— When 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] is 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] is 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],

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

— the variable availLR specifying left and right neighbouring blocks availabilities of luma coding block.

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

If 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:

$$\text{predSamples}[x][y] = \left(\sum_{x'=0}^{nCbW-1} p[x'][-1] + \sum_{y'=0}^{nCbH-1} p[-1][y'] + nCbW \right) >> \left(\text{Log2}(nCbW) + 1 \right) \quad (285)$$

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

— The scaling coefficients divScaleMult[idx] with idx = 0..7 are specified in Table 17. 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:

$$predSamples[x][y] = ((\sum_{x'=0}^{nCbW-1} p[x'][-1] + \sum_{y'=0}^{nCbH-1} p[-1][y'] + \sum_{y'=0}^{nCbH-1} p[nCbW][y'] + ((nCbW + nCbH + nCbH) >> 1) * divScaleMult[log2AspRatio]) >> (divScaleShift + aspRatioShift) \quad (286)$$

— 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:

$$predSamples[x][y] = ((\sum_{x'=0}^{nCbW-1} p[x'][-1] + \sum_{y'=0}^{nCbH-1} p[nCbW][y'] + ((nCbW + nCbH) >> 1)) * divScaleMult[log2AspRatio]) >> (divScaleShift + aspRatioShift) \quad (287)$$

— Otherwise (availLR is equal to LR_10 or LR_00), 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:

$$predSamples[x][y] = ((\sum_{x'=0}^{nCbW-1} p[x'][-1] + \sum_{y'=0}^{nCbH-1} p[-1][y'] + ((nCbW + nCbH) >> 1)) * divScaleMult[log2AspRatio]) >> (divScaleShift + aspRatioShift) \quad (288)$$

Table 17 — 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],

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

— the variable availLR specifying left and right neighbouring blocks availabilities of luma coding block.

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

If 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 follow:

predSamples[x][y] = p[−1][y] (289)

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

— The scaling coefficients divScaleMult[idx] with idx = 0..7 are specified in Table 17. 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] = ((p[−1][y] * (nCbW − x) + p[nCbW][y] * (x + 1) + (nCbW >> 1)) * divScaleMult[Log2(nCbW)]) >> divScaleShift (290)

  — 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] (291)

  — Otherwise (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] (292)

8.4.4.5 Specification of intra prediction mode INTRA_VER

Inputs to this process are:

— the neighbouring samples p[x][y], and

— 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] (293)

8.4.4.6 Specification of intra prediction mode INTRA_UL

Inputs to this process are:

— the neighbouring samples p[x][y], and

— two 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] (294)

— Otherwise, the following applies:

predSamples[x][y] = p[x − y − 1][−1] (295)

8.4.4.7 Specification of intra prediction mode INTRA_UR

Inputs to this process are:

— the neighbouring samples p[x][y], and

— two 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]) >> 1 (296)

8.4.4.8 Specification of intra prediction mode INTRA_BI

Inputs to this process are:

— the neighbouring samples p[x][y],

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

— two variables log2BlkWidth and log2BlkHeight specifying the width and height of the current coding block, and

— the variable availLR specifying left and right neighbouring blocks availabilities of luma 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 scaling coefficients divScaleMult[idx] with idx = 0..7 are specified in Table 17. The normalization factor divScaleShift = 12. The weighting factor weightFactor[absLog2DiffWH] with absLog2DiffWH = 1..5 are specified in Table 18.

Table 18 — Specification of weightFactor[absLog2DiffWH] for various input values of absLog2DiffWH

absLog2DiffWH	1	2	3	4	5
weightFactor[absLog2DiffWH]	341	205	114	60	31

Depending on the value of availLR, the following applies:

— If 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:

```
predSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1,
    ( ( ( ( p[ −1 ][ y ] * ( nCbW − x ) + p[ nCbW ][ y ] * ( x + 1 ) + (nCbW >> 1 ) ) *
    divScaleMult[ Log2(nCbW) ] ) >> divScaleShift ) +
    ( ( p[ x ][ − 1 ] * ( nCbH − 1 − y ) + ( ( ( p[ −1 ][ nCbH − 1 ] * ( nCbW − x ) +
    p[ nCbW ][ nCbH − 1 ] * ( x + 1 ) + ( nCbW >> 1 ) ) * divScaleMult[ Log2( nCbW ) ] ) >>
    divScaleShift ) * ( y + 1 ) + ( nCbH >> 1 ) ) >> Log2( nCbH ) ) + 1 ) >> 1 )          (297)
```

— 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 ]                                                                      (298)

iB = p[ nCbW ][ nCbH ]                                                                  (299)
```

– If nCbW is equal to nCbH, the variable iC is derived as follows:

```
iC = ( iA + iB + 1 ) >> 1                                                               (300)
```

– Otherwise, the variable iC is derived as follows:

```
iShift = Min( log2BlkWidth, log2BlkHeight ) )                                           (301)

absLog2DiffWH = Abs( log2BlkWidth − log2BlkHeight )                                     (302)

iC = ( ( ( iA << log2BlkWidth ) + ( iB << log2BlkHeight ) ) *
weightFactor[ absLog2DiffWH ]
            + ( 1 << ( iShift + 9 ) ) ) >> ( iShift + 10 )                              (303)
```

— The values of the prediction samples predSamples[x][y] are derived as follows:

```
predSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1,
    ( ( ( ( iA − p[ nCbW ][ y ] ) * ( x + 1 ) ) << log2BlkHeight ) + ( ( ( iB − p[ x ][ −1 ] ) *
    ( y + 1 ) ) << log2BlkWidth ) + ( ( p[ x ][ −1 ] + p[ nCbW ][ y ] ) << ( log2BlkWidth +
    log2BlkHeight ) ) + ( ( iC << 1 ) − iA − iB ) * x * y + ( 1 << ( log2BlkWidth +
    log2BlkHeight ) ) ) >> ( log2BlkWidth + log2BlkHeight + 1 ) )                       (304)
```

— Otherwise (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] (305)

iB = p[−1][nCbH] (306)

— If nCbW is equal to nCbH, the variable iC is derived as follows:

iC = (iA + iB + 1) >> 1 (307)

— Otherwise, the variable iC is derived as follows:

iShift = Min(log2BlkWidth, log2BlkHeight) (308)

absLog2DiffWH = Abs(log2BlkWidth − log2BlkHeight) (309)

iC = (((iA << log2BlkWidth) + (iB << log2BlkHeight)) *
weightFactor[absLog2DiffWH] +
(1 << (iShift + 9))) >> (iShift + 10) (310)

— 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]) * (y + 1))
<< log2BlkWidth) + ((p[x][−1] + p[−1][y]) << (log2BlkWidth + log2BlkHeight)) +
((iC << 1) − iA − iB) * x * y + (1 << (log2BlkWidth + log2BlkHeight))) >>
(log2BlkWidth + log2BlkHeight + 1)) (311)

8.4.4.9 Specification of intra prediction mode INTRA_PLN

Inputs to this process are:

— the neighbouring samples p[x][y],

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

— two variables log2BlkWidth and log2BlkHeight specifying the width and height of the current coding block, and

— the variable availLR specifying left and right neighbouring blocks availabilities of luma 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 scaling and shift factors mult[i] and shift[i] with i = 2..7 are specified in Table 19.

The variables idxW and idxH are derived as follows:

idxW = Max(log2BlkWidth, 2) (312)

idxH = Max(log2BlkHeight, 2) (313)

Depending on the value of availLR, the following applies:

— If availLR is equal to LR_11 or LR_01, the values of the prediction samples predSamples[x][y] are derived as follows:

$$iH = \sum_{x'=0}^{(nCbW/2)-1} (x'+1) * (p[(nCbW/2)-x'-1][-1] - p[(nCbW/2)+x'+1][-1]) \quad (314)$$

$$iV = \sum_{y'=0}^{(nCbH/2)-1} (y'+1) * (p[nCbW][(nCbH/2)+y'] - p[nCbW][(nCbH/2)-y'-2]) \quad (315)$$

$$iA = (p[\,0\,][\,-1\,] + p[\,nCbW\,][\,nCbH-1\,]) << 4 \quad (316)$$

$$iB = ((iH << 5) * mult[\,idxH\,] + (1 << (shift[\,idxH\,] - 1))) >> shift[\,idxH\,] \quad (317)$$

$$iC = ((iV << 5) * mult[\,idxW\,] + (1 << (shift[\,idxW\,] - 1))) >> shift[\,idxW\,] \quad (318)$$

$$predSamples[\,x\,][\,y\,] = Clip3(0, (1 << bitDepth) - 1, (iA + (x - ((nCbW >> 1) - 1)) * iB + (y - ((nCbH >> 1) - 1)) * iC + 16) >> 5) \quad (319)$$

— Otherwise (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:

$$iH = \sum_{x'=0}^{(nCbW/2)-1} (x'+1) * (p[(nCbW/2)+x'][-1] - p[(nCbW/2)-x'-2][-1] \quad (320)$$

$$iV = \sum_{y'=0}^{(nCbH/w)-1} (y'+1) * (p[-1][(nCbH/2)+y'] - p[-1][(nCbH/2)-y'-2] \quad (321)$$

$$iA = (p[\,nCbW-1\,][\,-1\,] + p[\,-1\,][\,nCbH-1\,]) << 4 \quad (322)$$

$$iB = ((iH << 5) * mult[\,idxH\,] + (1 << (shift[\,idxH\,] - 1))) >> shift[\,idxH\,] \quad (323)$$

$$iC = ((iV << 5) * mult[\,idxW\,] + (1 << (shift[\,idxW\,] - 1))) >> shift[\,idxW\,] \quad (324)$$

$$predSamples[\,x\,][\,y\,] = Clip3(0, (1 << bitDepth) - 1, (iA + (x - ((nCbW >> 1) - 1)) * iB + (y - ((nCbH >> 1) - 1)) * iC + 16) >> 5) \quad (325)$$

Table 19 — Specification of mult[i] and shift[i]

i	2	3	4	5	6	7
mult[i]	13	17	5	11	23	47
shift[i]	7	10	11	15	19	23

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],

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

— the variable availLR specifying left and right neighbouring blocks availabilities of luma coding block.

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

The variables dirXYSign, divDxy and divDyx are defined in Table 20.

Arrays refUp, refLeft, and refRight are derived as follows:

— refUp[x] is assigned with the neighbouring samples p[x][y] with x = −1..nCbH + nCbW − 1, y = −1.

— refLeft[y] is assigned with the neighbouring samples p[x][y], with x = −1, y = −1..nCbH + nCbW − 1.

— refRight[y] is assigned with the neighbouring samples p[x][y], with x = nCbW, y = −1..nCbH + nCbW − 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:

1) The values iOffset, iX, iY and refPosition are specified as follows:

— Depending on the values of availLR, the following applies:

  — If availLR is equal to LR_01 or LR_11, the following applies:

    iTanY = (((y + 1) * divDxy) >> 10) (326)

    iTanX = (((x + 1) * divDyx) >> 10) (327)

    — If predModeIntra is less than INTRA_VER, the following applies:

      — If x is less than nCbW − iTanY (refer to upper sample), the following applies:

        iOffset = (((y + 1) * divDxy) >> 5) − (iTanY << 5) (328)

        iX = x + iTanY (329)

        refPosition = refUp (330)

      — Otherwise (refer to right sample), the following applies:

        iOffset = (((nCbW − x) * divDyx) >> 5) − ((((nCbW − x) * divDyx) >> 10) << 5) (331)

        iX = x + (((nCbW − x) * divDyx) >> 10) (332)

        refPosition = refRight (333)

    — Otherwise, if predModeIntra is greater than INTRA_HOR, the following applies:

      iTanX = (((nCbW − x) * divDyx) >> 10) (334)

      iTanY = (((nCbW − x) * divDxy) >> 10) (335)

      — If y is less than iTanX (refer to upper sample), the following applies:

        iOffset = (((nCbW − x) * divDxy) >> 5) − (iTanY << 5) (336)

        iX = x + iTanY (337)

refPosition = refUp (338)

— Otherwise (refer to right sample), the following applies:

iOffset = (((nCbW − x) * divDxy) >> 5) − (iTanX << 5) (339)

iY = y − iTanX (340)

refPosition = refRight (341)

— Otherwise, the following applies:

— If y is less than iTanX (refer to upper sample), the following applies:

iOffset = (((y + 1) * divDxy) >> 5) − (iTanY << 5) (342)

iX = x − iTanY (343)

refPosition = refUp (344)

— Otherwise, if avail_LR is equal to LR_01 (refer to right sample), the following applies:

iOffset = (((nCbW − x) * divDxy) >> 5) − ((((nCbW − x) * divDyx) >> 10) << 5) (345)

iY = y + (((nCbW − x) * divDyx) >> 10) (346)

refPosition = refRight (347)

— Otherwise (refer to left sample), the following applies:

iOffset = (((x + 1) * divDyx) >> 5) − (iTanX << 5) (348)

iY = y − iTanX (349)

refPosition = refLeft (350)

— Otherwise (availLR is equal to LR_10 or LR_00), the following applies:

iTanY = (((y + 1) * divDxy) >> 10) (351)

iTanX = (((x + 1) * divDyx) >> 10) (352)

— If predModeIntra is less than INTRA_VER (refer to upper sample), the following applies:

iOffset = (((y + 1) * divDxy) >> 5) − (iTanY << 5) (353)

iX = x + iTanY (354)

refPosition = refUp (355)

— Otherwise, if predModeIntra is greater than INTRA_HOR (refer to left sample), the following applies:

iOffset = (((x + 1) * divDyx) >> 5) − (iTanX << 5) (356)

iY = y + iTanX (357)

refPosition = refLeft (358)

— Otherwise, the following applies:

— If y is less than iTanX (refer to upper sample), the following applies:

iOffset = (((y + 1) * divDxy) >> 5) − (iTanY << 5) (359)

iX = x − iTanY (360)

refPosition = refUp (361)

— Otherwise (refer to left sample), the following applies:

iOffset = (((x + 1) * divDyx) >> 5) − (iTanX << 5) (362)

iY = y − iTanX (363)

refPosition = refLeft (364)

2) The value of the prediction samples predSamples[x][y] are derived as follows:

— If refPosition is equal to refUp (refer to upper sample), the following applies:

— A variable clipMax is set equal to (nCbW + nCbH − 1).

— A variable clipMin is set equal to −1.

— If dirXYSign is equal to −1, the following applies:

iXn = Clip3(clipMin, clipMax, iX + 1) (365)

iXnP2 = Clip3(clipMin, clipMax, iX + 2) (366)

iXnN1 = Clip3(clipMin, clipMax, iX − 1) (367)

— Otherwise, the following applies:

iXn = Clip3(clipMin, clipMax, iX − 1) (368)

iXnP2 = Clip3(clipMin, clipMax, iX − 2) (369)

iXnN1 = Clip3(clipMin, clipMax, iX + 1) (370)

— 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 (371)

— Otherwise, if refPosition is equal to refLeft (refer to left sample), the following applies:

— A variable clipMax is set equal to (nCbW + nCbH − 1).

— A variable clipMin is set equal to −1.

— If dirXYSign is equal to –1, the following applies:

iYn = Clip3(clipMin, clipMax, iY + 1) (372)

iYnP2 = Clip3(clipMin, clipMax, iY + 2) (373)

iYnN1 = Clip3(clipMin, clipMax, iY − 1) (374)

— Otherwise, the following applies:

iYn = Clip3(clipMin, clipMax, iY − 1) (375)

iYnP2 = Clip3(clipMin, clipMax, iY − 2) (376)

iYnN1 = Clip3(clipMin, clipMax, iY + 1) (377)

— 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 (378)

— Otherwise (refPosition is equal to refRight), the following applies:

— A variable clipMax is set equal to (nCbW + nCbH − 1).

— A variable clipMin is set equal to –1.

— If dirXYSign is equal to –1, the following applies:

iYn = Clip3(clipMin, clipMax, iY − 1) (379)

iYnP2 = Clip3(clipMin, clipMax, iY − 2) (380)

iYnN1 = Clip3(clipMin, clipMax, iY + 1) (381)

— Otherwise, the following applies:

iYn = Clip3(clipMin, clipMax, iY + 1) (382)

iYnP2 = Clip3(clipMin, clipMax, iY + 2) (383)

iYnN1 = Clip3(clipMin, clipMax, iY − 1) (384)

— 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 (385)

Table 20 — Specification of dirXYSign, divDxy and divDyx

predModeIntra	dirXYSign	divDxy	divDyx
3	−1	2816	372
4	−1	2048	512

predModeIntra	dirXYSign	divDxy	divDyx
5	−1	1408	744
6	−1	1024	1024
7	−1	744	1408
8	−1	512	2048
9	−1	372	2816
10	−1	256	4096
11	−1	128	8192
12	-	-	-
13	1	128	8192
14	1	256	4096
15	1	372	2816
16	1	512	2048
17	1	744	1408
18	1	1024	1024
19	1	1408	744
20	1	2048	512
21	1	2816	372
22	1	4096	256
23	1	8192	128
24	-	-	-
25	−1	8192	128
26	−1	4096	256
27	−1	2816	372
28	−1	2048	512
29	−1	1408	744
30	−1	1024	1024
31	−1	744	1408
32	−1	512	2048

8.4.5 Decoding process for the residual signal

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 luma location (xTb, yTb) specifying the top-left samples of the current luma block relative to the top-left sample of the current luma coding block,

— two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

— two variables nTbW and nTbH specifying the width and the height of the current luma coding block,

— a variable cIdx specifying the colour componnet of the current block, and

— an (nCbW)x(nCbH) array resSamples of residual block.

Output of this process is a modified version of the (nCbW)x(nCbH) array resSamples of residual samples.

Depending on the variables nTbW and nTbH, the following applies:

— If both nTbW and nTbH are greater than MaxTbSizeY, the following ordered steps apply:

1) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW / 2 and nTbH set equal to nTbH / 2, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + nTbW / 2, yTb), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW / 2 and nTbH set equal to nTbH / 2, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

3) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb + nTbH / 2), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW / 2 and nTbH set equal to nTbH / 2, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

4) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + nTbW / 2, yTb + nTbH / 2), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW / 2 and nTbH set equal to nTbH / 2, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise, if nTbW is greater than MaxTbSizeY, the following ordered steps apply:

1) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW / 2 and nTbH set equal to nTbH, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + nTbW / 2, yTb), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW / 2 and nTbH set equal to nTbH, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise, if nTbH is greater than MaxTbSizeY, the following ordered steps apply:

1) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW and nTbH set equal to nTbH / 2, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for the residual signal as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb + nTbH / 2), the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, the variable nTbW set equal to nTbW and nTbH set equal to nTbH / 2, the variable cIdx set equal to cIdx, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise (both nTbW and nTbH are less than or equal to MaxTbSizeY), the following ordered steps apply:

1) When cIdx is not equal to 0, nTbW is set equal to nTbW / SubWidthC and nTbH is set equal to nTbH / SubHeightC.

2) The scaling and transformation process as specified in subclause 8.7.2 is invoked with the luma location (xCb + xTb, yCb + yTb), the variable cIdx set equal to cIdx, the transform width nTbW set equal to nTbW and the transform height nTbH 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:

$$\text{resSamples}[\, xTb / (cIdx\ ?\ SubWidthC : 1) + i,\ yTb / (cIdx\ ?\ SubHeightC : 1) + j\,] = \text{transformBlock}[\, i, j\,], \quad (386)$$

with i = 0..nTbW − 1, j = 0..nTbH − 1

8.5 Decoding process for coding units coded in inter prediction mode

8.5.1 General

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, and

— 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 picture before in-loop filtering.

The derivation process for quantization parameters as specified in subclause 8.7.1 is invoked with the luma location (xCb, yCb) as input.

The variables $nCbW_L$ and $nCbH_L$ are set equal to 1 << log2CbWidth and 1 << log2CbHeight, respectively. 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 motion vector components and reference indices of the current coding unit are derived as follows:

   — If affine_flag[xCb][yCb] is equal to 1, the derivation process for affine motion vector components and reference indices as specified in subclause8.5.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width $nCbW_L$, and the luma coding block height $nCbH_L$ as inputs, and the reference indices refIdxLX, the prediction list utilization flags predFlagLX, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the flag specifying motion vector clipping type clipMV, the luma motion vector array mvLX[xSbIdx][ySbIdx], and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, with X being 0 or 1, 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. The DMVR utilization flag dmvrAppliedFlag is set equal to 0.

   — Otherwise, the derivation process for motion vector components and reference indices as specified in subclause 8.5.2.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width $nCbW_L$, and the luma coding block height $nCbH_L$ as inputs, and the luma motion vectors mvL0[0][0] and mvL1[0][0], when ChromaArrayType is not equal to 0, the chroma motion vector mvCL0[0][0] and mvCL1[0][0], the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0], and the DMVR utilization flag dmvrAppliedFlag as outputs.

   — If dmvrAppliedFlag is equal to 1, it is modified as follows:

      — dmvrAppliedFlag is set equal to 0, if any of the following conditions are false:

         — mmvd_flag [xCb][yCb] is equal to 0.

         — predFlagL0 is equal to 1 and predFlagL1 is equal to 1.

         — DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) + DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) is equal to 0.

         — nCbW is greater than or equal to 8 and nCbH is greater than or equal to 8.

      — 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 as specified in subclause 8.5.4.2 with refIdxLX as input.

      — The coded block is partitioned into sub-blocks with the number of luma coding subblocks in horizontal direction numSbXLX and in vertical direction numSbYLX, 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:

$$numSbXL0 = numSbXL1 = (nCbW_L > 16) \ ? \ (nCbW_L >> 4) : 1 \quad (387)$$

$$numSbYL0 = numSbYL1 = (nCbH_L > 16) \ ? \ (nCbH_L >> 4) : 1 \quad (388)$$

$$sbWidth = (nCbW_L > 16) \ ? \ 16 : nCbW_L \quad (389)$$

$$sbHeight = (nCbH_L > 16) \ ? \ 16 : nCbH_L \quad (390)$$

— For xSbIdx = 0..numSbXL0 − 1 and ySbIdx = 0..numSbYL0 − 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:

$$mvLX[xSbIdx][ySbIdx] = mvLX[0][0] \quad (391)$$

$$predFlagLX[xSbIdx][ySbIdx] = predFlagLX[0][0] \quad (392)$$

$$xSb[xSbIdx][ySbIdx] = xCb + xSbIdx * sbWidth \quad (393)$$

$$ySb[xSbIdx][ySbIdx] = yCb + ySbIdx * sbHeight \quad (394)$$

— The decoder side motion vector refinement process as specified in subclause 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 as specified in subclause 8.5.2.6 is invoked with mvLX[xSbIdx][ySbIdx] and refIdxLX as inputs, and mvCLX[xSbIdx][ySbIdx] as outputs with X equal to 0 and 1.

— Otherwise (dmvrAppliedFlag 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 as specified in subclause 8.5.2.6 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 numSbXL0 and numSbXL1 and in vertical direction numSbYL0 and numSbYL1 are all set equal to 1.

2) 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..numSbXL0 − 1, ySbIdx = 0..numSbYL0 − 1:

— If dmvrAppliedFlag is equal to 1, the derivation process for chroma motion vectors as specified in subclause 8.5.2.6 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:

$$mvLX[xSbIdx][ySbIdx] = mvLX[xSbIdx][ySbIdx] << 2 \quad (395)$$

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

$$refMvLX[xSbIdx][ySbIdx] = mvLX[xSbIdx][ySbIdx] + dMvL0[xSbIdx][ySbIdx] \quad (396)$$

— Otherwise (X is equal to 1), the following applies:

refMvLX[xSbIdx][ySbIdx] = mvLX[xSbIdx][ySbIdx] − dMvL0[xSbIdx][ySbIdx] (397)

— The following applies:

refMvLX[xSbIdx][ySbIdx][0] = Clip3(-2^{17}, $2^{17} - 1$, refMvLX[xSbIdx][ySbIdx][0]) (398)

refMvLX[xSbIdx][ySbIdx][1] = Clip3(-2^{17}, $2^{17} - 1$, refMvLX[xSbIdx][ySbIdx][1]) (399)

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

refMvLX[xSbIdx][ySbIdx] = mvLX[xSbIdx][ySbIdx] << 2 (400)

refMvCLX[xSbIdx][ySbIdx] = mvCLX[xSbIdx][ySbIdx] << 2 (401)

NOTE The array refMvLX is used in the derivation process for collocated motion vectors in subclause8.5.2.3.4. mvLX is used in the derivation process for spatial motion vector predictors and boundary strength derivation process of deblocking filter.

3) When affine_flag[xCb][yCb] is equal to 0 and dmvrAppliedFlag is equal to 0, the following applies:

numSbX = 1 (402)

numSbY = 1 (403)

numCpMv = 2 (404)

cpMvL0[0] = mvL0[0][0][0] (405)

cpMvL0[1] = mvL0[0][0][1] (406)

cpMvL1[0] = mvL1[0][0][0] (407)

cpMvL1[1] = mvL1[0][0][1] (408)

mvL0[0][0][0] = mvL0[0][0][0] << 2 (409)

mvL0[0][0][1] = mvL0[0][0][1] << 2 (410)

mvL1[0][0][0] = mvL1[0][0][0] << 2 (411)

mvL1[0][0][1] = mvL1[0][0][1] << 2 (412)

mvCL0[0][0][0] = mvCL0[0][0][0] << 2 (413)

mvCL0[0][0][1] = mvCL0[0][0][1] << 2 (414)

mvCL1[0][0][0] = mvCL1[0][0][0] << 2 (415)

mvCL1[0][0][1] = mvCL1[0][0][1] << 2 (416)

4) The decoding process for inter sample prediction as specified in subclause 8.5.4.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width $nCbW_L$, the luma coding block height $nCbH_L$, the number of luma coding subblocks in horizontal direction numSbX, and in

vertical direction numSbY the luma motion vectors mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and the refined luma motion vectors refMvL0[xSbIdx][ySbIdx] and refMvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, and ySbIdx = 0..numSbY − 1, when ChromaArrayType is not equal to 0, the chroma motion vectors mvCL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the refined chroma motion vectors refMvCL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 and refMvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx] with cpIdx = 0..numCpMv − 1, cpMvL1[cpIdx] with cpIdx = 0..numCpMv − 1 and clipMV flag specifying motion vector clipping type 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.

5) The variable isChromaPresent is set equal to TRUE if ChromaArrayType is not equal to 0, and is set equal to FALSE otherwise.

6) The decoding process for the residual signal of coding units coded in inter prediction mode as specified in subclause 8.5.6.1 is invoked with the luma location (xCb, yCb) and the luma coding block width log2CbWidth, the luma coding block height log2CbHeight and the chroma component presence indicator isChromaPresent as inputs, and the array $resSamples_L$ and, when isChromaPresent, two arrays $resSamples_{Cb}$ and $resSamples_{Cr}$ as outputs.

7) 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 subclause 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, and the output is a modified reconstructed picture before the post-reconstruction filtering process and in-loop filtering.

   — When value of cbf_luma is equal to 1 and sps_htdf_flag is equal to 1, the post-reconstruction filtering process prior to in-loop filtering for a luma component as specified in subclause 8.7.6.1 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 Qp_Y set equal to Qp'_Y, which is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1, and the output is a modified reconstructed picture before in-loop filtering.

   — When isChromaPresent, the picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 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, and the output is a modified reconstructed picture before in-loop filtering.

   — When isChromaPresent, the picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 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, and the output is a modified reconstructed picture before in-loop filtering.

When 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..$nCbW_L$ − 1 and y = 0..$nCbH_L$ − 1:

MvL0[xCb + x][yCb + y] = mvL0[0][0] >> 2 (417)

MvL1[xCb + x][yCb + y] = mvL1[0][0] >> 2 (418)

RefIdxL0[xCb + x][yCb + y] = refIdxL0 (419)

RefIdxL1[xCb + x][yCb + y] = refIdxL1 (420)

PredFlagL0[xCb + x][yCb + y] = predFlagL0[0][0] (421)

PredFlagL1[xCb + x][yCb + y] = predFlagL1[0][0] (422)

When affine_flag[xCb][yCb] is equal to 0, for xSbIdx = 0..numSbXL0 − 1 and ySbIdx = 0..numSbYL0 − 1, the following applies:

— The following assignments are made for x = 0..sbWidth − 1 and y = 0..sbHeight − 1:

MvDmvrL0[xCb + xSbIdx*sbWidth + x][yCb + ySbIdx* sbHeight + y] =
refMvL0[xSbIdx][ySbIdx] >> 2 (423)

MvDmvrL1[xCb + xSbIdx*sbWidth + x][yCb + ySbIdx* sbHeight + y] =
refMvL1[xSbIdx][ySbIdx] >> 2 (424)

8.5.2 Derivation process for motion vector components and reference indices

8.5.2.1 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, and

— 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[0][0] and mvL1[0][0],

— when ChromaArrayType is not equal to 0, the chroma motion vectors mvCL0[0][0] and mvCL1[0][0],

— the reference indices refIdxL0 and refIdxL1,

— the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0], and

— the DMVR utilization flag dmvrAppliedFlag.

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

For the derivation of the variables mvL0[0][0] and mvL1[0][0], refIdxL0 and refIdxL1, predFlagL0[0][0] and predFlagL1[0][0], and dmvrAppliedFlag, the following applies:

— If sps_admvp_flag is equal to 1, and cu_skip_flag[xCb][yCb] or merge_mode_flag[xCb][yCb] is equal to 1, the derivation process for luma motion vectors for merge mode as specified in subclause 8.5.2.3.1 is invoked with the luma location (xCb, yCb), and the variables nCbW and nCbH as inputs, and the outputs being the luma motion vectors mvL0[0][0] and mvL1[0][0], the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0]. The DMVR utilization flag dmvrAppliedFlag is set equal to 1.

— Otherwise, 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 subclause 8.5.2.2 is invoked with the luma location (xCb, yCb), and the variables nCbW and nCbH as inputs, and the outputs being the luma motion vectors mvL0[0][0] and mvL1[0][0], the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0]. The variable dmvrAppliedFlag is set equal to 0.

— Otherwise, if sps_admvp_flag is equal to 0, and direct_mode_flag[xCb][yCb] is equal to 1, the derivation process for luma motion vectors for direct mode as specified in subclause 8.5.2.5 is invoked with the luma location (xCb, yCb), and the variables nCbW and nCbH as inputs, and the outputs being the luma motion vectors mvL0[0][0] and mvL1[0][0], the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0]. The variable dmvrAppliedFlag is set equal to 0.

— Otherwise (direct_mode_flag[xCb][yCb], merge_mode_flag[xCb][yCb] and cu_skip_flag[xCb][yCb] are all equal to 0), for X being replaced by either 0 or 1 in the variables predFlagLX[0][0], mvLX[0][0], refIdxLX and MvdLX, in PRED_LX, and in the syntax elements ref_idx_lX, the following ordered steps apply:

1) The variable dmvrAppliedFlag is set equal to 0.

2) The variables refIdxLX and predFlagLX[0][0] are derived as follows:

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

— If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI, the variables refIdxLX and predFlagLX[0][0] are specified by:

$$refIdxLX = ref\_idx\_lX[xCb][yCb] \quad (425)$$

$$predFlagLX[0][0] = 1 \quad (426)$$

— Otherwise, the variables refIdxLX and predFlagLX[0][0] are specified by:

$$refIdxLX = -1 \quad (427)$$

$$predFlagLX[0][0] = 0 \quad (428)$$

— Otherwise (sps_admvp_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, the variables refIdxLX and predFlagLX[0][0] are specified by:

refIdxLX = ref_idx_lX[xCb][yCb] (429)

predFlagLX[0][0] = 1 (430)

— Otherwise, the variables refIdxLX and predFlagLX[0][0] are specified by:

refIdxLX = −1 (431)

predFlagLX[0][0] = 0 (432)

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

— The variable predFlagLX[0][0] is specified by:

predFlagLX[0][0] = 1 (433)

— The derivation process for luma reference index as specified in subclause 8.5.2.4.5.1 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 refIdxLX.

3) The variable mvdLX is derived as follows:

mvdLX[0] = MvdLX[xCb][yCb][0] (434)

mvdLX[1] = MvdLX[xCb][yCb][1] (435)

4) When predFlagLX[0][0] is equal to 1, the derivation process for luma motion vector prediction as specified in subclause 8.5.2.4.1 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.

5) When predFlagLX[0][0] is equal to 1, the luma motion vector mvLX[0][0] is derived as follows:

uLX[0] = (mvpLX[0] + mvdLX[0] + 2^{16}) % 2^{16} (436)

mvLX[0][0][0] = (uLX[0] >= 2^{15}) ? (uLX[0] − 2^{16}) : uLX[0] (437)

uLX[1] = (mvpLX[1] + mvdLX[1] + 2^{16}) % 2^{16} (438)

mvLX[0][0][1] = (uLX[1] >= 2^{15}) ? (uLX[1] − 2^{16}) : uLX[1] (439)

NOTE The resulting values of mvLX[0][0][0] and mvLX[0][0][1] as specified above are always in the range of -2^{15} to $2^{15} - 1$, inclusive.

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 as specified in subclause 8.5.2.6 is invoked with mvLX[0][0] as input, and the output being mvCLX[0][0].

When sps_hmvp_flag is equal to 1, the updating process for the history-based motion vector predictor list as specified in subclause 8.5.2.7. is invoked with luma motion vectors mvL0[0][0] and mvL1[0][0], reference indices refIdxL0 and refIdxL1 as inputs.

8.5.2.2 Derivation process for luma motion vectors for skip 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, and

— 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[0][0] and mvL1[0][0],

— the reference indices refIdxL0 and refIdxL1, and

— the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0].

The motion vectors mvL0[0][0] and mvL1[0][0], the reference indices refIdxL0 and refIdxL1, and the prediction utilization flags predFlagL0[0][0] and predFlagL1[0][0] are derived by the following ordered steps:

1) When slice_type is equal to P, the variables mvL1[0][0], refIdxL1 and predFlagL1[0][0] are set as follows:

$$mvL1[0][0][0] = 0 \quad (440)$$

$$mvL1[0][0][1] = 0 \quad (441)$$

$$refIdxL1 = -1 \quad (442)$$

$$predFlagL1[0][0] = 0 \quad (443)$$

2) The derivation process for motion vector predictor from neighbouring coding unit partitions as specified in subclause 8.5.2.4.3 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[0][0], refIdxL0 and predFlagL0[0][0] are set as follows:

$$mvL0[0][0] = mvPred \quad (444)$$

$$refIdxL0 = 0 \quad (445)$$

$$predFlagL0[0][0] = 1 \quad (446)$$

3) When slice_type is equal to B, the derivation process for motion vector predictor from neighbouring coding unit partitions as specified in subclause 8.5.2.4.3 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[0][0], refIdxL1 and predFlagL1[0][0] are set as follows:

$$mvL1[0][0] = mvPred \quad (447)$$

$$refIdxL1 = 0 \quad (448)$$

predFlagL1[0][0] = 1 (449)

8.5.2.3 Derivation process for luma motion vectors and reference indices for merge mode

8.5.2.3.1 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, and

— 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[0][0] and mvL1[0][0],

— the reference indices refIdxL0 and refIdxL1, and

— the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0].

Variables mLSize is set equal to ((nCbW * nCbH <= 32) ? 4 : 6), merging candidate list mergeCandList is set equal to the empty list and number of elements within mergeCandList numCurrMergeCand is set equal to 0.

The motion vectors mvL0[0][0] and mvL1[0][0], the reference indices refIdxL0 and refIdxL1, and the prediction utilization flags predFlagL0[0][0] and predFlagL1[0][0] are derived by the following ordered steps:

1) The derivation process for spatial merge candidates as specified in subclause 8.5.2.3.2 is invoked with the luma coding block location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, mergeCandList, numCurrMergeCand and maximal size of merge candidate list mLSize as inputs, updated mergeCandList and number of elements numCurrMergeCand within mergeCandList as outputs.

2) The derivation process for temporal merge candidates as specified in subclause 8.5.2.3.3 is invoked with the luma location (xCb, yCb), the luma coding block width nCbW, the luma coding block height nCbH, mergeCandList and numCurrMergeCand as inputs, updated mergeCandList and number of elements numCurrMergeCand within mergeCandList as outputs.

3) When sps_hmvp_flag is equal to 1, 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 subclause 8.5.2.3.6 is invoked with mergeCandList, numCurrMergeCand, maximal size of merge candidate list mLSize, the coding block width nCbW and the coding block height nCbH as inputs, and modified mergeCandList and numCurrMergeCand as outputs.

4) 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 as specified in subclause 8.5.2.3.7 is invoked with mergeCandList, numCurrMergeCand and maximal size of merge candidate list mLSize as inputs, and modified mergeCandList and numCurrMergeCand as outputs.

5) When numCurrMergeCand is less than mLSize, the derivation process for zero motion vector merging candidates as specified in subclause 8.5.2.3.8 is invoked with the coding block width nCbW

and the coding block height nCbH, the mergeCandList, numCurrMergeCand and maximal size of merge candidate list mLSize as inputs, and modified mergeCandList and numCurrMergeCand as outputs.

6) The following assignments are made with N being the candidate at position merge_idx[xCb][yCb] in the merging candidate list mergeCandList (N = mergeCandList[merge_idx[xCb][yCb]]) and X being replaced by 0 or 1:

$$refIdxLX = refIdxLXN \quad (450)$$

$$mvLX[0][0][0] = mvLXN[0] \quad (451)$$

$$mvLX[0][0][1] = mvLXN[1] \quad (452)$$

— When refIdxLX is a valid reference picture, the following applies:

$$predFlagLX[0][0] = 1 \quad (453)$$

7) When mmvd_flag[xCb][yCb] is equal to 1, the following applies:

— The following assignments are made with N being the candidate at position mmvd_merge_idx[Cb][yCb] in the merging candidate list mergeCandList (N = mergeCandList[mmvd_merge_idx[xCb][yCb]]) and X being replaced by 0 or 1:

$$refIdxLX = refIdxLXN \quad (454)$$

$$mvLX[0][0][0] = mvLXN[0] \quad (455)$$

$$mvLX[0][0][1] = mvLXN[1] \quad (456)$$

— When refIdxLX is a valid reference picture, the following applies:

$$predFlagLX[0][0] = 1 \quad (457)$$

— The derivation process for MMVD motion vector as specified in subclause 8.5.2.3.9 is invoked with the luma location (xCb, yCb), the luma motion vectors mvL0[0][0] and mvL1[0][0], the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0] as inputs, and the MMVD motion vectors mMvL0 and mMvL1, the modified reference indices refIdxL0 and refIdxL1, and the modified 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 mMvLX for X being 0 and 1 as follows:

$$mvLX[0][0][0] = mMvLX[0] \quad (458)$$

$$mvLX[0][0][1] = mMvLX[1] \quad (459)$$

8.5.2.3.2 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,

— a merging candidate list mergeCandList,

— a number of elements numCurrMergeCand within mergeCandList, and

— a maximal number of elements mLSize within mergeCandList.

Outputs of this process are:

— the merging candidate list mergeCandList, and

— the number of elements numCurrMergeCand within mergeCandList.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in subclause 6.4.2 with the luma location (xCb, yCb) and the luma coding block width nCbW as inputs.

If 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 + nCbH − 1).

— The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW, yCb + nCbH − 1).

— 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, if 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, if 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 $refIdxL0A_1$, $refIdxL1A_1$, $predFlagL0A_1$, $predFlagL1A_1$, $mvL0A_1$ and $mvL1A_1$ the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_1$, $yNbA_1$) as input, and the output is assigned to the coding block availability flag $availableA_1$.

— The variables $refIdxLXA_1$, $predFlagLXA_1$ and $mvLXA_1$ are derived as follows for X being 0 or 1:
  — If $availableA_1$ is equal to FALSE, 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.
  — Otherwise, the following assignments are made:

$$mvLXA_1 = MvLX[\ xNbA_1\ ][\ yNbA_1\ ] \tag{460}$$

$$refIdxLXA_1 = RefIdxLX[\ xNbA_1\ ][\ yNbA_1\ ] \tag{461}$$

$$predFlagLXA_1 = PredFlagLX[\ xNbA_1\ ][\ yNbA_1\ ] \tag{462}$$

— When neither of $refIdxL0A_1$ and $refIdxL1A_1$ is equal to −1 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

$$refIdxL1A_1 = -1 \tag{463}$$

$$predFlagL1A_1 = 0 \tag{464}$$

— When $availableA_1$ is equal to TRUE, the variable mergeCandList[numCurrMergeCand++] is updated with $mvLXA_1$, $refIdxLXA_1$ and $predFlagLXA_1$ for X being 0 or 1.

For the derivation of $refIdxL0B_1$, $refIdxL1B_1$, $predFlagL0B_1$, $predFlagL1B_1$, $mvL0B_1$ and $mvL1B_1$ the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_1$, $yNbB_1$) as input, and the output is assigned to the coding block availability flag $availableB_1$.

— The variables $refIdxLXB_1$, $predFlagLXB_1$ and $mvLXB_1$ are derived as follows for X being 0 or 1:

  — If $availableB_1$ is equal to FALSE, 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.

  — Otherwise, $availableFlagB_1$ is set equal to 1 and the following assignments are made:

$$mvLXB_1 = MvLX[\ xNbB_1\ ][\ yNbB_1\ ] \quad (465)$$

$$refIdxLXB_1 = RefIdxLX[\ xNbB_1\ ][\ yNbB_1\ ] \quad (466)$$

$$predFlagLXB_1 = PredFlagLX[\ xNbB_1\ ][\ yNbB_1\ ] \quad (467)$$

— When neither of $refIdxL0B_1$ and $refIdxL1B_1$ is equal to −1 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

$$refIdxL1B_1 = -1 \quad (468)$$

$$predFlagL1B_1 = 0 \quad (469)$$

— When $availableB_1$ is equal to TRUE, the variable mergeCandList[numCurrMergeCand++] is updated with $mvLXB_1$, $refIdxLXB_1$ and $predFlagLXB_1$ for X being 0 or 1.

— When $availableB_1$ is equal to TRUE and numCurrMergeCand is greater than 1, the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

For the derivation of $refIdxL0B_0$, $refIdxL1B_0$, $predFlagL0B_0$, $predFlagL1B_0$, $mvL0B_0$ and $mvL1B_0$ the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location $(xNbB_0, yNbB_0)$ as input, and the output is assigned to the coding block availability flag $availableB_0$.

— The variables $refIdxLXB_0$, $predFlagLXB_0$ and $mvLXB_0$ are derived as follows for X being 0 or 1:

  — If $availableB_0$ is equal to FALSE, 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.

  — Otherwise, the following assignments are made:

$$mvLXB_0 = MvLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (470)$$

$$refIdxLXB_0 = RefIdxLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (471)$$

$$predFlagLXB_0 = PredFlagLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (472)$$

— When neither of $refIdxL0B_0$ and $refIdxL1B_0$ is equal to −1 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

$$refIdxL1B_0 = -1 \quad (473)$$

$$predFlagL1B0 = 0 \quad (474)$$

— When $availableB_0$ is equal to TRUE, the variable mergeCandList[numCurrMergeCand++] is updated with $mvLXB_0$, $refIdxLXB_0$ and $predFlagLXB_0$ for X being 0 or 1.

— When $availableB_0$ is equal to TRUE and numCurrMergeCand is greater than 1, the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

When numCurrMergeCand < mLSize − 1, for the derivation of refIdxL0A_0, refIdxL1A_0, predFlagL0A_0, predFlagL1A_0, mvL0A_0 and mvL1A_0 the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location (xNbA_0, yNbA_0) as input, and the output is assigned to the coding block availability flag availableA_0.

— The variables refIdxLXA_0, predFlagLXA_0 and mvLXA_0 are derived as follows for X being 0 or 1:

  — If availableA_0 is equal to FALSE, 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.

  — Otherwise, the following assignments are made:

$$mvLXA_0 = MvLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (475)$$

$$refIdxLXA_0 = RefIdxLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (476)$$

$$predFlagLXA_0 = PredFlagLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (477)$$

— When neither of refIdxL0A_0 and refIdxL1A_0 is equal to −1 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

$$refIdxL1A_0 = -1 \quad (478)$$

$$predFlagL1A_0 = 0 \quad (479)$$

— When availableA_0 is equal to TRUE, the variable mergeCandList[numCurrMergeCand++] is updated with mvLXA_0, refIdxLXA_0 and predFlagLXA_0 for X being 0 or 1.

— When availableA_0 is equal to TRUE and numCurrMergeCand is greater than 1, the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

When numCurrMergeCand < mLSize − 1, for the derivation of refIdxL0B_2, refIdxL1B_2, predFlagL0B_2, predFlagL1B_2, mvL0B_2 and mvL1B_2, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location (xNbB_2, yNbB_2) as input, and the output is assigned to the coding block availability flag availableB_2.

— The variables refIdxLXB_2, predFlagLXB_2 and mvLXB_2 are derived as follows for X being 0 or 1:

  — If availableB_2 is equal to FALSE, 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.

  — Otherwise, the following assignments are made:

$$mvLXB_2 = MvLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (480)$$

$$refIdxLXB_2 = RefIdxLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (481)$$

$$predFlagLXB_2 = PredFlagLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (482)$$

— When neither of refIdxL0B_2 and refIdxL1B_2 is equal to −1 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

$$refIdxL1B_2 = -1 \quad (483)$$

$$predFlagL1B_2 = 0 \quad (484)$$

— When availableB_2 is equal to TRUE, the variable mergeCandList[numCurrMergeCand++] is updated with mvLXB_2, refIdxLXB_2 and predFlagLXB_2 for X being 0 or 1.

— When availableB_2 is equal to TRUE and numCurrMergeCand is greater than 1, the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

8.5.2.3.3 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 merging candidate list mergeCandList, and

— a number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

— the merging candidate list mergeCandList, and

— the number of elements numCurrMergeCand within mergeCandList.

The variable currCb specifies the current luma coding block at luma location (xCb, yCb).

The variable refIdxLXCol, with X being equal to 0 or 1, is set equal to 0.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in subclause 6.4.2 with the luma location (xCb, yCb) and the luma coding block width nCbW as inputs.

Input merging candidate list mergeCandList is modified as the following ordered steps:

1) The central collocated motion vector is derived as follows:

$$xColCtr = xCb + (nCbW >> 1) \quad (485)$$

$$yColCtr = yCb + (nCbH >> 1) \quad (486)$$

— The luma location (xColCb, yColCb) is set equal to ((xColCtr >> 3) << 3, (yColCtr >> 3) << 3).

— The derivation process for collocated motion vectors as specified in subclause 8.5.2.3.4 is invoked with luma location coordinates (xColCb, yColCb) as input, and the output is assigned to mvLXCol, with X being equal to 0 or 1, and availableFlagCol.

— When availableFlagCol is equal to 3 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

predFlagL1Col = 0 (487)

refIdxL1Col = –1 (488)

— When availableFlagCol is not equal to 0, the variable mergeCandList[numCurrMergeCand++] is updated with mvLXCol, refIdxLXCol and predFlagLXCol, for X being equal to 0 or 1.

— When availableFlagCol is not equal to 0 and numCurrMergeCand is greater than 1, variable origNumCurrMergeCand is set equal to numCurrMergeCand and the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

— When numCurrMergeCand is less than origNumCurrMergeCand, availableFlagCol is set equal to 0.

2) When availableFlagCol is equal to 0, the bottom collocated motion vector is derived as follows:

— If availLR is equal to LR_01, the following applies:

xColBot = xCb (489)

yColBot = yCb + nCbH (490)

— Otherwise, the following applies:

xColBot = xCb + nCbW – 1 (491)

yColBot = yCb + nCbH (492)

— If yColBot is less than pic_height_in_luma_samples, and yCb >> CtbLog2SizeY is equal to yColBot >> CtbLog2SizeY, the following applies:

— The luma location (xColCb, yColCb) is set equal to ((xColBot >> 3) << 3, (yColBot >> 3) << 3).

— The derivation process for collocated motion vectors as specified in subclause 8.5.2.3.4 is invoked with luma location coordinates (xColCb, yColCb) as input, and the output is assigned to mvLXCol, with X being equal to 0 or 1, and availableFlagCol.

— Otherwise, both components of mvLXCol, with X being equal to 0 or 1, are set equal to 0, and availableFlagCol is set equal to 0.

— When availableFlagCol is equal to 3 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

predFlagL1Col = 0 (493)

refIdxL1Col = –1 (494)

— When availableFlagCol is not equal to 0, the variable mergeCandList[numCurrMergeCand++] is updated with mvLXCol, refIdxLXCol and predFlagLXCol, for X being equal to 0 or 1.

— When availableFlagCol is not equal to 0 and numCurrMergeCand is greater than 1, variable origNumCurrMergeCand is set equal to numCurrMergeCand and the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

— When numCurrMergeCand is less than origNumCurrMergeCand, availableFlagCol is set equal to 0.

3) When availableFlagCol is equal to 0, the side collocated motion vector is derived as follows:

— If availLR is equal to LR_01, the following applies:

$$xColSide = xCb - 1 \tag{495}$$

$$yColSide = yCb + nCbH - 1 \tag{496}$$

— Otherwise, the following applies:

$$xColSide = xCb + nCbW \tag{497}$$

$$yColSide = yCb + nCbH - 1 \tag{498}$$

— If xColSide larger than 0, and xColSide is less than pic_width_in_luma_samples, the following applies:

— The luma location coordinates (xColCb, yColCb) is set equal to ((xColSide >> 3) << 3, (yColSide >> 3) << 3).

— The derivation process for collocated motion vectors as specified in subclause 8.5.2.3.4 is invoked with luma location coordinates (xColCb, yColCb) as input, and the output is assigned to mvLXCol, with X being equal to 0 or 1, and availableFlagCol.

— Otherwise, both components of mvLXCol, with X being equal to 0 or 1, are set equal to 0 and availableFlagCol is set equal to 0.

— When availableFlagCol is equal to 3 and sum of the nCbW and nCbH is less than or equal to 12, and the following assignments are made:

$$predFlagL1Col = 0 \tag{499}$$

$$refIdxL1Col = -1 \tag{500}$$

— When availableFlagCol is not equal to 0, the variable mergeCandList[numCurrMergeCand++] is updated with mvLXCol, refIdxLXCol and predFlagLXCol for X being equal to 0 or 1.

— When availableFlagCol is not equal to 0 and numCurrMergeCand is greater than 1, variable origNumCurrMergeCand is set equal to numCurrMergeCand and the derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList, and the number of elements numCurrMergeCand within mergeCandList as inputs, and the number of elements numCurrMergeCand within mergeCandList as output.

— When numCurrMergeCand is less than origNumCurrMergeCand, availableFlagCol is set equal to 0.

8.5.2.3.4 Derivation process for collocated motion vectors

Input to this process is a luma location coordinates (xColCb, yColCb) in the collocated picture specified by ColPic.

Outputs of this process are:

— the motion vector prediction mvpLXCol in 1/4 fractional-sample accuracy, with X being equal to 0 or 1, and

— 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], MvDmvrL0[x][y] and RefIdxL0[x][y], respectively, of the collocated picture specified by ColPic, and the arrays predFlagL1Col[x][y], mvL1Col[x][y] and refIdxL1Col[x][y] are set equal to PredFlagL1[x][y], MvDmvrL1[x][y] and RefIdxL1[x][y], respectively, of the collocated picture specified by ColPic.

The variables mvCol and availableFlagLXCol, for X being equal to 0 or 1, are derived as follows:

— If refIdxLXCol[xColCb][yColCb] is invalid, both components of mvpLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

— Otherwise, the motion vector mvCol[colX], the reference index refIdxCol[colX] and the reference list identifier listCol[colX], with colX being equal to 0 or 1, are derived as follows:

  — The variable refIdxCol[colX], with colX being equal to 0 or 1, is set equal to invalid.

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

    — If predFlagL0Col[x][y] is equal to 0, and predFlagL1Col[x][y] is equal to 1, mvCol[colX], refIdxCol[colX] and listCol[colX] are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] with X being equal to 1 and colX being equal to 1, respectively.

    — Otherwise, if predFlagL0Col[x][y] is equal to 1, and predFlagL1Col[x][y] is equal to 0, mvCol[colX], refIdxCol[colX] and listCol[colX] are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] with X being equal to 0 and colX being equal to 0, respectively.

    — Otherwise (predFlagL0Col[x][y] is equal to 1, and predFlagL1Col[x][y] is equal to 1), mvCol[colX], refIdxCol[colX] and listCol[colX] are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] with X being 0 or 1 and colX being equal to 0 or 1, respectively.

  — Otherwise (temporal_mvp_assigned_flag is equal to 1), the following applies:

    — When predFlagLXCol[x][y] with X equal to col_pic_list_idx is equal to 1, mvCol[colX], refIdxCol[colX] and listCol[colX] are derived as follows:

      — mvCol[colX], refIdxCol[colX] and listCol[colX] set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] with X being equal to col_source_mvp_list_idx and colX being equal to 0, respectively.

— mvCol[colX], refIdxCol[colX] and listCol[colX] set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] with X being equal to col_source_mvp_list_idx, and colX being equal to 1.

— The variables mvpLXCol and availableFlagLXCol are derived as follows:

— The variable refPicOfColPic[X] is set equal to the picture with reference index refIdxCol[X] in the reference picture list listCol[X] of the slice containing luma location coordinates (xColCb, yColCb) in the collocated picture specified by ColPic, and the following applies:

— The POC distance (denoted as currPocDiffX) between current picture and current picture's reference picture specified by RefPicListX[refIdxLX] with refIdxLX being equal to 0, is computed as follows:

currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) (501)
, with X being equal to 0 or 1, respectively

— The POC distance (denoted as colPocDiffLX) between the collocated picture ColPic and refPicOfColPic[X] is computed as follows:

colPocDiffLX = DiffPicOrderCnt(ColPic, refPicOfColPic[X]) (502)
, with X being equal to 0 or 1, respectively

— If refIdxCol[X] is valid and colPocDiffLX is not equal to 0, the variable availableFlagLXCol is set equal to 1, the mvpLXCol is derived as a scaled version of the motion vector mvCol[X] as follows:

distScaleFactorLX = (currPocDiffLX << 5) / colPocDiffLX (503)

mvpLXCol = Clip3(−32768, 32767, Sign(distScaleFactorLX * mvCol[X]) *
((Abs(distScaleFactorLX * mvCol[X]) + 16) >> 5)) (504)
, with X being equal to 0 or 1, respectively

— Otherwise (invalid refIdxCol[X] or colPocDiffLX is equal to 0), the variable availableFlagLXCol is set equal to 0 and mvpLXCol is derived as follows:

mvpLXCol = 0 (505)
, 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 mvpLXCol is invoked as specified in subclause 8.5.2.3.5 with mvpLXCol and (xCb, yCb) as inputs.

The availableFlagCol is derived as follows:

— If availableFlagLXCol for X being 0 and 1 are both equal to 1, availableFlagCol is set equal to 3.

— Otherwise, if availableFlagLXCol for X being equal to 0 is equal to 1and availableFlagLXCol for X being 1 is equal to 0, availableFlagCol is set equal to 1.

— Otherwise, if availableFlagLXCol for X being equal to 1 is equal to 1 and availableFlagLXCol for X being 0 is equal to 0, availableFlagCol is set equal to 2.

— Otherwise, availableFlagCol is set equal to 0.

8.5.2.3.5 Derivation process for constrained scaled motion

Inputs to this process are:

— the motion vector prediction mvLXCol, and

— a luma location (xCb, yCb) specifying the top-left sample of the luma coding block.

Output of this process is the motion vector prediction mvLXCol.

The variable mvLXCol for X being set equal to 0 or 1 is derived as follows:

— The variable picPaddingSize is equal to 144.

— The variables paddedWidth and paddedHeight are derived as follows:

paddedWidth = pic_width_in_luma_samples + picPaddingSize (506)

paddedHeight = pic_height_in_luma_samples + picPaddingSize (507)

mvLXCol[0] = (xCb + mvLXCol[0]) < −picPaddingSize ? −(xCb + picPaddingSize) : mvLXCol[0] (508)

mvLXCol[1] = (yCb + mvLXCol[1]) < −picPaddingSize ? −(yCb + picPaddingSize) : mvLXCol[1] (509)

mvLXCol[0] = (xCb + mvLXCol[0]) > paddedWidth ? (paddedWidth − xCb) : mvLXCol[0] (510)

mvLXCol[1] = (yCb + mvLXCol[1]) > paddedHeight ? (paddedHeight − yCb) : mvLXCol[1] (511)

8.5.2.3.6 Derivation process for history-based merging candidates

Inputs to this process are:

— a merge candidate list mergeCandList,

— a number of available merging candidates in the list numCurrMergeCand,

— a maximal number of elements mLSize within mergeCandList, and

— two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Outputs to this process are:

— the modified merging candidate list mergeCandList, and

— the modified number of merging candidates in the list numCurrMergeCand.

The variable numOrigMergeCand is set equal to numCurrMergeCand. The variable hmvpStop is set equal to FALSE.

The variable maxNumCheckedHistory is set equal to (((NumHmvpCand + 1) >> 2) << 2) − 1.

For each candidate in HmvpCandList with hMvpIdx = 3..Min(maxNumCheckedHistory, (mLSize = = 4) ? 15 : 23), the following ordered steps are repeated until hmvpStop is equal to TRUE.

1) The variable hMvpCand is derived as follows:

— hMvpCand is set equal to HmvpCandList[NumHmvpCand – hMvpIdx].

— When refIdxL0 and refIdxL1 of hMvpCand are both valid and sum of the nCbW and nCbH is less than or equal to 12, then refIdxL1 of hMvpCand is set equal to –1.

2) The variable mergeCandList[numCurrMergeCand++] is updated with hMvpCand.

3) When numCurrMergeCand is greater than 1, derivation process for redundancy check as specified in subclause 8.5.2.3.10 is invoked with a merging candidate list mergeCandList and the number of elements numCurrMergeCand within mergeCandList as inputs, and with updated merging candidate list mergeCandList and the number of elements numCurrMergeCand within mergeCandList as outputs.

4) When numCurrMergeCand is equal to mLSize, hmvpStop is set equal to TRUE.

5) hMvpIdx is increased by 4.

8.5.2.3.7 Derivation process for combined bi-predictive merging candidates

Inputs to this process are:

— a merging candidate list mergeCandList,

— a number of elements numCurrMergeCand within mergeCandList, and

— a maximal number of elements mLSize within mergeCandList.

Outputs of this process are:

— the merging candidate list mergeCandList, and

— the number of elements numCurrMergeCand within mergeCandList.

The arrays refIdxL0 and refIdxL1, each size of numCurrMergeCand, are composed of 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 of 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 21.

2) The following assignments are made:

$$refIdxL0l0Cand = refIdxL0[\ l0CandIdx\ ] \tag{512}$$

$$refIdxL1l1Cand = refIdxL1[\ l1CandIdx\ ] \tag{513}$$

$$mvL0l0Cand = mvL0[\ l0CandIdx\ ] \tag{514}$$

$$mvL1l1Cand = mvL1[\ l1CandIdx\ ] \tag{515}$$

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:

$$refIdxL0combCand_k = refIdxL0l0Cand \quad (516)$$

$$refIdxL1combCand_k = refIdxL1l1Cand \quad (517)$$

$$mvL0combCand_k[0] = mvL0l0Cand[0] \quad (518)$$

$$mvL0combCand_k[1] = mvL0l0Cand[1] \quad (519)$$

$$mvL1combCand_k[0] = mvL1l1Cand[0] \quad (520)$$

$$mvL1combCand_k[1] = mvL1l1Cand[1] \quad (521)$$

$$numCurrMergeCand = numCurrMergeCand + 1 \quad (522)$$

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 21 — Specification of l0CandIdx and l1CandIdx

combIdx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
l0CandIdx	0	1	0	2	1	2	0	3	1	3
l1CandIdx	1	0	2	0	2	1	3	0	3	1
combIdx	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
l0CandIdx	2	3	0	4	1	4	2	4	3	4
l1CandIdx	3	2	4	0	4	1	4	2	4	3

8.5.2.3.8 Derivation process for zero motion vector merging candidates

Inputs to this process are:

— two variables nCbW and nCbH specifying the width and the height of the current luma coding block,

— a merging candidate list mergeCandList,

— a number of elements numCurrMergeCand within mergeCandList, and

— a maximal number of elements mLSize within mergeCandList.

Outputs of this process are:

— the merging candidate list mergeCandList, and

— the number of elements numCurrMergeCand within mergeCandList.

Candidate zeroCand consists of the luma motion vectors mvL0zeroCand and mvL1zeroCand, the reference indices refIdxL0zeroCand and refIdxL1zeroCand and the availability flags predFlagL0zeroCand and predFlagL1zeroCand.

A variable biPredAllowed is set equal to FALSE if slice_type is equal to P, or slice_type is equal to B and sum of the nCbW and nCbH is less than or equal to 12, otherwise biPredAllowed is set equal to TRUE.

While numCurrMergeCand is less than mLSize, the following applies:

— Motion vectors, reference indices and availability flags of candidate zeroCand are derived as follows:

mvL0zeroCand[0] = 0 (523)

mvL0zeroCand[1] = 0 (524)

mvL1zeroCand[0] = 0 (525)

mvL1zeroCand[1] = 0 (526)

refIdxL0zeroCand = 0 (527)

refIdxL1zeroCand = (biPredAllowed ? 0 : −1) (528)

predFlagL0zeroCand = 1 (529)

predFlagL1zeroCand = (biPredAllowed ? 1 : 0) (530)

— mergeCandList[numCurrMergeCand++] is set equal to zeroCand.

8.5.2.3.9 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[0][0] and mvL1[0][0],

— reference indices refIdxL0 and refIdxL1, and

— prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0].

Outputs of this process are:

— the MMVD motion vector mMvL0 and mMvL1,

— the modified reference indices refIdxL0 and refIdxL1, and

— the modified prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0].

The variable currPic specifies the current picture.

The MMVD motion vectors mMvL0 and mMvL1 are derived, and refIdxL0, refIdxL1, predFlagL0[0][0] and predFlagL1[0][0] are updated as follows:

— The MMVD motion vectors are set as follows:

mMvL0 = mvL0[0][0] (531)

mMvL1 = mvL1[0][0] (532)

— If the value of mmvd_group_idx is equal to 1, the following applies:

— If both predFlagL0[0][0] and predFlagL1[0][0] are equal to 1, the following applies:

refIdxL1 = −1 (533)

predFlagL1[0][0] = 0 (534)

mMvL1[0] = 0 (535)

mMvL1[1] = 0 (536)

— Otherwise, if slice_type is equal to B and predFlagL0[0][0] is equal to 1, the following applies:

predFlagL1[0][0] = 1 (537)

— If NumRefIdxActive[1] is greater than 1 and DiffPicOrderCnt(RefPicList1[1], currPic) is equal to DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]), the following applies:

refIdxL1 = 1 (538)

— Otherwise, the following applies:

refIdxL1= 0 (539)

— mMvL1 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (540)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (541)

distScaleFactor = (currPocDiffL1 << 5) / currPocDiffL0 (542)

mMvL1[0] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[0]) *
((Abs(distScaleFactor * mMvL0[0]) + 16) >> 5)) (543)

mMvL1[1] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[1]) *
((Abs(distScaleFactor * mMvL0[1]) + 16) >> 5)) (544)

— Otherwise, if slice_type is equal to P, the following applies:

— If NumRefIdxActive[0] is equal to 1, the following applies:

targetRefIdxL0 = refIdxL0 (545)

— Otherwise, the following applies:

targetRefIdxL0 = !refIdxL0 (546)

— mMvL0 is set as follows:

— If targetRefIdxL0 is equal to refIdxL0, the following applies:

mMvL0[0] = mMvL0[0] + 3 (547)

mMvL0[1] = mMvL0[1] (548)

— Otherwise, the following applies:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (549)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList0[targetRefIdxL0]) (550)

distScaleFactor = (currPocDiffL1 << 5) / currPocDiffL0 (551)

mMvL0[0] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[0]) *
((Abs(distScaleFactor * mMvL0[0]) + 16) >> 5)) (552)

mMvL0[1] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[1]) *
((Abs(distScaleFactor * mMvL0[1]) + 16) >> 5)) (553)

— refIdxL0 is set equal to targetRefIdxL0.

— Otherwise, if predFlagL1[0][0] are equal to 1, the following applies:

predFlagL0[0][0] = 1 (554)

— If NumRefIdxActive[0] is greater than 1 and DiffPicOrderCnt(RefList0[1], currPic) is equal to DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]), the following applies:

refIdxL0 = 1 (555)

— Otherwise, the following applies:

refIdxL0 = 0 (556)

— mMvL0 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (557)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (558)

distScaleFactor = (currPocDiffL0 << 5) / currPocDiffL1 (559)

mMvL0[0] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL1[0]) *
((Abs(distScaleFactor * mMvL1[0]) + 16) >> 5)) (560)

mMvL0[1] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL1[1]) *
((Abs(distScaleFactor * mMvL1[1]) + 16) >> 5)) (561)

— Otherwise, if the value of mmvd_group_idx is equal to 2, the following applies:

— If both predFlagL0[0][0] and predFlagL1[0][0] are equal to 1, the following applies:

refIdxL0 = −1 (562)

predFlagL0[0][0] = 0 (563)

mMvL0[0] = 0 (564)

mMvL0[1] = 0 (565)

— Otherwise, if slice_type is equal to B and predFlagL0[0][0] is equal to 1, the following applies:

predFlagL1[0][0] = 1 (566)

— If NumRefIdxActive[1] is greater than 1 and DiffPicOrderCnt(RefPicList[1], currPic) is equal to DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]), the following applies:

refIdxL1 = 1 (567)

— Otherwise, the following applies:

refIdxL1 = 0 (568)

— mMvL1 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (569)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (570)

distScaleFactor = (currPocDiffL1 << 5) / currPocDiffL0 (571)

mMvL1[0] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[0]) *
((Abs(distScaleFactor * mMvL0[0]) + 16) >> 5)) (572)

mMvL1[1] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[1]) *
((Abs(distScaleFactor * mMvL0[1]) + 16) >> 5)) (573)

— refIdxL0 is set equal to –1, and predFlagL0[0][0], mMvL0[0] and mMvL0[1] are set equal to 0.

— Otherwise, if slice_type is equal to P, the following applies:

— If NumRefIdxActive[0] is less than 3, the following applies:

targetRefIdxL0 = refIdxL0 (574)

— Otherwise, the following applies:

targetRefIdxL0 = refIdxL0 < 2 ? 2 : 1 (575)

— mMvL0 is set as follows:

— If targetRefIdxL0 is equal to refIdxL0, the following applies:

mMvL0[0] = mMvL0[0] − 3 (576)

mMvL0[1] = mMvL0[1] (577)

— Otherwise, the following applies:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (578)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList0[targetRefIdxL0]) (579)

distScaleFactor = (currPocDiffL1 << 5) / currPocDiffL0 (580)

mMvL0[0] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[0]) *
((Abs(distScaleFactor * mMvL0[0]) + 16) >> 5)) (581)

mMvL0[1] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL0[1]) *
((Abs(distScaleFactor * mMvL0[1]) + 16) >> 5)) (582)

— refIdxL0 is set equal to targetRefIdxL0.

— Otherwise, if predFlagL1[0][0] is equal to 1, the following applies:

predFlagL0[0][0] = 1 (583)

— If NumRefIdxActive[0] is greater than 1 and DiffPicOrderCnt(RefPicList0[1], currPic) is equal to DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]), the following applies:

refIdxL0 = 1 (584)

— Otherwise, the following applies:

refIdxL0 = 0 (585)

— mMvL0 is set as follows:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (586)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (587)

distScaleFactor = (currPocDiffL0 << 5) / currPocDiffL1 (588)

mMvL0[0] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL1[0]) *
((Abs(distScaleFactor * mMvL1[0]) + 16) >> 5)) (589)

mMvL0[1] = Clip3(−32768, 32767, Sign(distScaleFactor * mMvL1[1]) *
((Abs(distScaleFactor * mMvL1[1]) + 16) >> 5)) (590)

— refIdxL1 is set equal to –1, and predFlagL1[0][0], mMvL1[0] and mMvL1[1] are set equal to 0.

— If both predFlagL0[0][0] and predFlagL1[0][0] are equal to 1, the following applies:

currPocDiffL0 = DiffPicOrderCnt(currPic, RefPicList0[refIdxL0]) (591)

currPocDiffL1 = DiffPicOrderCnt(currPic, RefPicList1[refIdxL1]) (592)

— If Abs(currPocDiffL0) is equal to Abs(currPocDiffL1), the following applies:

mMvdL0[0] = MmvdOffset[0] (593)

mMvdL0[1] = MmvdOffset[1] (594)

mMvdL1[0] = MmvdOffset[0] (595)

mMvdL1[1] = MmvdOffset[1] (596)

— Otherwise, if Abs(currPocDiffL0) is greater than Abs(currPocDiffL1), the following applies:

mMvdL1[0] = MmvdOffset[0] (597)

mMvdL1[1] = MmvdOffset[1] (598)

distScaleFactor = (Abs(currPocDiffL1) << 5) / Abs(currPocDiffL0) (599)

mMvdL0[0] = Clip3(−32768, 32767, (distScaleFactor * mMvdL1[0] + 16) >> 5) (600)

mMvdL0[1] = Clip3(−32768, 32767 (distScaleFactor * mMvdL1[1] + 16) >> 5) (601)

— Otherwise, if Abs(currPocDiffL0) is less than Abs(currPocDiffL1), the following applies:

mMvdL0[0] = MmvdOffset[0] (602)

mMvdL0[1] = MmvdOffset[1] (603)

distScaleFactor = (Abs(currPocDiffL0) << 5) / Abs(currPocDiffL1) (604)

mMvdL1[0] = Clip3(−32768, 32767, (distScaleFactor * mMvdL0[0] + 16) >> 5) (605)

mMvdL1[1] = Clip3(−32768, 32767, (distScaleFactor * mMvdL0[1] + 16) >> 5) (606)

— If currPocDiffL0 * currPocDiffL1 is smaller than 0, the following applies:

mMvdL0[0] = mMvdL0[0] (607)

mMvdL0[1] = mMvdL0[1] (608)

mMvdL1[0] = −mMvdL1[0] (609)

mMvdL1[1] = −mMvdL1[1] (610)

— Otherwise (predFlagL0[0][0] or predFlagL1[0][0] are equal to 1), the following applies for X being 0 and 1:

mMvdLX[0] = (predFlagLX[0][0] = = 1) ? MmvdOffset[0] : 0 (611)

mMvdLX[1] = (predFlagLX[0][0] = = 1) ? MmvdOffset[1] : 0 (612)

— The MMVD motion vectors are updated as follows:

mMvL0[0] += mMvdL0[0] (613)

mMvL0[1] += mMvdL0[1] (614)

mMvL1[0] += mMvdL1[0] (615)

mMvL1[1] += mMvdL1[1] (616)

8.5.2.3.10 Derivation process for motion vector prediction redundancy check

Inputs to this process are:

— a merging candidate list mergeCandList, and

— a number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

— the merging candidate list mergeCandList, and

— the number of elements numCurrMergeCand within mergeCandList.

When numCurrMergeCand is greater than 1, the variable candIsNew is set equal to TRUE, the variable candIndx is set equal to 0 and the following ordered steps are repeated until candIsNew is equal to FALSE or candIndx is equal to numCurrMergeCand – 2:

1) The variable candIsNew is set equal to FALSE if mergeCandList[candIndx] and mergeCandList[numCurrMergeCand – 1] have all the following conditions met:

   — number of available reference lists for compared entries are same.

   — entries have same available reference list indices (e.g. list 0, list 1 or list 0 and list 1).

   — entries have same valid reference indices in corresponding references lists.

   — entries have same motion vectors corresponding to available reference lists.

2) If candIsNew is equal to TRUE, the variable candIndx is incremented by 1.

When candIsNew is equal to FALSE, the variable numCurrMergeCand is decremented by 1.

8.5.2.4 Derivation process for luma motion vector prediction

8.5.2.4.1 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,

— two variables nCbW and nCbH specifying the width and the height of the current luma coding block, and

— 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.

If sps_admvp_flag is equal to 0, the following applies:

The motion vector predictor mvpLX with X being 0 or 1 is derived in the following ordered steps:

1) The variable mvpLX is set as follows:

mvpLX[0] = 0 (617)

mvpLX[1] = 0 (618)

2) When refIdxLX is not equal to −1, the derivation process for motion vector predictor from neighbouring coding unit partitions as specified in subclause 8.5.2.4.3 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_lX[xCb][yCb] and the reference list identifier listX set equal to X, with X being 0 or 1, as inputs, and the output being the motion vector predictor mvPred. The variable mvpLX is set equal to mvPred.

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

A variable mvpAvailFlag is set equal to 0.

The variable availLR is derived by invoking the derivation process for left and right neighbouring blocks availabilities as specified in subclause 6.4.2 with the luma location (xCb, yCb) and the luma coding block width nCbW as inputs.

If 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 +nCbH − 1).

— The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW, yCb +nCbH − 1).

— 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, if 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, if 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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_1$, $yNbA_1$) as input, and the output is assigned to the coding block availability flag $availableA_1$.

— When $availableA_1$ is equal to TRUE and RefIdxLX[$xNbA_1$][$yNbA_1$] is not equal to −1, the following applies:

  — mvpAvailFlag is set equal to 1.

  — mvpLX is set equal to MvLX[$xNbA_1$][$yNbA_1$].

  — When refIdxLX is not equal to RefIdxLX[$xNbA_1$][$yNbA_1$], mvpLX is derived as follows:

    — The POC distance (denoted as currPocDiffX) between current picture and current picture's reference picture list RefPicListX[RefIdxLX[$xNbA_1$][$yNbA_1$]], is computed as follows:

$$currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[xNbA_1][yNbA_1]]) \quad (619)$$

    — The POC distance (denoted as targetPocDiffLX) between current picture and the list X target reference picture of current picture is computed as follows:

$$targetPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) \quad (620)$$

    — mvpLX is derived as a scaled version of the motion vector as follows:

$$distScaleFactorLX = (targetPocDiffLX \ll 5) / currPocDiffLX \quad (621)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactorLX * mvpLX) * ((Abs(distScaleFactorLX * mvpLX) + 16) \gg 5)) \quad (622)$$

Otherwise, if amvr_idx[x0][y0] is equal to 1, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_1$, $yNbB_1$) as input, and the output is assigned to the coding block availability flag $availableB_1$.

— When $availableB_1$ is equal to TRUE and RefIdxLX[$xNbB_1$][$yNbB_1$] is not equal to −1, the following applies:

— mvpAvailFlag is set equal to 1.

— mvpLX is set equal to MvLX[$xNbB_1$][$yNbB_1$].

— When refIdxLX is not equal to RefIdxLX[$xNbB_1$][$yNbB_1$], mvpLX is derived as follows:

— The POC distance (denoted as currPocDiffX) between current picture and current picture's reference picture list RefPicListX[RefIdxLX[$xNbB_1$][$yNbB_1$]], is computed as follows:

currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[$xNbB_1$][$yNbB_1$]]) (623)

— The POC distance (denoted as targetPocDiffLX) between current picture and the list X target reference picture of current picture is computed as follows:

targetPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) (624)

— mvpLX is derived as a scaled version of the motion vector as follows:

distScaleFactorLX = (targetPocDiffLX << 5) / currPocDiffLX (625)

mvpLX = Clip3(−32768, 32767, Sign(distScaleFactorLX * mvpLX) *
((Abs(distScaleFactorLX * mvpLX) + 16) >> 5)) (626)

Otherwise, if amvr_idx[x0][y0] is equal to 2, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_0$, $yNbB_0$) as input, and the output is assigned to the coding block availability flag $availableB_0$.

— When $availableB_0$ is equal to TRUE and RefIdxLX[$xNbB_0$][$yNbB_0$] is not equal to −1, the following applies:

— mvpAvailFlag is set equal to 1.

— mvpLX is set equal to MvLX[$xNbB_0$][$yNbB_0$].

— When refIdxLX is not equal to RefIdxLX[$xNbB_0$][$yNbB_0$], mvpLX is derived as follows:

— The POC distance (denoted as currPocDiffX) between current picture and current picture's reference picture list RefPicListX[RefIdxLX[$xNbB_0$][$yNbB_0$]], is computed as follows:

currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[$xNbB_0$][$yNbB_0$]]) (627)

— The POC distance (denoted as targetPocDiffLX) between current picture and the list X target reference picture of current picture is computed as follows:

targetPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) (628)

— mvpLX is derived as a scaled version of the motion vector as follows:

distScaleFactorLX = (targetPocDiffLX << 5) / currPocDiffLX (629)

mvpLX = Clip3(−32768, 32767, Sign(distScaleFactorLX * mvpLX) *
((Abs(distScaleFactorLX * mvpLX) + 16) >> 5)) (630)

Otherwise, if amvr_idx[x0][y0] is equal to 3, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_0$, $yNbA_0$) as input, and the output is assigned to the coding block availability flag $availableA_0$.

— When $availableA_0$ is equal to TRUE and $RefIdxLX[xNbA_0][yNbA_0]$ is not equal to –1, the following applies:

  — mvpAvailFlag is set equal to 1.

  — mvpLX is set equal to $MvLX[xNbA_0][yNbA_0]$.

  — When refIdxLX is not equal to $RefIdxLX[xNbA_0][yNbA_0]$, mvpLX is derived as follows:

    — The POC distance (denoted as currPocDiffX) between current picture and current picture's reference picture list $RefPicListX[RefIdxLX[xNbA_0][yNbA_0]]$, is computed as follows:

$$currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[xNbA_0][yNbA_0]]) \quad (631)$$

    — The POC distance (denoted as targetPocDiffLX) between current picture and the list X target reference picture of current picture is computed as follows:

$$targetPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) \quad (632)$$

    — mvpLX is derived as a scaled version of the motion vector as follows:

$$distScaleFactorLX = (targetPocDiffLX << 5) / currPocDiffLX \quad (633)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactorLX * mvpLX) * ((Abs(distScaleFactorLX * mvpLX) + 16) >> 5)) \quad (634)$$

Otherwise, if amvr_idx[x0][y0] is equal to 4, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_2$, $yNbB_2$) as input, and the output is assigned to the coding block availability flag $availableB_2$.

— When $availableB_2$ is equal to TRUE and $RefIdxLX[xNbB_2][yNbB_2]$ is not equal to –1, the following applies:

  — mvpAvailFlag is set equal to 1.

  — mvpLX is set equal to $MvLX[xNbB_2][yNbB_2]$.

  — When refIdxLX is not equal to $RefIdxLX[xNbB_2][yNbB_2]$, mvpLX is derived as follows:

    — The POC distance (denoted as currPocDiffX) between current picture and current picture's reference picture list $RefPicListX[RefIdxLX[xNbB_2][yNbB_2]]$, is computed as follows:

$$currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[RefIdxLX[xNbB_2][yNbB_2]]) \quad (635)$$

    — The POC distance (denoted as targetPocDiffLX) between current picture and the list X target reference picture of current picture is computed as follows:

$$targetPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) \quad (636)$$

    — mvpLX is derived as a scaled version of the motion vector as follows:

$$distScaleFactorLX = (targetPocDiffLX << 5) / currPocDiffLX \quad (637)$$

$$mvpLX = Clip3(-32768, 32767, Sign(distScaleFactorLX * mvpLX) * ((Abs(distScaleFactorLX * mvpLX) + 16) >> 5)) \quad (638)$$

If mvpAvailFlag is equal to 0, the following applies:

— The derivation process for default reference index as specified in subclause 8.5.2.4.5.2 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 subclause 8.5.2.4.2 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.

— When refIdxLX is not equal to DefaultRefIdxLX, DefaultMvLX is derived as follows:

  — The POC distance (denoted as currPocDiffX) between current picture and reference picture list RefPicListX[DefaultRefIdxLX], is computed as follows:

$$currPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[DefaultRefIdxLX]) \quad (639)$$

  — The POC distance (denoted as targetPocDiffLX) between current picture and the list X target reference picture of current picture is computed as follows:

$$targetPocDiffLX = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) \quad (640)$$

  — DefaultMvLX is derived as a scaled version of the motion vector as follows:

$$distScaleFactorLX = (targetPocDiffLX << 5) / currPocDiffLX \quad (641)$$

$$DefaultMvLX = Clip3(-32768, 32767, Sign(distScaleFactorLX * DefaultMvLX) * ((Abs(distScaleFactorLX * DefaultMvLX) + 16) >> 5)) \quad (642)$$

— The following applies:

$$mvpLX[0] = DefaultMvLX[0] \quad (643)$$

$$mvpLX[1] = DefaultMvLX[1] \quad (644)$$

When sps_amvr_flag is equal to 1 and amvr_idx[x0][y0] is not equal to 0, the following applies:

$$mvpLX[0] = mvpLX[0] >= 0 ? ((mvpLX[0] + (1 << (amvr_idx[x0][y0] - 1))) >> amvr_idx[x0][y0]) << amvr_idx[x0][y0] : -(((-mvpLX[0] + (1 << (amvr_idx[x0][y0] - 1))) >> amvr_idx[x0][y0]) << amvr_idx[x0][y0]) \quad (645)$$

$$mvpLX[1] = mvpLX[1] >= 0 ? ((mvpLX[1] + (1 << (amvr_idx[x0][y0] - 1))) >> amvr_idx[x0][y0]) << amvr_idx[x0][y0] : -(((-mvpLX[1] + (1 << (amvr_idx[x0][y0] - 1))) >> amvr_idx[x0][y0]) << amvr_idx[x0][y0]) \quad (646)$$

8.5.2.4.2 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, and

— 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 subclause 6.4.2 with the luma location (xCb, yCb) and the luma coding block width nCbW as inputs.

If 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 + nCbH − 1).

— The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW, yCb + nCbH − 1).

Otherwise, if 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, if 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 with X being 0 or 1, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_1$, $yNbA_1$) as input, and the output is assigned to the coding block availability flag $availableA_1$.

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_1$, $yNbB_1$) as input, and the output is assigned to the coding block availability flag $availableB_1$.

— Both components of DefaultMvLX are set equal to 0.

— If availableA_1 is equal to TRUE and refIdxLX is equal to RefIdxLX[$xNbA_1$][$yNbA_1$], DefaultMvLX is set equal to MvLX[$xNbA_1$][$yNbA_1$].

— Otherwise, if availableB_1 is equal to TRUE and refIdxLX is equal to RefIdxLX[$xNbB_1$][$yNbB_1$], DefaultMvLX is set equal to MvLX[$xNbB_1$][$yNbB_1$].

— Otherwise, if availableA_1 is equal to TRUE and RefIdxLX[$xNbA_1$][$yNbA_1$] is not equal to −1, DefaultMvLX is set equal to MvLX[$xNbA_1$][$yNbA_1$].

— Otherwise, if availableB_1 is equal to TRUE and RefIdxLX[$xNbB_1$][$yNbB_1$] is not equal to −1, DefaultMvLX is set equal to MvLX[$xNbB_1$][$yNbB_1$].

— Otherwise, if sps_hmvp_flag is equal to 1 and NumHmvpCand (number of entries in HmvpCandList) is greater than 0, the DefaultMvLX is set equal to the DefaultMvLX of the history motion vector candidates derived as specified in subclause 8.5.2.4.4 with refIdxLX as input.

8.5.2.4.3 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, and

— a reference list identifier listX.

Output of this process is the motion vector predictor mvPred.

For X being replaced by listX in the variable MvLX, the motion vector predictors list, mvpList, is derived as follows:

For the derivation of mvpList[0], the following applies:

— The luma location (xNb_0, yNb_0) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb).

— The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (xNb_0, yNb_0) as input, and the output is assigned to the coding block availability flag availableNb_0.

— If availableNb_0 is equal to TRUE, mvpList[0] is derived as follows:

$$mvpList[\ 0\ ] = MvLX[\ xNb_0\ ][\ yNb_0\ ] \tag{647}$$

— Otherwise (availableNb_0 is equal to FALSE), mvpList[0] is derived as follows:

$$mvpList[\ 0\ ][\ 0\ ] = 1 \tag{648}$$

$$mvpList[\ 0\ ][\ 1\ ] = 1 \tag{649}$$

For the derivation of mvpList[1], the following applies:

— The luma location (xNb_1, yNb_1) inside the neighbouring luma coding block is set equal to (xCb, yCb − 1).

— The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked the luma location (xNb_1, yNb_1) as input, and the output is assigned to the coding block availability flag $availableNb_1$.

— If $availableNb_1$ is equal to TRUE, mvpList[1] is derived as follows:

$$mvpList[1] = MvLX[xNb_1][yNb_1] \quad (650)$$

— Otherwise ($availableNb_1$ is equal to FALSE), mvpList[1] is derived as follows:

$$mvpList[1][0] = 1 \quad (651)$$

$$mvpList[1][1] = 1 \quad (652)$$

For the derivation of mvpList[2], the following applies:

— The luma location (xNb_2, yNb_2) inside the neighbouring luma coding block is set equal to (xCb + xCbW, yCb − 1).

— The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked the luma location (xNb_2, yNb_2) as input, and the output is assigned to the coding block availability flag $availableNb_2$.

— If $availableNb_2$ is equal to TRUE, mvpList[2] is derived as follows:

$$mvpList[2] = MvLX[xNb_2][yNb_2] \quad (653)$$

— Otherwise ($availableNb_2$ is equal to FALSE), mvpList[2] is derived as follows:

$$mvpList[2][0] = 1 \quad (654)$$

$$mvpList[2][1] = 1 \quad (655)$$

For the derivation of mvpList[3], the following applies:

— The variable TempPic is set equal to RefPicListX[0].

— The luma location (xNb_3, yNb_3) inside the neighbouring luma coding block is set equal to (xCb, yCb).

— The variable mvTemp specifies the refined motion vector of list 0 (MvDmvrL0) covering the luma location (xNb_3, yNb_3) inside the temporal picture specified by TempPic.

— mvpList[3] is derived as follows:

$$mvpList[3] = mvTemp \quad (656)$$

The variable mvPred is set equal to mvpList[mvpIdx].

8.5.2.4.4 Derivation process for history-based motion vector prediction candidates

Input to this process is 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, and

— the DefaultRefIdxLX, with X being 0 or 1.

A variable HmvpMVList is defined as the history based motion vector table derived from HmvpCandList. A variable HmvpRefList is defined as the reference indices table derived from HmvpCandList. A variable NumHmvpCand is set equal 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 the variable HMVPDerived is equal to TRUE:

1) hMvpIdx is set equal to 1, HMVPDerived is set equal to FALSE.

2) If curRefIdxLX is equal to HmvpRefList[NumHmvpCand – hMvpIdx][X], with X being 0 or 1, DefaultRefIdxLX is set equal to HmvpRefList[NumHmvpCand – hMvpIdx][X] and DefaultMvLX is set equal to NumHmvpCand[HMVPCandNum – hMvpIdx][X] and HMVPDerived is set equal to TRUE, otherwise hMvpIdx++.

3) If 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 equal to 1.

2) If HmvpRefList[NumHmvpCand – hMvpIdx][X] is valid, DefaultRefIdxLX is set equal to HmvpRefList[NumHmvpCand – hMvpIdx][X] and DefaultMvLX is set equal to HmvpMVList[NumHmvpCand – hMvpIdx][X] and HMVPDerived is set equal to TRUE, otherwise hMvpIdx++.

3) If HMVPDerived is equal to FALSE and hMvpIdx < Min(4, NumHmvpCand), repeat step 2.

If HMVPDerived is equal to FALSE, DefaultMvLX[0], and DefaultMvLX[1] and DefaultRefIdxLX are set equal to 0.

8.5.2.4.5 Derivation process for the luma reference index

8.5.2.4.5.1 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, and

— 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 subclause 6.4.2 with the luma location (xCb, yCb) and the luma coding block width nCbW as inputs.

If 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 +nCbH − 1).

— The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW, yCb + nCbH − 1).

— 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, if 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, if 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 subclause 8.5.2.4.5.2 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 equal 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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_1$, $yNbA_1$) as input, 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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_1$, $yNbB_1$) as input, 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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_0$, $yNbB_0$) as input, 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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_0$, $yNbA_0$) as input, 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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_2$, $yNbB_2$) as input, 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.

8.5.2.4.5.2 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, and

— 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 subclause 6.4.2 with the luma location (xCb, yCb) and the luma coding block width nCbW as inputs.

If 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 + nCbH – 1).

— The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb+ nCbW, yCb + nCbH – 1).

Otherwise, if 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, if 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 with X being 0 or 1, the following applies:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbA_1$, $yNbA_1$) as input, and the output is assigned to the coding block availability flag $availableA_1$.

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location ($xNbB_1$, $yNbB_1$) as input, and the output is assigned to the coding block availability flag $availableB_1$.

— DefaultRefIdxLX is set equal to 0.

— If $availableA_1$ is equal to TRUE and refIdxLX is equal to $RefIdxLX[xNbA_1][yNbA_1]$, DefaultRefIdxLX is set equal to refIdxLX.

— Otherwise, if $availableB_1$ is equal to TRUE and refIdxLX is equal to $RefIdxLX[xNbB_1][yNbB_1]$, DefaultRefIdxLX is set equal to refIdxLX.

— Otherwise, if $availableA_1$ is equal to TRUE and $RefIdxLX[xNbA_1][yNbA_1]$ is not equal to −1, DefaultRefIdxLX is set equal to $RefIdxLX[xNbA_1][yNbA_1]$.

— Otherwise, if $availableB_1$ is equal to TRUE and $RefIdxLX[xNbB_1][yNbB_1]$ is not equal to −1, DefaultRefIdxLX is set equal to $RefIdxLX[xNbB_1][yNbB_1]$.

— Otherwise, if sps_hmvp_flag is equal to 1 and NumHmvpCand (number of entries in HmvpCandList) is greater than 0, the DefaultRefIdxLX is set equal to the DefaultRefIdxLX of the history motion vector candidates derived as specified in subclause 8.5.2.4.4 with refIdxLX as input.

8.5.2.5 Derivation process for luma motion vectors for direct 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, and

— 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[0][0] and mvL1[0][0],

— the reference indices refIdxL0 and refIdxL1, and

— the prediction list utilization flags predFlagL0[0][0] and predFlagL1[0][0].

The variable currPic specifies the current picture.

For the derivation of the variables mvL0[0][0] and mvL1[0][0], refIdxL0 and refIdxL1, and predFlagL0[0][0] and predFlagL1[0][0], the following applies:

— The variables mvL0[0][0], mvL1[0][0], refIdxL0, refIdxL1, predFlagL0[0][0] and predFlagL1[0][0] are set as follows:

mvL0[0][0][0] = 0 (657)

mvL0[0][0][1] = 0 (658)

mvL1[0][0][0] = 0 (659)

mvL1[0][0][1] = 0 (660)

refIdxL0 = 0 (661)

refIdxL1 = 0 (662)

predFlagL0[0][0] = 1 (663)

predFlagL1[0][0] = 0 (664)

— The variable TempPic is set equal to RefPicList1[0].

— The motion vector mvTemp is set equal to the refined motion vector of list 0 (MvDmvrL0) covering the luma location (xCb + nCbW – 1, yCb + nCbH – 1) inside the temporal picture specified by TempPic.

— The variable refPicListTemp[0] is set equal to 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]) (665)

diffPocNormL0 = DiffPicOrderCnt(currPic, RefPicList0[0]) (666)

diffPocNormL1 = DiffPicOrderCnt(RefPicList1[0], currPic) (667)

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

mvL0[0][0][0] = 0 (668)

mvL0[0][0][1] = 0 (669)

mvL1[0][0][0] = 0 (670)

mvL1[0][0][1] = 0 (671)

— Otherwise, the following applies:

mvL0[0][0][0] = diffPocNormL0 * mvTemp[0] / diffPocDeNorm (672)

$$mvL0[\,0\,][\,0\,][\,1\,] = diffPocNormL0 * mvTemp[\,1\,] / diffPocDeNorm \quad (673)$$

$$mvL1[\,0\,][\,0\,][\,0\,] = -diffPocNormL1 * mvTemp[\,0\,] / diffPocDeNorm \quad (674)$$

$$mvL1[\,0\,][\,0\,][\,1\,] = -diffPocNormL1 * mvTemp[\,1\,] / diffPocDeNorm \quad (675)$$

8.5.2.6 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:

$$mvCLX[\,0\,] = mvLX[\,0\,] * 2 / SubWidthC \quad (676)$$

$$mvCLX[\,1\,] = mvLX[\,1\,] * 2 / SubHeightC \quad (677)$$

8.5.2.7 Updating process for the history-based motion vector predictor candidate list

Inputs to this process are:

— luma motion vectors in 1/4 fractional-sample accuracy mvL0 and mvL1, and

— reference indices refIdxL0 and refIdxL1.

HmvpCandList is a list of entries, each of those consists of the luma motion vectors mvL0candN and mvL1candN, the reference indices refIdxL0candN and refIdxL1candN. Number of entries in HmvpCandList is equal to NumHmvpCand. If NumHmvpCand is equal to zero, HmvpCandList is set equal to the empty list.

The MVP candidate hMvpCand is set equal to entry consisting of the luma motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1. If NumHmvpCand is smaller than 23, for each index hMvpIdx = NumHmvpCand..22, both components of motion vectors of HmvpCandList[hMvpIdx] mvL0 and mvL1 are set equal to 0 and reference indices of HmvpCandList[hMvpIdx] refIdxL0 and refIdxL1 is set equal to −1.

If slice_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.

8.5.2.8 Conversion luma motion vectors from 1/16 fractional-sample accuracy to 1/4 fractional-sample accuracy

Input to this process is the motion vector mv in 1/16 pel accuracy.

Output of the process is the motion vector mv in 1/4 pel accuracy.

For the conversion of the luma motion vector mv, the following applies:

$$mv[0] = mv[0] >> 2 \quad (678)$$

$$mv[1] = mv[1] >> 2 \quad (679)$$

8.5.2.9 Conversion luma motion vectors from 1/4 fractional-sample accuracy to 1/16 fractional-sample accuracy

Input to this process is the motion vector mv in 1/4 pel accuracy.

Output of the process is the motion vector mv in 1/16 pel accuracy.

For the conversion of the luma motion vector mv, the following applies:

$$mv[0] = mv[0] << 2 \quad (680)$$

$$mv[1] = mv[1] << 2 \quad (681)$$

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, and

— a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

— the reference indices refIdxLX with X being 0 and 1,

— the prediction list utilization flags predFlagLX with X being 0 and 1,

— the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

— the flag specifying motion vector clipping type clipMV,

— the luma subblock motion vector array in 1/16 fractional-sample accuracy mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 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, and

— 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], mvCL1[xSbIdx][ySbIdx], refIdxL0, refIdxL1, numSbX, numSbY, 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_mode_flag[xCb][yCb] is equal to 1, the derivation process for motion vectors and reference indices in affine merge mode as specified in subclause 8.5.3.2 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the variable availLR specifying left and right neighbouring blocks' availability of luma coding block as inputs, the number of control point motion vectors numCpMv, the luma affine control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as outputs.

  — Otherwise (merge_mode_flag[xCb][yCb] is equal to 0), for X being replaced by either 0 or 1 in the variables predFlagLX, cpMvLX, MvdCpLX, and refIdxLX, in PRED_LX, and in the syntax element ref_idx_lX, the following ordered steps apply:

    1) The number of control point motion vectors numCpMv is set equal to MotionModelIdc[xCb][yCb] + 1.

    2) The variables refIdxLX and predFlagLX are derived as follows:

      — If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI, the following applies:

        refIdxLX = ref_idx_lX[xCb][yCb] (682)

        predFlagLX[0][0] = 1 (683)

      — Otherwise, the variables refIdxLX and predFlagLX are specifiedas follows:

        refIdxLX = −1 (684)

        predFlagLX[0][0] = 0 (685)

    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] (686)

      mvdCpLX[cpIdx][1] = MvdCpLX[xCb][yCb][cpIdx][1] (687)

    4) When predFlagLX[0][0] is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in subclause 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, the variable availLR specifying left and right neighbouring blocks' availability of luma coding block as inputs, and the outputs being the

luma affine control point motion vector predictors 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:

$$uLX[cpIdx][0] = (mvpCpLX[cpIdx][0] + mvdCpLX[cpIdx][0] + 2^{16}) \% 2^{16} \quad (688)$$

$$cpMvLX[cpIdx][0] = (uLX[cpIdx][0] >= 2^{15}) ? (uLX[cpIdx][0] - 2^{16}) : uLX[cpIdx][0] \quad (689)$$

$$uLX[cpIdx][1] = (mvpCpLX[cpIdx][1] + mvdCpLX[cpIdx][1] + 2^{16}) \% 2^{16} \quad (690)$$

$$cpMvLX[cpIdx][1] = (uLX[cpIdx][1] >= 2^{15}) ? (uLX[cpIdx][1] - 2^{16}) : uLX[cpIdx][1] \quad (691)$$

— The derivation process for affine subblock size as specified in subclause 8.5.3.8 is invoked with 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 cpMvL0[cpIdx], cpMvL1[cpIdx] with cpIdx being 0..2, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, 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, and the flag clipMV specifying motion vector clipping type as outputs.

— 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 cpMvL0[cpIdx], cpMvL1[cpIdx] with cpIdx being 0..2, the prediction list utilization flags predFlagL0 and predFlagL1, and the reference indices refIdxL0 and refIdxL1 as inputs, and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the size of luma coding subblocks in horizontal direction sizeSbX and in vertical direction sizeSbY, 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 outputs.

— For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 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:

$$(xSb, ySb) = (xCb + xSbIdx * sizeSbX, yCb + ySbIdx * sizeSbY) \quad (692)$$

  — If xSbIdx and ySbIdx are both equal to 0 (top-left subblock), the following applies for x = 0..sizeSbX − 1 and y = 0..sizeSbY − 1:

$$MvLX[xSb + x][ySb + y] = cpMvLX[0] \quad (693)$$

  — Otherwise, if xSbIdx is equal to numSbX − 1 and ySbIdx is equal to 0 (top-right subblock), the following applies for x = 0..sizeSbX − 1 and y = 0..sizeSbY − 1:

$$MvLX[xSb + x][ySb + y] = cpMvLX[1] \quad (694)$$

  — Otherwise, if numCpMv is equal to 3 and xSbIdx is equal to 0 and ySbIdx is equal to numSbY − 1 (below-left subblock), the following applies for x = 0..sizeSbX − 1 and y = 0..sizeSbY − 1:

$$MvLX[xSb + x][ySb + y] = cpMvLX[2] \quad (695)$$

— Otherwise, for x = 0..sizeSbX − 1 and y = 0..sizeSbY − 1 motion vector MvLX[xSb + x][ySb + y] is set equal to motion vector mvLX[xSbIdx][ySbIdx] converted from 1/16 fractional-sample accuracy to 1/4 fractional-sample accuracy as specified in subclause 8.5.2.8.

— When sps_hmvp_flag is equal to 1, the updating process for the history-based motion vector predictor list as specified in subclause 8.5.2.7. is invoked with affine luma motion vectors center_mvLX and refIdxLX with X being 0 and 1.

— Affine motion vector center_mvLX for position cbWidth >> 1 and cbHeight >> 1 are derived as follows:

— The variable numCpMv is set equal to the number of control point motion vectors, cpMvLX[cpIdx] are set equal to the control point motion vectors of current block, with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

— Horizontal change of motion vector dX, vertical change of motion vector dY and base motion vector mvBaseScaled are derived by invoking the process as specified in subclause 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.

$$xPosSb = cbWidth >> 1 \quad (696)$$

$$yPosSb = cbHeight >> 1 \quad (697)$$

$$mvLX[0] = (mvBaseScaled[0] + dX[0] * xPosSb + dY[0] * yPosSb) \quad (698)$$

$$mvLX[1] = (mvBaseScaled[1] + dX[1] * xPosSb + dY[1] * yPosSb) \quad (699)$$

— The rounding process for motion vectors as specified in subclause 8.5.3.10 is invoked the with mvX set equal to mvLX, rightShift set equal to 7, and leftShift set equal to 0 as inputs, and the rounded mvLX as output.

— The motion vectors mvLX are clipped as follows:

$$center\_mvLX[0] = Clip3(-2^{15}, 2^{15} - 1, mvLX[0]) \quad (700)$$

$$center\_mvLX[1] = Clip3(-2^{15}, 2^{15} - 1, mvLX[1]) \quad (701)$$

— 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:

$$RefIdxL0[xCb + x][yCb + y] = refIdxL0 \quad (702)$$

$$RefIdxL1[xCb + x][yCb + y] = refIdxL1 \quad (703)$$

$$PredFlagL0[xCb + x][yCb + y] = predFlagL0 \quad (704)$$

$$PredFlagL1[xCb + x][yCb + y] = predFlagL1 \quad (705)$$

$$MvDmvrL0[xCb + x][yCb + y] = MvL0[xCb + x][yCb + y] \quad (706)$$

$$MvDmvrL1[xCb + x][yCb + y] = MvL1[xCb + x][yCb + y] \quad (707)$$

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, and

— 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 luma affine control point motion vector cpMvLX[cpIdx] with X being 0 or 1, and cpIdx = 0..numCpMv – 1,

— the reference indices refIdxL0 and refIdxL1, and

— the prediction list utilization flags predFlagL0 and predFlagL1.

The affine merging candidate list, affineMergeCandList, is derived by the following ordered steps:

1) The sample locations ($xNbA_0$, $yNbA_0$), ($xNbA_1$, $yNbA_1$), ($xNbA_2$, $yNbA_2$), ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$), ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$), ($xNbC_0$, $yNbC_0$), ($xNbC_1$, $yNbC_1$) and ($xNbC_2$, $yNbC_2$) are derived as follows:

$$(xNbA_0, yNbA_0) = (xCb - 1, yCb + cbHeight) \quad (708)$$

$$(xNbA_1, yNbA_1) = (xCb - 1, yCb + cbHeight - 1) \quad (709)$$

$$(xNbA_2, yNbA_2) = (xCb - 1, yCb) \quad (710)$$

$$(xNbB_0, yNbB_0) = (xCb + cbWidth, yCb - 1) \quad (711)$$

$$(xNbB_1, yNbB_1) = (xCb + cbWidth - 1, yCb - 1) \quad (712)$$

$$(xNbB_2, yNbB_2) = (xCb - 1, yCb - 1) \quad (713)$$

$$(xNbB_3, yNbB_3) = (xCb, yCb - 1) \quad (714)$$

$$(xNbC_0, yNbC_0) = (xCb + cbWidth, yCb + cbHeight) \quad (715)$$

$$(xNbC_1, yNbC_1) = (xCb + cbWidth, yCb + cbHeight - 1) \quad (716)$$

$$(xNbC_2, yNbC_2) = (xCb + cbWidth, yCb) \quad (717)$$

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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the neighbouring luma location (xNbBLK, yNbBLK) as input, 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, the following applies:

      — When availableFlagB_3 and availableFlagB_2 are both equal to 1, 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.

      — When availableFlagC_1 and availableFlagC_0 are both equal to 1, 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.

      — When availableFlagB_3 and availableFlagB_0 are both equal to 1, 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.

      — When availableFlagC_1 and availableFlagB_0 are both equal to 1, 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 (availLR is not equal to LR_01), the following applies:

      — When availableFlagB_1 and availableFlagB_0 are both equal to 1, 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.

      — When availableFlagA_1 and availableFlagA_0 are both equal to 1, 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.

      — When availableFlagB_1 and availableFlagB_2 are both equal to 1, 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.

      — When availableFlagA_1 and availableFlagB_2 are both equal to 1, 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], and numCpMv is set equal to MotionModelIdc[xNbBLK][yNbBLK] + 1.

— For X being replaced by either 0 or 1, the following applies:

— When PredFlagLX[xNbBLK][yNbBLK] is equal to 1, the derivation process for luma affine control point motion vectors from a neighbouring block as specified in subclause 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height cbWidth and cbHeight, the neighbouring luma coding block location (xNb, yNb), the neighbouring luma coding block width and height nbW and nbH, and the number of control point motion vectors numCpMv as inputs, the control point motion vector predictor candidates cpMvLXBLK[cpIdx] with cpIdx = 0..numCpMv − 1 as outputs.

— The following assignments are made:

$$predFlagLXBLK = PredFlagLX[xNbBLK][yNbBLK] \quad (718)$$

$$refIdxLXBLK = RefIdxLX[xNbBLK][yNbBLK] \quad (719)$$

6) The derivation process for constructed affine control point motion vector merging candidates as specified in subclause 8.5.3.4 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height cbWidth and cbHeight, the availability flags $availableA_0$, $availableA_1$, $availableA_2$, $availableB_0$, $availableB_1$, $availableB_2$, $availableB_3$, $availableC_0$, $availableC_1$, and $availableC_2$, 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 ($xNbC_2$, $yNbC_2$) as inputs, and 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 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 is set equal to 0.

— If availLR is equal to LR_01, the following applies:

```
if( availableFlagC1 && i < 5 )
    affineMergeCandList[ i++ ] = C1
if( availableFlagB3 && i < 5 )
    affineMergeCandList[ i++ ] = B3
if( availableFlagB2 && i < 5 )
    affineMergeCandList[ i++ ] = B2
if( availableFlagC0 && i < 5 )
    affineMergeCandList[ i++ ] = C0
if( availableFlagB0 && i < 5 )
    affineMergeCandList[ i++ ] = B0
```
(720)

— Otherwise (availLR is not equal to LR_01), the following applies:

```
if( availableFlagA1 && i < 5 )
    affineMergeCandList[ i++ ] = A1
if( availableFlagB1 && i < 5 )
    affineMergeCandList[ i++ ] = B1
if( availableFlagB0 && i < 5 )
    affineMergeCandList[ i++ ] = B0
if( availableFlagA0 && i < 5 )
    affineMergeCandList[ i++ ] = A0
```
(721)

```
if( availableFlagB2 && i < 5 )
    affineMergeCandList[ i++ ] = B2
```

— 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 )                                    (722)
    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:

$$refIdxL0ZeroCand_m = 0 \quad (723)$$

$$predFlagL0ZeroCand_m = 1 \quad (724)$$

$$cpMvL0ZeroCand_m[\ 0\ ] = mvZero \quad (725)$$

$$cpMvL0ZeroCand_m[\ 1\ ] = mvZero \quad (726)$$

$$cpMvL0ZeroCand_m[\ 2\ ] = mvZero \quad (727)$$

$$refIdxL1ZeroCand_m = (slice\_type == B)\ ?\ 0 : -1 \quad (728)$$

$$predFlagL1ZeroCand_m = (slice\_type == B)\ ?\ 1 : 0 \quad (729)$$

$$cpMvL1ZeroCand_m[\ 0\ ] = mvZero \quad (730)$$

$$cpMvL1ZeroCand_m[\ 1\ ] = mvZero \quad (731)$$

$$cpMvL1ZeroCand_m[\ 2\ ] = mvZero \quad (732)$$

$$motionModelIdcZeroCand_m = 1 \quad (733)$$

— 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:

$$affineMergeCandList[\ numCurrMergeCand{+}{+}\ ] = zeroCand_m \quad (734)$$

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 (735)

predFlagLX = predFlagLXN (736)

cpMvLX[0] = cpMvLXN[0] (737)

cpMvLX[1] = cpMvLXN[1] (738)

cpMvLX[2] = cpMvLXN[2] (739)

numCpMv = motionModelIdxN + 1 (740)

— The following assignment is made for x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[x][y] = numCpMv − 1 (741)

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, and

— the number of control point motion vectors numCpMv.

Outputs 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:

$$log2NbW = Log2(nNbW) \tag{742}$$

$$log2NbH = Log2(nNbH) \tag{743}$$

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

— If isCTUboundary is equal to TRUE, the following applies:

$$mvScaleHor = MvLX[xNb][yNb + nNbH - 1][0] << 7 \tag{744}$$

$$mvScaleVer = MvLX[xNb][yNb + nNbH - 1][1] << 7 \tag{745}$$

$$dHorX = (MvLX[xNb + nNbW - 1][yNb + nNbH - 1][0] - MvLX[xNb][yNb + nNbH - 1][0]) << (7 - log2NbW) \tag{746}$$

$$dVerX = (MvLX[xNb + nNbW - 1][yNb + nNbH - 1][1] - MvLX[xNb][yNb + nNbH - 1][1]) << (7 - log2NbW) \tag{747}$$

— Otherwise (isCTUboundary is equal to FALSE), the following applies:

$$mvScaleHor = MvLX[xNb][yNb][0] << 7 \tag{748}$$

$$mvScaleVer = MvLX[xNb][yNb][1] << 7 \tag{749}$$

$$dHorX = (MvLX[xNb + nNbW - 1][yNb][0] - MvLX[xNb][yNb][0]) << (7 - log2NbW) \tag{750}$$

$$dVerX = (MvLX[xNb + nNbW - 1][yNb][1] - MvLX[xNb][yNb][1]) << (7 - log2NbW) \tag{751}$$

The variables dHorY and dVerY are derived as follows:

— If isCTUboundary is equal to FALSE and MotionModelIdc[xNb][yNb] is equal to 2, the following applies:

$$dHorY = (MvLX[xNb][yNb + nNbH - 1][0] - MvLX[xNb][yNb][0]) << (7 - log2NbH) \tag{752}$$

$$dVerY = (MvLX[xNb][yNb + nNbH - 1][1] - MvLX[xNb][yNb][1]) << (7 - log2NbH) \tag{753}$$

— Otherwise (isCTUboundary is equal to TRUE or MotionModelIdc[xNb][yNb] is equal to 1), the following applies:

$$dHorY = -dVerX \tag{754}$$

$$dVerY = dHorX \tag{755}$$

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:

$$cpMvLX[\,0\,][\,0\,] = (\,mvScaleHor + dHorX * (\,xCb - xNb\,) + dHorY * (\,yCb - yNb\,)\,) \quad (756)$$

$$cpMvLX[\,0\,][\,1\,] = (\,mvScaleVer + dVerX * (\,xCb - xNb\,) + dVerY * (\,yCb - yNb\,)\,) \quad (757)$$

$$cpMvLX[\,1\,][\,0\,] = (\,mvScaleHor + dHorX * (\,xCb + cbWidth - xNb\,) + dHorY * (\,yCb - yNb\,)\,) \quad (758)$$

$$cpMvLX[\,1\,][\,1\,] = (\,mvScaleVer + dVerX * (\,xCb + cbWidth - xNb\,) + dVerY * (\,yCb - yNb\,)\,) \quad (759)$$

— If numCpMv is equal to 3, the third control point vector cpMvLX[2] is derived as follows:

$$cpMvLX[\,2\,][\,0\,] = (\,mvScaleHor + dHorX * (\,xCb - xNb\,) + dHorY * (\,yCb + cbHeight - yNb\,)\,) \quad (760)$$

$$cpMvLX[\,2\,][\,1\,] = (\,mvScaleVer + dVerX * (\,xCb - xNb\,) + dVerY * (\,yCb + cbHeight - yNb\,)\,) \quad (761)$$

— The rounding process for motion vectors as specified in subclause 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:

$$cpMvLX[\,cpIdx\,][\,0\,] = Clip3(\,-2^{15},\,2^{15}-1,\,cpMvLX[\,cpIdx\,][\,0\,]\,) \quad (762)$$

$$cpMvLX[\,cpIdx\,][\,1\,] = Clip3(\,-2^{15},\,2^{15}-1,\,cpMvLX[\,cpIdx\,][\,1\,]\,) \quad (763)$$

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 $availableA_0$, $availableA_1$, $availableA_2$, $availableB_0$, $availableB_1$, $availableB_2$, $availableB_3$, $availableC_0$, $availableC_1$, and $availableC_2$, and

— 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 ($xNbC_2$, $yNbC_2$).

Outputs 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, and

— the constructed affine control point motion vectors cpMvLXConstK[cpIdx] with cpIdx = 0..2, K = 1..6 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLXCorner[0], reference index refIdxLXCorner[0], prediction list utilization flag predFlagLXCorner[0] and the availability flag availableFlagCorner[0] with X being 0 and 1 are derived as follows:

— The availability flag availableFlagCorner[0] is set equal to FALSE.

— The following applies for (xNbTL, yNbTL) with TL being replaced in order 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] (764)

predFlagLXCorner[0] = PredFlagLX[xNbTL][yNbTL] (765)

cpMvLXCorner[0] = MvLX[xNbTL][yNbTL] (766)

availableFlagCorner[0] = TRUE (767)

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 in order by B_0, B_1, and C_2:

  — 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] (768)

predFlagLXCorner[1] = PredFlagLX[xNbTR][yNbTR] (769)

cpMvLXCorner[1] = MvLX[xNbTR][yNbTR] (770)

availableFlagCorner[1] = TRUE (771)

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.

— If availLR is equal to LR_10 or LR_11, the following applies for (xNbBL, yNbBL) with BL being replaced in order by A_0 and A_1:

  — 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] (772)

predFlagLXCorner[2] = PredFlagLX[xNbBL][yNbBL] (773)

cpMvLXCorner[2] = MvLX[xNbBL][yNbBL] (774)

availableFlagCorner[2] = TRUE (775)

— Otherwise, the reference indices for the temporal merging candidate, refIdxLXCorner[2], 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:

xColBl = xCb – 1 (776)

yColBl = yCb + cbHeight (777)

— If yCb >> CtbLog2SizeY is equal to yColBl >> CtbLog2SizeY, yColBl is less than pic_height_in_luma_samples and xColBl is larger than 0, the following applies:

— The luma location (xColCb, yColCb) is set equal to ((xColBl >> 3) << 3, (yColBl >> 3) << 3).

— The derivation process for collocated motion vectors as specified in subclause 8.5.2.3.4 is invoked with luma location coordinates (xColCb, yColCb) as input, 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[2], predFlagL0Corner[2], cpMvL0Corner[2] and predFlagL1Corner[2] are derived as follows:

availableFlagCorner[2] = availableFlagL0Col (778)

predFlagL0Corner[2] = availableFlagL0Col (779)

cpMvL0Corner[2] = mvL0Col (780)

predFlagL1Corner[2] = 0 (781)

— When slice_type is equal to B, the variables availableFlagCorner[2], predFlagL1Corner[2] and cpMvL1Corner[2] are derived as follows:

availableFlagCorner[2] = availableFlagL0Col || availableFlagL1Col (782)

predFlagL1Corner[2] = availableFlagL1Col (783)

cpMvL1Corner[2] = mvL1Col (784)

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 BR being replaced in order by C_0 and C_1:

— 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] (785)

predFlagLXCorner[3] = PredFlagLX[xNbBR][yNbBR] (786)

cpMvLXCorner[3] = MvLX[xNbBR][yNbBR] (787)

availableFlagCorner[3] = TRUE (788)

— 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 (789)

yColBr = yCb + cbHeight (790)

— 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 luma location (xColCb, yColCb) is set equal to ((xColBr >> 3) << 3, (yColBr >> 3) << 3).

— The derivation process for collocated motion vectors as specified in subclause 8.5.2.3.4 is invoked with the luma location coordinates (xColCb, yColCb) as input, 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 (791)

predFlagL0Corner[3] = availableFlagL0Col (792)

cpMvL0Corner[3] = mvL0Col (793)

predFlagL1Corner[3] = 0 (794)

— When slice_type is equal to B, the variables availableFlagCorner[3], predFlagL1Corner[3] and cpMvL1Corner[3] are derived as follows:

availableFlagCorner[3] = availableFlagL0Col || availableFlagL1Col (795)

predFlagL1Corner[3] = availableFlagL1Col (796)

cpMvL1Corner[3] = mvL1Col (797)

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 the 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:

$$predFlagLXConst1 = 1 \tag{798}$$

$$refIdxLXConst1 = refIdxLXCorner[0] \tag{799}$$

$$cpMvLXConst1[0] = cpMvLXCorner[0] \tag{800}$$

$$cpMvLXConst1[1] = cpMvLXCorner[1] \tag{801}$$

$$cpMvLXConst1[2] = cpMvLXCorner[2] \tag{802}$$

   — 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 the 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:

$$predFlagLXConst2 = 1 \tag{803}$$

$$refIdxLXConst2 = refIdxLXCorner[0] \tag{804}$$

$$cpMvLXConst2[0] = cpMvLXCorner[0] \tag{805}$$

$$cpMvLXConst2[1] = cpMvLXCorner[1] \tag{806}$$

$$cpMvLXConst2[2] = cpMvLXCorner[3] + cpMvLXCorner[0] - cpMvLXCorner[1] \tag{807}$$

$$cpMvLXConst2[2][0] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst2[2][0]) \tag{808}$$

$$cpMvLXConst2[2][1] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst2[2][1]) \tag{809}$$

— 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 the 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:

$$predFlagLXConst3 = 1 \tag{810}$$

refIdxLXConst3 = refIdxLXCorner[0] (811)

cpMvLXConst3[0] = cpMvLXCorner[0] (812)

cpMvLXConst3[1] = cpMvLXCorner[3] + cpMvLXCorner[0] − cpMvLXCorner[2] (813)

cpMvLXConst3[1][0] = Clip3(-2^{15}, $2^{15} - 1$, cpMvLXConst3[1][0]) (814)

cpMvLXConst3[1][1] = Clip3(-2^{15}, $2^{15} - 1$, cpMvLXConst3[1][1]) (815)

cpMvLXConst3[2] = cpMvLXCorner[2] (816)

— 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 (817)

refIdxLXConst4 = refIdxLXCorner[1] (818)

cpMvLXConst4[0] = cpMvLXCorner[1] + cpMvLXCorner[2] − cpMvLXCorner[3] (819)

cpMvLXConst4[0][0] = Clip3(-2^{15}, $2^{15} - 1$, cpMvLXConst4[0][0]) (820)

cpMvLXConst4[0][1] = Clip3(-2^{15}, $2^{15} - 1$, cpMvLXConst4[0][1]) (821)

cpMvLXConst4[1] = cpMvLXCorner[1] (822)

cpMvLXConst4[2] = cpMvLXCorner[2] (823)

— 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:

$$predFlagLXConst5 = 1 \tag{824}$$

$$refIdxLXConst5 = refIdxLXCorner[\ 0\ ] \tag{825}$$

$$cpMvLXConst5[\ 0\ ] = cpMvLXCorner[\ 0\ ] \tag{826}$$

$$cpMvLXConst5[\ 1\ ] = cpMvLXCorner[\ 1\ ] \tag{827}$$

— 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:

$$\begin{aligned} cpMvLXCorner[\,1\,][\,0\,] = {} & (cpMvLXCorner[\,0\,][\,0\,] << 7) + \\ & ((cpMvLXCorner[\,2\,][\,1\,] - cpMvLXCorner[\,0\,][\,1\,]) << \\ & (7 + Log2(cbWidth) - Log2(cbHeight / cbWidth))) \end{aligned} \quad (828)$$

$$\begin{aligned} cpMvLXCorner[\,1\,][\,1\,] = {} & (cpMvLXCorner[\,0\,][\,1\,] << 7) - \\ & ((cpMvLXCorner[\,2\,][\,0\,] - cpMvLXCorner[\,0\,][\,0\,]) << \\ & (7 + Log2(cbWidth) - Log2(cbHeight / cbWidth))) \end{aligned} \quad (829)$$

— The rounding process for motion vectors as specified in subclause 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:

$$predFlagLXConst6 = 1 \quad (830)$$

$$refIdxLXConst6 = refIdxLXCorner[\,0\,] \quad (831)$$

$$cpMvLXConst6[\,0\,] = cpMvLXCorner[\,0\,] \quad (832)$$

$$cpMvLXConst6[\,1\,] = cpMvLXCorner[\,1\,] \quad (833)$$

$$cpMvLXConst6[\,1\,][\,0\,] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst6[\,1\,][\,0\,]) \quad (834)$$

$$cpMvLXConst6[\,1\,][\,1\,] = Clip3(-2^{15}, 2^{15} - 1, cpMvLXConst6[\,1\,][\,1\,]) \quad (835)$$

— 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, and

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

Outputs 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 ($xNbA_0$, $yNbA_0$), ($xNbA_1$, $yNbA_1$), ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$), ($xNbB_2$, $yNbB_2$), ($xNbC_0$, $yNbC_0$), and ($xNbC_1$, $yNbC_1$) are derived as follows:

$$(xNbA_0, yNbA_0) = (xCb - 1, yCb + cbHeight) \quad (836)$$

$$(xNbA_1, yNbA_1) = (xCb - 1, yCb + cbHeight - 1) \quad (837)$$

$$(xNbB_0, yNbB_0) = (xCb + cbWidth, yCb - 1) \quad (838)$$

$$(xNbB_1, yNbB_1) = (xCb + cbWidth - 1, yCb - 1) \quad (839)$$

$$(xNbB_2, yNbB_2) = (xCb - 1, yCb - 1) \quad (840)$$

$$(xNbC_0, yNbC_0) = (xCb + cbWidth, yCb + cbHeight) \quad (841)$$

$$(xNbC_1, yNbC_1) = (xCb + cbWidth, yCb + cbHeight - 1) \quad (842)$$

4) The following applies for ($xNbA_k$, $yNbA_k$) from ($xNbA_0$, $yNbA_0$) to ($xNbA_1$, $yNbA_1$):

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the neighbouring luma location ($xNbA_k$, $yNbA_k$) as input, 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 subclause 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height cbWidth and cbHeight, the neighbouring luma coding block location (xNb, yNb), the neighbouring luma coding block width and height nbW and 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 outputs.

— The following assignments are made:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (843)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 1\ ] = cpMvpLX[\ 1\ ] \quad (844)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 2\ ] = cpMvpLX[\ 2\ ] \quad (845)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (846)$$

5) The following applies for ($xNbB_k$, $yNbB_k$) from ($xNbB_0$, $yNbB_0$) to ($xNbB_2$, $yNbB_2$):

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the neighbouring luma location ($xNbB_k$, $yNbB_k$) as input, 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 subclause 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height cbWidth and cbHeight, the neighbouring luma coding block location (xNb, yNb), the neighbouring luma coding block width and height nbW and 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 outputs.

— The following assignments are made:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (847)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 1\ ] = cpMvpLX[\ 1\ ] \quad (848)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 2\ ] = cpMvpLX[\ 2\ ] \quad (849)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (850)$$

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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the neighbouring luma location ($xNbC_k$, $yNbC_k$) as input, 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 subclause 8.5.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height cbWidth and cbHeight, the neighbouring luma coding block location (xNb, yNb), the neighbouring luma coding block width and height nbW and nbH, and the number of control point motion vectors numCpMv as inputs, the control point motion vector predictor candidates cpMvpLX[cpIdx] with cpIdx = 0..numCpMv − 1 as outputs.

— The following assignments are made:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (851)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 1\ ] = cpMvpLX[\ 1\ ] \quad (852)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 2\ ] = cpMvpLX[\ 2\ ] \quad (853)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (854)$$

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 subclause 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..3 as outputs.

— When availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 0\ ] = cpMvpLX[\ 0\ ] \quad (855)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 1\ ] = cpMvpLX[\ 1\ ] \quad (856)$$

$$cpMvpListLX[\ numCpMvpCandLX\ ][\ 2\ ] = cpMvpLX[\ 2\ ] \quad (857)$$

$$numCpMvpCandLX = numCpMvpCandLX + 1 \quad (858)$$

8) The following applies for cpIdx = 2..0:

— If cpIdx is equal to 2 and availableFlagLX[cpIdx] is equal to 0, cpIdx is modified to 3.

— When numCpMvpCandLX is less than 2 and availableFlagLX[cpIdx] is equal to 1, the following assignments are made:

cpMvpListLX[numCpMvpCandLX][0] = cpMvpLX[cpIdx] (859)

cpMvpListLX[numCpMvpCandLX][1] = cpMvpLX[cpIdx] (860)

cpMvpListLX[numCpMvpCandLX][2] = cpMvpLX[cpIdx] (861)

numCpMvpCandLX = numCpMvpCandLX + 1 (862)

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 (863)

cpMvpListLX[numCpMvpCandLX][1] = mvZero (864)

cpMvpListLX[numCpMvpCandLX][2] = mvZero (865)

numCpMvpCandLX = numCpMvpCandLX + 1 (866)

The affine control point motion vector predictor cpMvpLX with X being 0 or 1 is derived as follows:

cpMvpLX = cpMvpListLX[affine_mvp_flag_lX[xCb][yCb]] (867)

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, and

— the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Outputs 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..3 and X being 0 or 1, and

— the constructed affine control point motion vector prediction candidates cpMvLX[cpIdx] with cpIdx = 0..3 and X being 0 or 1.

The sample locations ($xNbA_0$, $yNbA_0$), ($xNbA_1$, $yNbA_1$), ($xNbA_2$, $yNbA_2$), ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$), ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$), and ($xNbC_2$, $yNbC_2$) are derived as follows:

— The sample locations ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$)($xNbA_2$, $yNbA_2$) and ($xNbC_2$, $yNbC_2$) are set equal to (xCb − 1, yCb − 1), (xCb, yCb − 1), (xCb − 1, yCb) and (xCb + cbWidth, 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.

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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked the luma location (xNbY, yNbY) set equal to (xNbTL, yNbTL) as input, 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:

$$cpMvLX[\ 0\ ] = MvLX[\ xNbTL\ ][\ yNbTL\ ] \tag{868}$$

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) with TR being replaced in given order by B_0, B_1 and C_2:

   — The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location (xNbY, yNbY) set equal to (xNbTR, yNbTR) as input, 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:

$$cpMvLX[\ 1\ ] = MvLX[\ xNbTR\ ][\ yNbTR\ ] \tag{869}$$

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 derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked the luma location (xNbY, yNbY) set equal to (xNbBL, yNbBL) as input, 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:

$$cpMvLX[\ 2\ ] = MvLX[\ xNbBL\ ][\ yNbBL\ ] \tag{870}$$

The fourth (bottom-right) control point motion vector cpMvLX[3] and the availability flag availableFlagLX[3] are derived in the following ordered steps:

1) 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.

2) The availability flag availableFlagLX[3] is set equal to 0 and both components of cpMvLX[3] are set equal to 0.

3) The following applies for (xNbBR, yNbBR) with BR being replaced by C_1 and C_0:

— The derivation process for neighbouring block motion vector candidate availability as specified in subclause 6.4.3 is invoked with the luma location (xNbY, yNbY) set equal to (xNbBR, yNbBR) as input, 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:

$$cpMvLX[\ 3\ ] = MvLX[\ xNbBR\ ][\ yNbBR\ ] \tag{871}$$

The variable availableConsFlagLX is derived as follows:

— If availableFlagLX[0] is equal to 1, 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, 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:

$$cpMvLX[\ 2\ ] = Clip3(\ -2^{15},\ 2^{15} - 1,\ cpMvLX[\ 3\ ] + cpMvLX[\ 0\ ] - cpMvLX[\ 1\ ]\ ) \tag{872}$$

— Otherwise, if availableFlagLX[0] is equal to 1, 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 prediction list utilization flags predFlagLX, with X being 0 or 1, and

— the reference index refIdxLX and X being 0 or 1.

Outputs of this process are:

— the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

— the size of luma coding subblocks in horizontal direction sizeSbX and in vertical direction sizeSbY,

— the luma subblock motion vector array mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1 and X being 0 or 1, and

— the chroma subblock motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1 and X being 0 or 1.

The variables sizeSbX, sizeSbY, numSbX, numSbY, and clipMV flag are derived according to subclause 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 as specified in subclause 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..numSbY − 1:

  — For xSbIdx = 0..numSbX − 1:

    — The luma motion vector mvLX[xSbIdx][ySbIdx] is derived as follows:

$$xPosSb = sizeSbX * xSbIdx + (sizeSbX >> 1) \quad (873)$$

$$yPosSb = sizeSbY * ySbIdx + (sizeSbY >> 1) \quad (874)$$

$$mvLX[xSbIdx][ySbIdx][0] = (mvBaseScaled[0] + dX[0] * xPosSb + dY[0] * yPosSb) \quad (875)$$

$$mvLX[xSbIdx][ySbIdx][1] = (mvBaseScaled[1] + dX[1] * xPosSb + dY[1] * yPosSb) \quad (876)$$

    — The rounding process for motion vectors as specified in subclause 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:

$$mvLX[xSbIdx][ySbIdx][0] = Clip3(-2^{17}, 2^{17} - 1, mvLX[xSbIdx][ySbIdx][0]) \quad (877)$$

$$mvLX[xSbIdx][ySbIdx][1] = Clip3(-2^{17}, 2^{17} - 1, mvLX[xSbIdx][ySbIdx][1]) \quad (878)$$

— The derivation process for chroma motion vectors as specified in subclause 8.5.2.6 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, and

— the prediction list utilization flags predFlagLX, with 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, and

— the flag clipMV indicating motion vector clipping type for blocks processed with EIF.

The variables sizeSbX and sizeSbY are derived as follows:

— sizeSbX is set equal to cbWidth.

— sizeSbY is set equal to cbHeight.

The variables eifCanBeAppliedX and clipMVX are derived as follows for X being 0 and 1:

— eifCanBeAppliedX is set equal to TRUE.

— clipMVX is set equal to FALSE.

The variable eifSubblockSize is set equal to 4.

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 as specified in subclause 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.

— The variable mvWx and mvWy are derived as follows:

$$mvWx = Max(Abs(dX[0]), Abs(dX[1])) \quad (879)$$

$$mvWy = Max(Abs(dY[0]), Abs(dY[1])) \quad (880)$$

— The variable sizeSbXTemp is specificed in Table 22 according to the value of mvWx.

— The variable sizeSbYTemp is specificed in Table 23 according to the value of mvWy.

— The variable sizeSbX is modified as follow:

sizeSbX = Min(sizeSbX, sizeSbXTemp) (881)

— The variable sizeSbY is modified as follow:

sizeSbY = Min(sizeSbY, sizeSbYTemp) (882)

Table 22 — Specification of sizeSbXTemp for various input values of mvWx

mvWx	0	1	2	3	4	> 4
sizeSbX	cbWidth	32	16	8	8	4

Table 23 — Specification of sizeSbYTemp for various input values of mvWy

mvWy	0	1	2	3	4	> 4
sizeSbY	cbHeight	32	16	8	8	4

— The variables eifCanBeAppliedX and clipMVX are modified as follows:

— The arrays X[i] and Y[i] are derived as follows:

X[0] = 0 (883)

X[1] = (eifSubblockSize + 1) * (dX[0] + (1 << 9)) (884)

X[2] = (eifSubblockSize + 1) * dY[0] (885)

X[3] = X[1] + X[2] (886)

Y[0] = 0 (887)

Y[1] = (eifSubblockSize + 1) * dX[1] (888)

Y[2] = (eifSubblockSize + 1) * (dY[1] + (1 << 9)) (889)

Y[3] = Y[1] + Y[2] (890)

— The variable Xmax is set equal to maximum of X[i] for i is equal 0..3.

— The variable Xmin is set equal to minimum of X[i] for i is equal 0..3.

— The variable Ymax is set equal to maximum of Y[i] for i is equal 0..3.

— The variable Ymin is set equal to minimum of Y[i] for i is equal 0..3.

— The variable W is set equal to (Xmax – Xmin + (1 << 9) – 1) >> 9.

— The variable H is set equal to (Ymax − Ymin + (1 << 9) − 1) >> 9.

— If (W + 2) * (H + 2) is greater than 72, the variable clipMVX is set equal to TRUE.

— If dY[1] is less than ((−1) << 9), then the variable eifCanBeAppliedX is set equal to FALSE.

— Otherwise, the following applies:

— If (Max(0, dY[1]) + Abs(dX[1])) * (1 + eifSubblockSize) is greater than (1 << 9), then the variable eifCanBeAppliedX is set equal to FALSE.

The variables eifCanBeApplied and clipMV are derived as follows:

eifCanBeApplied = eifCanBeApplied0 & eifCanBeApplied1 (891)

clipMV = clipMV0 | clipMV1 (892)

If eifCanBeApplied is equal to FALSE, then the variables sizeSbX and sizeSbY are modified as follows:

sizeSbX = Max(8, sizeSbX) (893)

sizeSbY = Max(8, sizeSbY) (894)

The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

numSbX = cbWidth / sizeSbX (895)

numSbY = cbHeight / sizeSbY (896)

8.5.3.9 Derivation 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, and

— the control point motion vectors cpMvLX[cpIdx], with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

Outputs of this process are:

— the horizontal change of motion vector dX,

— the vertical change of motion vector dY, and

— the 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) (897)

log2CbH = Log2(cbHeight) (898)

Horizontal change of motion vector dX is derived as follows:

dX[0] = (cpMvLX[1][0] − cpMvLX[0][0]) << (7 − log2CbW) (899)

dX[1] = (cpMvLX[1][1] − cpMvLX[0][1]) << (7 − log2CbW) (900)

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) (901)

dY[1] = (cpMvLX[2][1] − cpMvLX[0][1]) << (7 − log2CbH) (902)

— Otherwise (numCpMv is equal to 2), dY is derived as follows:

dY[0] = −dX[1] (903)

dY[1] = dX[0] (904)

Motion vector mvBaseScaled corresponding to the top-left corner of the luma coding block is derived as follows:

mvBaseScaled[0] = cpMvLX[0][0] << 7 (905)

mvBaseScaled[1] = cpMvLX[0][1] << 7 (906)

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, and

— 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)) (907)

mvX[0] = ((mvX[0] + offset − (mvX[0] >= 0)) >> rightShift) << leftShift (908)

mvX[1] = ((mvX[1] + offset − (mvX[1] >= 0)) >> rightShift) << leftShift (909)

8.5.4 Decoding process for inter prediction samples

8.5.4.1 General

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 numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,

— the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1,

— the refined motion vectors refMvL0[xSbIdx][ySbIdx] and refMvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1,

— the chroma motion vectors in 1/32 fractional-sample accuracy mvCL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,

— the refined chroma motion vectors in 1/32 fractional-sample accuracy refMvCL0[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, and refMvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,

— the reference indices refIdxL0 and refIdxL1,

— the prediction list utilization flags, predFlagL0, and predFlagL1,

— 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, and

— clipMV flag specifying motion vector clipping type.

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 in this subclause,

— 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 in this subclause, and

— 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 in this subclause.

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 as specified in subclause 8.5.4.2 with refIdxLX as input.

— The width and the height of the current luma coding subblock sbWidth, sbHeight are derived as follows:

$$sbWidth = nCbW / numSbX \quad (910)$$

$$sbHeight = nCbH / numSbY \quad (911)$$

— 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 as specified in subclause 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 as specified in subclause 8.5.4.4.1 with the luma location (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 bit depth $BitDepth_Y$, picture width pic_width_in_luma_samples, height pic_height_in_luma_samples, clipMV flag and flag isLuma equal to TRUE as inputs.

  — When ChromaArrayType is not equal to 0, the following applies:

    — The arrays predSamplesLXCb, is derived by invoking interpolation process for enhanced interpolation filter as specified in subclause 8.5.4.4.1 with the luma location (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 $refPicLX_{Cb}$, sample bit depth $BitDepth_C$, picture width pic_width_in_luma_samples and height pic_height_in_luma_samples, clipMV flag and flag isLuma equal to FALSE as inputs.

    — The arrays $predSamplesLX_{Cr}$, is derived by invoking interpolation process for enhanced interpolation filter as specified in subclause 8.5.4.4.1 with the luma location (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 $refPicLX_{Cr}$, sample bit depth $BitDepth_C$, picture width pic_width_in_luma_samples and height pic_height_in_luma_samples, clipMV flag and flag isLuma equal to FALSE as inputs.

— Otherwise, for each coding subblock at subblock index (xSbIdx, ySbIdx) with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the following applies:

  — The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

$$(xSb, ySb) = (xCb + xSbIdx * sbWidth, yCb + ySbIdx * sbHeight) \quad (912)$$

— The motion vector offset mvOffset is set equal to refMvLX[xSbIdx][xSbIdx] − mvLX[xSbIdx][ySbIdx].

— The variable isChromaPresent is set equal to TRUE if ChromaArrayType is not equal to 0, and is set equal to FALSE otherwise.

— The arrays predSamplesLX$_L$ and, when isChromaPresent, predSamplesLX$_{Cb}$, and predSamplesLX$_{Cr}$ are derived by invoking the fractional sample interpolation process as specified in subclause 8.5.4.3.1 with the luma locations (xSb, ySb), the luma coding block width sbWidth, the luma coding block height sbHeight, the refined motion vector refMvLX[xSb][ySb], the luma motion vector offset mvOffset, the reference array refPicLX$_L$, the chroma component presence indicator isChromaPresent, and, when isChromaPresent, motion vector refMvCLX[xSb][ySb], and the reference arrays refPicLX$_{Cb}$, 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 sample prediction process as specified in subclause 8.5.4.5 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 sample prediction process as specified in subclause 8.5.4.5 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 sample prediction process as specified in subclause 8.5.4.5 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.

8.5.4.2 Reference picture selection process

Input to this process is a reference index refIdxLX.

Outputs of this process are:

— 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 subclause 8.8 for a previously-decoded picture.

8.5.4.3 Fractional sample interpolation process

8.5.4.3.1 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,

— the selected reference picture sample array $refPicLX_L$,

— chroma component presence indicator isChromaPresent,

— when isChromaPresent, a chroma motion vector mvCLX given in 1/32-chroma-sample units, and

— when isChromaPresent, selected reference picture sample arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

Outputs of this process are:

— an (sbWidthLX)x(sbHeightLX) array $predSamplesLX_L$ of prediction luma sample values, and

— when isChromaPresent, two (sbWidthLX / SubWidthC)x(sbHeightLX / SubHeightC) 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 subclause 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:

$$xInt_L = xSb + (refMvLX[0] >> 4) + x_L \quad (913)$$

$$yInt_L = ySb + (refMvLX[1] >> 4) + y_L \quad (914)$$

$$xFrac_L = refMvLX[0] \& 15 \quad (915)$$

$yFrac_L = refMvLX[1] \& 15$ (916)

— The prediction luma sample value $predSamplesLX_L[x_L][y_L]$ is derived by invoking the process as specified in subclause 8.5.4.3.2 with ($xInt_L$, $yInt_L$), ($xFrac_L$, $yFrac_L$), ($xSbInt_L$, $ySbInt_L$), sbWidth, sbHeight and $refPicLX_L$ as inputs.

If isChromaPresent the following applies:

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 subclause 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 / SubWidthC − 1, y_C = 0..sbHeight / SubHeightC − 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:

$xInt_C = (xSb / SubWidthC) + (mvCLX[0] >> 5) + x_C$ (917)

$yInt_C = (ySb / SubHeightC) + (mvCLX[1] >> 5) + y_C$ (918)

$xFrac_C = mvCLX[0] \& 31$ (919)

$yFrac_C = mvCLX[1] \& 31$ (920)

— The prediction sample value $predSamplesLX_{Cb}[x_C][y_C]$ is derived by invoking the process as specified in subclause 8.5.4.3.3 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$), (xSbIntC, ySbIntC), sbWidth / SubWidthC, sbHeight / SubHeightC and $refPicLX_{Cb}$ as inputs.

— The prediction sample value $predSamplesLX_{Cr}[x_C][y_C]$ is derived by invoking the process as specified in subclause 8.5.4.3.3 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$), (xSbIntC, ySbIntC), sbWidth / SubWidthC, sbHeight / SubHeightC and $refPicLX_{Cr}$ as inputs.

8.5.4.3.2 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 refPicLXL,

— a variable sbWidth specifying the width of the current subblock, and

— 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 offset2 is set equal to 1 << (shift2 – 1).

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 24 or Table 25.

The luma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..7:

$$xInt_i = Clip3(0, picW - 1, xInt_L + i - 3) \quad (921)$$

$$yInt_i = Clip3(0, picH - 1, yInt_L + i - 3) \quad (922)$$

The luma locations in full-sample are further modified as follows for i = 0..7:

$$xInt_i = Clip3(xSbInt_L - 3, xSbInt_L + sbWidth + 3, xInt_i) \quad (923)$$

$$yInt_i = Clip3(ySbInt_L - 3, ySbInt_L + sbHeight + 3, yInt_i) \quad (924)$$

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:

$$predSampleLX_L = refPicLX_L[xInt_3][yInt_3] \quad (925)$$

— 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:

$$predSampleLX_L = Clip3(0, (1 << BitDepth_Y) - 1, (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_3]) >> 6) \quad (926)$$

— 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:

$$predSampleLX_L = Clip3(0, (1 << BitDepth_Y) - 1, (\sum_{i=0}^{7} f_L[yFrac_L][i] * refPicLX_L[xInt_3][yInt_i]) >> 6) \quad (927)$$

— 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:

$$temp[n] = (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_n]) >> shift1 \quad (928)$$

  — The predicted luma sample value $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = Clip3(0, (1 << BitDepth_Y) - 1, ((\sum_{i=0}^{7} f_L[yFrac_L][i] * temp[i]) + offset2) >> shift2) \quad (929)$$

Table 24 — Specification of the luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p for sps_admvp_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 25 — Specification of the luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p for sps_admvp_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	na	na	na	na	na	na	na	na
2	na	na	na	na	na	na	na	na
3	na	na	na	na	na	na	na	na
4	0	1	−5	52	20	−5	1	0
5	na	na	na	na	na	na	na	na
6	na	na	na	na	na	na	na	na
7	na	na	na	na	na	na	na	na
8	0	2	−10	40	40	−10	2	0
9	na	na	na	na	na	na	na	na
10	na	na	na	na	na	na	na	na
11	na	na	na	na	na	na	na	na
12	0	1	−5	20	52	−5	1	0
13	na	na	na	na	na	na	na	na
14	na	na	na	na	na	na	na	na
15	na	na	na	na	na	na	na	na

NOTE The value 0 specified in Tables 24 and 25 can be considered as the value "na" for the implementation.

8.5.4.3.3 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, and

— 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 26 or Table 27.

The chroma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..3:

$$xInt_i = Clip3(0, picW_C - 1, xInt_C + i - 1) \quad (930)$$

$$yInt_i = Clip3(0, picH_C - 1, yInt_C + i - 1) \quad (931)$$

The chroma locations in full-sample units ($xInt_i$, $yInt_i$) are further modified as follows for i = 0..3:

$$xInt_i = Clip3(xSbIntC - 1, xSbIntC + sbWidth + 1, xInt_i) \quad (932)$$

$$yInt_i = Clip3(ySbIntC - 1, ySbIntC + sbHeight + 1, yInt_i) \quad (933)$$

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:

$$predSampleLX_C = refPicLX_C[xInt_1][yInt_1] \quad (934)$$

— 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:

$$predSampleLX_C = Clip3(0, (1 << BitDepth_C) - 1, (\sum_{i=0}^{3} f_C[xFrac_C][i] * refPicLX_C[xInt_i][yInt_1]) >> 6) \quad (935)$$

— 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:

$$predSampleLX_C = Clip3(0, (1 << BitDepth_C) - 1, (\sum_{i=0}^{3} f_C[yFrac_C][i] * refPicLX_C[xInt_1][yInt_i]) >> 6) \quad (936)$$

— 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:

$$temp[n] = (\sum_{i=0}^{3} f_C[xFrac_C][i] * refPicLX_C[xInt_i][yInt_n]) >> shift1 \quad (937)$$

  — The predicted chroma sample value $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = Clip3(0, (1 << BitDepth_C) - 1, (offset2 + f_C[yFrac_C][0] * temp[0] + f_C[yFrac_C][1] * temp[1] + f_C[yFrac_C][2] * temp[2] + f_C[yFrac_C][3] * temp[3]) >> shift2) \quad (938)$$

Table 26 — Specification of the chroma interpolation filter coefficients fC[p] for each 1/32 fractional sample position p for sps_admvp_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

Fractional sample position p	interpolation filter coefficients			
	$f_C[p][0]$	$f_C[p][1]$	$f_C[p][2]$	$f_C[p][3]$
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 27 — Specification of the chroma interpolation filter coefficients fC[p] for each 1/32 fractional sample position p for sps_admvp_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	na	na	na	na
2	na	na	na	na
3	na	na	na	na
4	–2	58	10	–2
5	na	na	na	na
6	na	na	na	na
7	na	na	na	na
8	–4	52	20	–4
9	na	na	na	na
10	na	na	na	na
11	na	na	na	na
12	–6	46	30	–6
13	na	na	na	na
14	na	na	na	na
15	na	na	na	na
16	–8	40	40	–8
17	na	na	na	na
18	na	na	na	na
19	na	na	na	na
20	–4	28	46	–6
21	na	na	na	na
22	na	na	na	na
23	na	na	na	na
24	–4	20	52	–4
25	na	na	na	na
26	na	na	na	na
27	na	na	na	na
28	–2	10	58	–2
29	na	na	na	na
30	na	na	na	na
31	na	na	na	na

8.5.4.4 Interpolation process for the enhanced interpolation filter

8.5.4.4.1 General

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 picWidth,

— height of the picture in samples picHeight,

— clipMV flag specifying MV clipping type, and

— isLuma flag specifying whether luma or chroma is processed.

Output of this process is an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesLX of prediction sample values.

Interpolation filter coefficients T[p] for each fractional sample position p equal to xFrac or yFrac are specified in Table 28.

The variables horMax, verMax, horMin and verMin are derived by invoking the process as specifed in subclause 8.5.4.4.2 with 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, width of the picture in samples picWidth, height of the picture in samples picHeight and clipMV flag as inputs, and horMax, verMax, horMin and verMin as outputs.

When isLuma is equal to FALSE, then variables mvBaseScaled, horMin, horMax, verMin and verMax are modified as follows:

xCb = xCb / SubWidthC (939)

yCb = yCb / SubHeigthC (940)

cbWidth = cbWidth / SubWidthC (941)

cbHeight = cbHeight / SubHeightC (942)

mvBaseScaled[0] = mvBaseScaled[0] / SubWidthC (943)

mvBaseScaled[1] = mvBaseScaled[1] / SubHeightC (944)

horMin = horMin / SubWidthC (945)

horMax = horMax / SubWidthC (946)

verMin = verMin / SubHeightC (947)

verMax = verMax / SubHeightC (948)

The variables shift0, shift1, offset0 and offset1 are derived as follows:

— shift0 is set equal to bitDepth − 8, offset0 is equal to 0.

— shift1 is set equal to 12 − shift0, offset1 is equal to $2^{shift1 - 1}$.

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) >> 4 (949)

mvX[1] = (mvBaseScaled[1] + dX[1] * x + dY[1] * y) >> 4 (950)

mvX[0] = Clip3(horMin, horMax, mvX[0]) (951)

mvX[1] = Clip3(verMin, verMax, mvX[1]) (952)

— The variables xInt, yInt, xFrac and yFrac are derived as follows:

xInt = xCb + (mvX[0] >> 5) + x (953)

yInt = yCb + (mvX[1] >> 5) + y (954)

xFrac = mvX[0] & 31 (955)

yFrac = mvX[1] & 31 (956)

— The variables A and B are derived as follows:

A = (refPicLX[xInt][yInt] * T[xFrac][0] +
+ refPicLX[xInt + 1][yInt] * T[xFrac][1] + offset0) >> shift0 (957)

B = (refPicLX[xInt][yInt +1] * T[xFrac][0] +
+ refPicLX[xInt + 1][yInt + 1] * T[xFrac][1] + offset0) >> shift0 (958)

— The sample value $b_{x,y}$ corresponding to location (x, y) is derived as follows:

$b_{x,y}$ = (A * T[yFrac][0] + B * T[yFrac][1] + offset1) >> shift1 (959)

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 Max(bitDepth − 11, 0), offset2 is equal to $2^{shift2 - 1}$.

— shift3 is set equal to (6 − Max(bitDepth − 11, 0)), 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 (960)

For x = 0..cbWidth − 1 and y = 0..cbHeight − 1, the following applies:

$$predSamplesLX_L[\,x\,][\,y\,] = Clip3(\,0, (\,1 << bitDepth\,) - 1, (\,eF[\,0\,] * h_{x,y-1} + eF[\,1\,] * h_{x,y} + eF[\,2\,] * h_{x,y+1} + offset3\,) >> shift3) \quad (961)$$

Table 28 — Specification of the interpolation filter coefficients T[p] for each fractional sample position p

Fractional sample position p	Interpolation filter coefficients	
	T[p][0]	T[p][1]
0	64	0
1	62	2
2	60	4
3	58	6
4	56	8
5	54	10
6	52	12
7	50	14
8	48	16
9	46	18
10	44	20
11	42	22
12	40	24
13	38	26
14	36	28
15	34	30
16	32	32
17	30	34
18	28	36
19	26	38
20	24	40
21	22	42
22	20	44
23	18	46
24	16	48
25	14	50
26	12	52
27	10	54
28	8	56
29	6	58
30	4	60
31	2	62

8.5.4.4.2 Derivation of clipping parameters for affine motion vector

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,

— width of the picture in samples picWidth,

— height of the picture in samples picHeight, and

— clipMV flag specifying MV clipping type.

Outputs of this process are:

— horMax, verMax, horMin and verMin that denotes the maximum and minimum allowed motion vector horizontal and vertical components.

The variables log2CbW and log2CbH are derived as follows:

log2CbWidth = Log2(cbWidth) (962)

log2CbHeight = Log2(cbHeight) (963)

The variables horMaxPic, verMaxPic, horMinPic and verMinPic are derived as follows:

horMaxPic = (picWidth + 128 − xCb − cbWidth − 1) << 5 (964)

verMaxPic = (picHeight + 128 − yCb − cbHeight − 1) << 5 (965)

horMinPic = (−128 − xCb) << 5 (966)

verMinPic = (−128 − yCb) << 5 (967)

The centre motion vector mvCenter is derived as follows:

mvCenter[0] = (mvBaseScaled[0] + dX[0] * (cbWidth >> 1) + dY[0] * (cbHeight >> 1)) (968)

mvCenter[1] = (mvBaseScaled[1] + dX[1] * (cbWidth >> 1) + dY[1] * (cbHeight >> 1)) (969)

The rounding process for motion vectors as specified in subclause 8.5.3.10 is invoked with mvCenter, rightShift set equal to 4, and leftShift set equal to 0 as inputs, and the rounded motion vector is return as mvCenter.

If clipMV is equal to FALSE, then the output variables horMax, verMax, horMin and verMin that denotes the maximum and minimum allowed motion vector horizontal and vertical components are set equal to variables horMaxPic, verMaxPic, horMinPic and verMinPic, respectively.

Otherwise, the following steps are applied:

— The array deviationMV is set equal to { 128, 256, 544, 1120, 2272 }.

— The variables mvHorMin, mvVerMin, mvHorMax and mvVerMax are derived as follows:

$$horMin = mvCenter[0] - deviationMV[log2CbWidth - 3] \quad (970)$$

$$verMin = mvCenter[1] - deviationMV[log2CbHeight - 3] \quad (971)$$

$$horMax = mvCenter[0] + deviationMV[log2CbWidth - 3] \quad (972)$$

$$verMax = mvCenter[1] + deviationMV[log2CbHeight - 3] \quad (973)$$

— If horMin is less than horMinPic, the variables horMin and horMax are updated as follows:

$$horMin = horMinPic \quad (974)$$

$$horMax = Min(horMaxPic, horMinPic + 2* deviationMV[log2CbWidth - 3]) \quad (975)$$

— Otherwise, if horMax is greater than horMaxPic, the variables horMin and horMax are updated as follows:

$$horMin = Max(horMinPic, horMaxPic - 2* deviationMV[log2CbWidth - 3]) \quad (976)$$

$$horMax = horMaxPic \quad (977)$$

— If verMin is less than verMinPic, the variables verMin and verMax are updated as follows:

$$verMin = verMinPic \quad (978)$$

$$verMax = Min(verMaxPic, verMinPic + 2* deviationMV[log2CbHeight - 3]) \quad (979)$$

— Otherwise, if verMax is greater than verMaxPic, the variables verMin and verMax are updated as follows:

$$verMin = Max(verMinPic, verMaxPic - 2* deviationMV[log2CbHeight - 3]) \quad (980)$$

$$verMax = verMaxPic \quad (981)$$

— The output variables horMax, verMax, horMin and verMin are clipped as follows:

$$horMax = Clip3(-2^{17}, 2^{17} - 1, horMax) \quad (982)$$

$$verMax = Clip3(-2^{17}, 2^{17} - 1, verMax) \quad (983)$$

$$horMin = Clip3(-2^{17}, 2^{17} - 1, horMin) \quad (984)$$

$$verMin = Clip3(-2^{17}, 2^{17} - 1, verMin) \quad (985)$$

8.5.4.5 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, and

— 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:

$$pbSamples[x][y] = predSamplesL0[x][y] \quad (986)$$

— Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

$$pbSamples[x][y] = predSamplesL1[x][y] \quad (987)$$

— Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

$$pbSamples[x][y] = (predSamplesL0[x][y] + predSamplesL1[x][y] + 1) >> 1 \quad (988)$$

8.5.5 Decoder-side motion vector refinement process

8.5.5.1 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,

— the luma motion vectors in 1/4 fractional-sample accuracy mvL0 and mvL1, and

— the selected reference picture sample arrays $refPicL0_L$ and $refPicL1_L$.

Output of this process is the delta luma motion vector in 1/16 fractional sample accuracy dMvL0.

For X being replaced by either 0 or 1, motion vectors mvLtX are derived as conversion mvLX from 1/4 accuracy to 1/16 accuracy in subclause 8.5.2.9.

A variable srRange is set as 2. Variables offsetH[0], offsetV[0], offsetH[1] and offsetV[1] are set equal to 2. A variable fracPelAppliedflag is set equal to FALSE.

For X being replaced by either 0 or 1, motion vectors mvLsX are derived as follows:

$$mvLsX[0] = mvLtX[0] - 16 * srRange \quad (989)$$

mvLsX[1] = mvLtX[1] – 16 * srRange (990)

For X being replaced by either 0 or 1, the (sbWidth + 2 * srRange)x(sbHeight + 2 * srRange) array predSamplesLX$_L$ of prediction luma sample values is derived by invoking the fractional sample bilinear interpolation process as specified in subclause 8.5.5.2 with luma location (xSb, ySb), width of subblock (sbWidth + 2 * srRange), height of subblock (sbHeight + 2 * srRange), reference picture sample array refPicLX$_L$ and motion vector mvLsX as inputs, and predSamplesLX$_L$ as output.

The 9x1 array of sadVal is derived by invoking the sum of the absolute differences calculation process as specified in subclause 8.5.5.3 with sbWidth, sbHeight, predSamplesL0$_L$, predSamplesL1$_L$, offsetH and offsetV as inputs, and sadVal as output.

If sadVal[4] greater than or equal to sbWidth * sbHeight, then the following ordered steps are applied:

1) Array entry selection process as specified in subclause 8.5.5.4 is invoked with sadVal and n = 9 as input, and bestIdx as output.

2) If bestIdx is equal to 4, fracPelAppliedflag is set equal to TRUE.

   Otherwise, the following applies:

   dMvL0[0] = 16 * (bestIdx / 3 – 1) (991)

   dMvL0[1] = 16 * (bestIdx % 3 – 1) (992)

   offsetH[0] = offsetH[0] + (bestIdx / 3 – 1) (993)

   offsetV[0] = offsetV[0] + (bestIdx % 3 – 1) (994)

   offsetH[1] = offsetH[1] + (1 – bestIdx / 3) (995)

   offsetV[1] = offsetV[1] + (1 – bestIdx % 3) (996)

   — The 9x1 array of sadVal is modified by invoking the sum of the absolute differences calculation process as specified in subclause 8.5.5.3 with sbWidth, sbHeight, predSamplesL0$_L$, predSamplesL1$_L$, offsetH and offsetV as inputs, and modified sadVal as output.

   — Array entry selection process as specified in subclause 8.5.5.4 is invoked with sadVal as input, and bestIdx as output.

   — If bestIdx is equal to 4 and sadVal[4] is greater than 0, fracPelAppliedflag is set equal to TRUE.

   — dMvL0 is modified as follows:

     dMvL0[0] = dMvL0[0] + 16 * (bestIdx / 3 – 1) (997)

     dMvL0[1] = dMvL0[1] + 16 * (bestIdx % 3 – 1) (998)

3) If fracPelAppliedflag is TRUE, then parametric motion vector refinement process as specified in subclause 8.5.5.5 is invoked with sadVal and dMvL0 as inputs, and modified dMvL0 as output.

8.5.5.2 Fractional sample bilinear interpolation process

8.5.5.2.1 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 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, and

— the selected reference picture sample array $refPicLX_L$.

Output of this process is an (sbWidth)x(sbHeight) array $predSamplesLX_L$ of prediction luma sample values.

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 subclause for specifying fractional-sample locations inside the reference sample arrays $refPicLX_L$.

For each luma sample location (xL = 0..sbWidth − 1, yL = 0..sbHeight − 1) inside the prediction luma sample array $predSamplesLX_L$, the corresponding prediction luma sample value $predSamplesLX_L$[xL][yL] is derived as follows:

— The variables $xInt_L$, $yInt_L$, $xFrac_L$ and $yFrac_L$ are derived as follows:

$$xInt_L = xSb + (mvLX[0] >> 4) + x_L \quad (999)$$

$$yInt_L = ySb + (mvLX[1] >> 4) + y_L \quad (1000)$$

$$xFrac_L = mvLX[0] \& 15 \quad (1001)$$

$$yFrac_L = mvLX[1] \& 15 \quad (1002)$$

The luma prediction sample value $predSamplesLX_L$[x_L][y_L] is derived by invoking the process as specified in subclause 8.5.5.2.2 with ($xInt_L$, $yInt_L$), ($xFrac_L$, $yFrac_L$) and $refPicLX_L$ as inputs.

8.5.5.2.2 Luma sample bilinear 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 ($xFrac_L$, $yFrac_L$), and

— the luma reference sample array $refPicLX_L$.

Output of this process is a predicted luma sample value $predSampleLX_L$.

The variables shift0, shift1, shift2, shift3, shift4, offset1, offset2, offset3 and offset4 are derived as follows:

$$shift0 = 10 - BitDepth_Y \quad (1003)$$

$$shift1 = BitDepth_Y - 10 \quad (1004)$$

$$offset1 = 1 << (shift1 - 1) \quad (1005)$$

$$shift2 = BitDepth_Y - 4 \quad (1006)$$

$$offset2 = 1 << (shift2 - 1) \quad (1007)$$

$$shift3 = BitDepth_Y - 8 \quad (1008)$$

$$offset3 = 0 \quad (1009)$$

$$shift4 = 10 \quad (1010)$$

$$offset4 = 1 << 9 \quad (1011)$$

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 29.

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:

$$predSampleLX_L = BitDepth_Y <= 10 \ ? \ refPicLX_L[xInt_L][yInt_L] << shift0 : ((refPicLX_L[xInt_L][yInt_L] + offset1) >> shift1) \quad (1012)$$

— 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:

$$predSampleLX_L = (fb_L[xFrac_L][0] * refPicLX_L[Clip3(0, picW - 1, xInt_L)][yInt_L] + fb_L[xFrac_L][1] * refPicLX_L[Clip3(0, picW - 1, xInt_L + 1)][yInt_L] + offset2) >> shift2 \quad (1013)$$

— 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:

$$predSampleLX_L = (fb_L[yFrac_L][0] * refPicLX_L[xInt_L][Clip3(0, picH - 1, yInt_L)] + fb_L[yFrac_L][1] * refPicLX_L[xInt_L][Clip3(0, picH - 1, yInt_L + 1)] + offset2) >> shift2 \quad (1014)$$

— 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..1, is derived as follows:

$$yPos_L = Clip3(0, PicH - 1, yInt_L + n) \quad (1015)$$

$$temp[n] = (fb_L[xFrac_L][0] * refPicLX_L[Clip3(0, picW - 1, xInt_L)][yPos_L] + fb_L[xFrac_L][1] * refPicLX_L[Clip3(0, picW - 1, xInt_L + 1)][yPos_L] + offset3) >> shift3 \quad (1016)$$

  — The predicted luma sample value $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (fb_L[yFrac_L][0] * temp[0] + fb_L[yFrac_L][1] * temp[1] + offset4) >> shift4 \quad (1017)$$

Table 29 — Specification of the luma interpolation filter coefficients $fb_L[p]$ 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

8.5.5.3 Sum of absolute differences calculation process

Inputs to this process are:

— two variables nSbW and nSbH specifying the width and the height of the current coding subblock,

— two (nSbW + 4)x(nSbH + 4) array predSamplesL0 and predSamplesL1, and

— two 2x1 arrays offsetH and offsetV.

Outputs of this process is the (9) array sadValues of the sum of absolute differences.

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 with i = 0..8, sadValues[i] is derived based on predSampleL0, predSampleL1, bC, offsetH[0], offsetV[0], offsetH[1], and offsetV[1] as follows:

— Set initial value of sadValues[i] to 0.

— For each y with y = bC[1][i]..(nSbH + bC[1][i]), the SAD value sadValues[i] is accumulated as follows:

— For each x with x = bC[0][i]..(nSbW+bC[0][i]), the following applies:

$$sadValues[i] = sadValues[i] + Abs(predSamplesL0[x + offsetH[0]][y + offsetV[0]] - predSamplesL1[x - 2\times bC[0][i] + offsetH[1]][y - 2\times bC[1][i] + offsetV[1]]) \quad (1018)$$

8.5.5.4 Array entry selection process

Input to this process is:

— a 9x1 array sadVal.

Output of this process is bestIdx specifying an index.

The following steps are applied:

— If sadVal[1] < sadVal[7] and sadVal[3] < sadVal[5], the following applies:

— A variable idxVal is set equal to 0.

— If sadVal[1] < sadVal[3], a variable bestIdx is set equal to 1.

— Otherwise, bestIdx is set equal to 3.

— Otherwise, if sadVal[1] >= sadVal[7] and sadVal[3]< sadVal[5], the following applies:

— A variable idxVal is set equal to 6.

— If sadVal[7] < sadVal[3], bestIdx is set equal to 7.

— Otherwise, bestIdx is set equal to 3.

— Otherwise, if sadVal[1] < sadVal[7] and sadVal[3] >= sadVal[5], the following applies:

— A variable idxVal is set equal to 2.

— If sadVal[1] < sadVal[5], bestIdx is set equal to 1.

— Otherwise, bestIdx is set equal to 5.

— Otherwise, if sadVal[1] >= sadVal[7] and sadVal[3] >= sadVal[5], the following applies:

— A variable idxVal is set equal to 8.

— If sadVal[7] < sadVal[5], bestIdx is set equal to 7.

— Otherwise, bestIdx is set equal to 5.

— When sadVal[4] <= sadVal[bestIdx], bestIdx is set equal to 4.

— When sadVal[idxVal] < sadVal[bestIdx], bestIdx is set equal to idxVal.

8.5.5.5 Parametric motion vector refinement process

Inputs to this process are:

— the evaluated cost values in sadVal 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), and

— 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 dMvL0 and dMvL1:

— Compute delta motion vector, dMv, as follows:

```
if( sadVal[ 1 ] + sadVal[ 7 ] = = ( sadVal[ 4 ] << 1 ) )
    dMv[ 0 ] = 0                                                                    (1019)
else
    dMv[ 0 ] = ( ( sadVal[ 1 ] − sadVal[ 7 ] ) << 4 ) / ( 2 * ( sadVal[ 1 ] + sadVal[ 7 ] − ( sadVal[ 4 ] << 1 ) ) )

if( sadVal[ 3 ] + sadVal[ 5 ] = = ( sadVal[ 4 ] << 1 ) )
    dMv[ 1 ] = 0                                                                    (1020)
else
    dMv[ 1 ] = ( ( sadVal[ 3 ] − sadVal[ 5 ] ) << 4 ) / ( 2 * ( sadVal[ 3 ] + sadVal[ 5 ] − ( sadVal[ 4 ] << 1 ) ) )
```

— dMvL0 is modified as follows:

$$dMvL0[0] = dMvL0[0] + dMv[0] \quad (1021)$$

$$dMvL0[1] = dMvL0[1] + dMv[1] \quad (1022)$$

8.5.6 Decoding process for the residual signal of coding units coded in inter prediction mode

8.5.6.1 General

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, and

— a variable, isChromaPresent, specifying the chroma component presence indicator.

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 in this subclause,

— when isChromaPresent, 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 in this subclause, and

— when isChromaPresent, 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 in this subclause.

The variables $nCbW_L$ and $nCbH_L$ are set equal to 1 << log2CbWidth and 1 << log2CbHeight. When isChromaPresent, 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 isChromaPresent, 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[xCb][yCb], the following applies:

— If cbf_all[xCb][yCb] 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 isChromaPresent, all samples of the two ($nCbW_C$)x($nCbH_C$) arrays $resSamples_{Cb}$ and $resSamples_{Cr}$ are set equal to 0.

— Otherwise (cbf_all[xCb][yCb] is equal to 1), the following ordered steps apply:

1) The decoding process for luma residual blocks as specified in subclause 8.5.6.2 is invoked with the luma location (xCb, yCb), the luma location (0, 0), 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 isChromaPresent, the decoding process for chroma residual blocks as specified in subclause 8.5.6.3 is invoked with the luma location (xCb, yCb), the luma location (0, 0), 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 isChromaPresent, the decoding process for chroma residual blocks as specified in subclause 8.5.6.3 is invoked with the luma location (xCb, yCb), the luma location (0, 0), the variables log2TrafoWidth and log2TrafoHeight set equal to log2CbWidth and log2CbHeight, 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 $resSamples_{Cr}$.

8.5.6.2 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,

— a luma location (xTb, yTb) specifying the top-left samples of the current luma block relative to the top-left sample of the current luma coding block,

— 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, and

— an (nCbW)x(nCbH) array resSamples of luma residual samples.

Output of this process is a modified version of the (nCbW)x(nCbH) array resSamples of luma residual samples.

Depending on the variables log2TrafoWidth and log2TrafoHeight, the following applies:

— If both log2TrafoWidth and log2Trafoheight are greater than MaxTbLog2SizeY, the following ordered steps apply:

1) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamplesas inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + (1 << (log2TrafoWidth – 1)), yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

3) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb + (1 << (log2TrafoHeight – 1))), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

4) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + (1 << (log2TrafoWidth – 1)), yTb + (1 << (log2TrafoHeight – 1))), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise, if log2TrafoWidth is greater than MaxTbLog2SizeY, the following ordered steps apply:

1) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + (1 << (log2TrafoWidth – 1)), yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise, if log2TrafoHeight is greater than MaxTbLog2SizeY, the following ordered steps apply:

1) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable log2TrafoWidth set equal to log2TrafoWidth and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb + (1 << (log2TrafoHeight – 1))), the variable log2TrafoWidth set equal to log2TrafoWidth and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise (both log2TrafoWidth and log2TrafoHeight are less than or equal to MaxTbLog2SizeY), the following ordered steps apply:

1) The (nCbW)x(nCbH) residual sample array of the current coding block resSamples is set equal to 0.

2) Depending on the value of ats_cu_inter_flag[x0][y0], ats_cu_inter_quad_flag[x0][y0], ats_cu_inter_horizontal_flag[x0][y0], the following applies:

   — If ats_cu_inter_flag[x0][y0] is equal to 0, nTbW and nTbH are set equal to nCbW and nCbH.

   — Otherwise, if ats_cu_inter_flag[x0][y0] is equal to 1 and ats_cu_inter_horizontal_flag[x0][y0] is equal to 0, nTbW and nTbH are set equal to (nCbW >> (ats_cu_inter_quad_flag[x0][y0] + 1)) and nCbH.

   — Otherwise (both ats_cu_inter_flag[x0][y0] and ats_cu_inter_horizontal_flag[x0][y0] are equal to 1), nTbW and nTbH are set equal to nCbW and (nCbH >> (ats_cu_inter_quad_flag[x0][y0] + 1)).

3) Depending on the value of ats_cu_inter_flag[x0][y0] and ats_cu_inter_pos_flag[x0][y0], the following applies:

   — If ats_cu_inter_flag[x0][y0] is equal to 0, xTb and yTb are set equal to xCb and yCb.

   — Otherwise, if ats_cu_inter_flag[x0][y0] is equal to 1 and cu_inter_pos_flag[x0][y0] is equal to 0, xTb and yTb are set equal to xCb and yCb.

   — Otherwise (both ats_cu_inter_flag[x0][y0] and cu_inter_pos_flag[x0][y0] are equal to 1), xTb is set equal to (xCb + nCbW – nTbW) and yTb is set equal to (yCb + nCbH – nTbH).

4) The scaling and transformation process as specified in subclause 8.7.2 is invoked with the luma location (xTb, yTb), the variable cIdx set equal to 0, the transform width nTbW set equal to nTbW and the transform height nTbH set equal to nTbH as inputs, and the output is an (nTbW)x(nTbH) array transformBlock.

5) The (nCbW)x(nCbH) residual sample array of the current coding block resSamples is modified as follows:

$$resSamples[xTb + i, yTb + j] = transformBlock[i, j], \text{ with } i = 0..nTbW - 1, j = 0..nTbH - 1 \quad (1023)$$

8.5.6.3 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,

— a luma location (xTb, yTb) specifying the top-left samples of the current luma block relative to the top-left sample of the current luma coding block,

— 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, and

— an (nCbW)x(nCbH) array resSamples of chroma residual samples.

Output of this process is a modified version of the (nCbW)x(nCbH) array resSamples of chroma residual samples.

Depending on the variables log2TrafoWidth and log2TrafoHeight, the following applies:

— If both log2TrafoWidth and log2Trafoheight are greater than MaxTbLog2SizeY, the following ordered steps apply:

1) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + (1 << (log2TrafoWidth – 1)), yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

3) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb + (1 << (log2TrafoHeight – 1))), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

4) The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + (1 << (log2TrafoWidth – 1)), yTb + (1 << (log2TrafoHeight – 1))), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to

log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise, if log2TrafoWidth is greater than MaxTbLog2SizeY, the following ordered steps apply:

1) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb + (1 << (log2TrafoWidth – 1)), yTb), the variable log2TrafoWidth set equal to log2TrafoWidth – 1 and log2TrafoHeight set equal to log2TrafoHeight, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise, if log2TrafoHeight is greater than MaxTbLog2SizeY, the following ordered steps apply:

1) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb), the variable log2TrafoWidth set equal to log2TrafoWidth and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

2) The decoding process for chroma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb), the luma location (xTb, yTb + (1 << (log2TrafoHeight – 1))), the variable log2TrafoWidth set equal to log2TrafoWidth and log2TrafoHeight set equal to log2TrafoHeight – 1, the variable nCbW set equal to nCbW and the variable nCbH set equal to nCbH, and the (nCbW)x(nCbH) array resSamples as inputs, and the output is a modified version of the (nCbW)x(nCbH) array resSamples.

— Otherwise (both log2TrafoWidth and log2TrafoHeight are less than or equal to MaxTbLog2SizeY), the folowing ordered steps apply:

1) The (nCbW)x(nCbH) residual sample array of the current coding block resSamples is set equal to 0.

2) Depending on the value of ats_cu_inter_flag[x0][y0], ats_cu_inter_quad_flag[x0][y0], ats_cu_inter_horizontal_flag[x0][y0], the following applies:

— If ats_cu_inter_flag[x0][y0] is equal to 0, nTbW and nTbH are set equal to nCbW and nCbH.

— Otherwise, if ats_cu_inter_flag[x0][y0] is equal to 1 and ats_cu_inter_horizontal_flag[x0][y0] is equal to 0, nTbW and nTbH are set equal to (nCbW >> (ats_cu_inter_quad_flag[x0][y0] + 1)) and nCbH.

— Otherwise (both ats_cu_inter_flag[x0][y0] and ats_cu_inter_horizontal_flag[x0][y0] are equal to 1), nTbW and nTbH are set equal to nCbW and (nCbH >> (ats_cu_inter_quad_flag[x0][y0] + 1)).

3) Depending on the value of ats_cu_inter_flag[x0][y0] and ats_cu_inter_pos_flag[x0][y0], the following applies:

   — If ats_cu_inter_flag[x0][y0] is equal to 0, xTb and yTb are set equal to xCb and yCb.

   — Otherwise, if ats_cu_inter_flag[x0][y0] is equal to 1 and cu_inter_pos_flag[x0][y0] is equal to 0, xTb and yTb are set equal to xCb and yCb.

   — Otherwise (both ats_cu_inter_flag[x0][y0] and cu_inter_pos_flag[x0][y0] are equal to 1), xTb is set equal to (xCb + (nCbW – nTbW) * SubWidthC) and yTb is set equal to (yCb + (nCbH – nTbH) * SubHeightC).

4) The scaling and transformation process as specified in subclause 8.7.2 is invoked with the luma location (xTb, yTb), the variable cIdx, the transform width nTbW set equal to nTbW and the transform width nTbH set equal to nTbHas inputs, and the output is an (nTbW)x(nTbH) array transformBlock.

5) 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:

$$resSamples[xTb / SubWidthC + i, yTb / SubHeightC + j] = transformBlock[i, j] \quad (1024)$$

8.6 Decoding process for coding units coded in ibc prediction mode

8.6.1 General

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, and

— 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 in-loop filtering.

The derivation process for quantization parameters as specified in subclause 8.7.1 is invoked with the luma location (xCb, yCb) as input.

The variable isChromaPresent is set equal to TRUE, if treeType is equal to SINGLE_TREE and ChromaArrayType is not equal to 0; otherwise, the variable isChromaPresent is set equal to FALSE.

The variables $nCbW_L$ and $nCbH_L$ are set equal to 1 << log2CbWidth and 1 << log2CbHeight, respectively. If isChromaPresent, the variable $nCbW_C$ is set equal to $nCbW_L$ / SubWidthC and the variable $nCbH_C$ is set equal to $nCbH_L$ / SubHeightC.

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

1) The derivation process for motion vector as specified in subclause 8.6.2.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width $nCbW_L$, and the luma coding block height $nCbH_L$ as inputs, and the luma motion vector mvL as output.

2) If isChromaPresent, the derivation process for chroma motion vector as specified in subclause 8.6.2.2 is invoked with luma motion vector mvL as input, and chroma motion vector mvC as output.

3) The decoding process for inter sample prediction as specified in subclause 8.6.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width $nCbW_L$, the luma coding block height $nCbH_L$, the luma motion vector mvL, the chroma component presence indicator isChromaPresent, and, when isChromaPresent, 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 isChromaPresent, two $(nCbW_C)x(nCbH_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$ of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

4) The decoding process for the residual signal of coding units coded in inter prediction mode specified in subclause 8.5.6.1 is invoked with the luma location (xCb, yCb), the luma coding block width log2CbWidth, luma coding block height log2CbHeight and chroma component presence indicator isChromaPresent as inputs, and the array $resSamples_L$, and, when isChromaPresent, two arrays $resSamples_{Cb}$ and $resSamples_{Cr}$ as outputs.

5) 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 subclause 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, and the output is a modified reconstructed picture before in-loop filtering.

   — If isChromaPresent, the picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 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, and the output is a modified reconstructed picture before in-loop filtering.

   — If isChromaPresent, the picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 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, and the output is a modified reconstructed picture before in-loop filtering.

8.6.2 Derivation process for motion vector components

8.6.2.1 Derivation process for luma motion vector for ibc prediction 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, and

— two variables nCbW and nCbH specifying the width and the height of the current luma coding block.

Output of this process is the luma motion vector mvL.

The luma motion vector mvL is derived as follows:

1) The variable mvd is derived as follows:

$$mvd[0] = MvdL0[xCb][yCb][0] \quad (1025)$$

$$mvd[1] = MvdL0[xCb][yCb][1] \quad (1026)$$

2) The variables mvp[0] and mvp[1] are set equal to 0.

3) The luma motion vector mvL is derived as follows:

$$u[0] = (mvp[0] + mvd[0] + 2^{16}) \% 2^{16} \quad (1027)$$

$$mvL[0] = (u[0] >= 2^{15}) ? (u[0] - 2^{16}) : u[0] \quad (1028)$$

$$u[1] = (mvp[1] + mvd[1] + 2^{16}) \% 2^{16} \quad (1029)$$

$$mvL[1] = (u[1] >= 2^{15}) ? (u[1] - 2^{16}) : u[1] \quad (1030)$$

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:

$$(xRefTL, yRefTL) = (xCb + (mvL[0]), yCb + (mvL[1])) \quad (1031)$$

$$(xRefTR, yRefTR) = (xRefTL + nCbW - 1, yRefTL) \quad (1032)$$

$$(xRefBL, yRefBL) = (xRefTL, yRefTL + nCbH - 1) \quad (1033)$$

$$(xRefBR, yRefBR) = (xRefTL + nCbW - 1, yRefTL + nCbH - 1) \quad (1034)$$

It is a requirement of bitstream conformance that the luma motion vector mvL shall obey the following constraints:

— When the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xRefTL, yRefTL) as input, and the output shall be equal to TRUE.

— When the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xRefBR, yRefBR) as input, and the output shall be equal to TRUE.

— When sps_suco_flag is equal to 1, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xRefBL, yRefBL) as input, and the output shall be equal to TRUE.

— When sps_suco_flag is equal to 1, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xRefTL + width / 2, yRefBR) as input, and the output shall be equal to TRUE.

— At least one of the following conditions shall be true:

  — The value of mvL[0] + nCbW is less than or equal to 0.

  — The value of mvL[1] + nCbH 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 nCbW.

— The following conditions shall be true:

yRefTL >> CtbLog2SizeY = = yCb >> CtbLog2SizeY (1035)

yRefBL >> CtbLog2SizeY = = yCb >> CtbLog2SizeY (1036)

xRefTL >> CtbLog2SizeY >= (xCb >> CtbLog2SizeY) − 1 (1037)

xRefTR >> CtbLog2SizeY <= (xCb >> CtbLog2SizeY) (1038)

— When xRefTL >> CtbLog2SizeY is equal to (xCb >> CtbLog2SizeY) − 1, and CtbLog2SizeY is equal to 7, the follow conditions shall be true:

  — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location ((xRefTL + 128) / 64 * 64, yRefTL / 64 * 64) as input, 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 xRefTL >> CtbLog2SizeY is equal to (xCb >> CtbLog2SizeY) − 1, and CtbLog2SizeY is equal to 7, the follow conditions shall be true:

  — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location ((xRefTL + 128) / 64 * 64 + 63, yRefTL / 64 * 64) as input, 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 + nCbW − 1, yCb).

  — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location ((xRefTR + 128) / 64 * 64, yRefTR / 64 * 64) as input, and the output shall be equal to FALSE.

  — The luma location ((xRefTR + 128) / 64 * 64, yRefTR / 64 * 64) shall not be equal to (xCb + nCbW − 1, yCb).

  — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked the neighbouring luma location ((xRefTR + 128) / 64 * 64 + 63, yRefTR / 64 * 64) as input, 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 + nCbW − 1, yCb).

The following is applied:

mvL[0] = mvL[0] << 4, mvL[1] = mvL[1] << 4 (1039)

8.6.2.2 Derivation process for chroma motion vector for ibc prediction mode

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:

$$mvC[0] = ((mvL[0] >> (3 + SubWidthC)) * 32 \quad (1040)$$

$$mvC[1] = ((mvL[1] >> (3 + SubHeightC)) * 32 \quad (1041)$$

8.6.3 Decoding process for ibc blocks

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,

— chroma component presence indicator isChromaPresent, and

— when isChromaPresent 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 in this subclause,

— when isChromaPresent, 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 in this subclause,

— when isChromaPresent, 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 in this subclause.

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 isChromaPresent, $predSamples_{Cb}$ and $predSamples_{Cr}$ be (nCbW / SubWidthC)x(nCbH / SubHeightC) arrays of predicted chroma sample values.

The current decoded picture prior to in-loop filter process as specified in subclause 8.8, consists of a pic_width_in_luma_samples by pic_height_in_luma_samples array of luma samples $currPic_L$ and, when isChromaPresent, two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples $currPic_{Cb}$ and $currPic_{Cr}$.

The arrays $predSamples_L$ and, when isChromaPresent, $predSamples_{Cb}$, and $predSamples_{Cr}$ are derived by invoking the fractional sample interpolation process as specified in subclause 8.5.4.3.1 with the luma locations (xCb, yCb), the luma coding block width nCbW, the luma coding block height nCbH, the motion vector mvL[xCb][yCb], the luma motion vector offset mvOffset with both components equal to zero, the reference array $currPic_L$, the chroma component presence indicator isChromaPresent, and, when isChromaPresent, motion vector mvC[xCb][yCb], and the reference arrays $currPic_{Cb}$, and $currPic_{Cr}$ as inputs.

8.7 Scaling, transformation and array construction 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:

$$Qp_Y = (Qp_{Y\_PREV} + CuQpDelta[xCb][yCb] + 52) \% 52 \quad (1042)$$

where Qp_{Y_PREV} is the luma quantization parameter, Qp_Y of the previous coding unit in decoding order in the current tile. For the first coding unit in the tile, Qp_{Y_PREV} is initially set equal to slice_qp at the start of each tile.

The luma quantization parameter Qp'_Y is derived as follows:

$$Qp'_Y = Qp_Y + QpBdOffset_Y \quad (1043)$$

When ChromaArrayType is not equal to 0, the following applies:

— The variables Qp_{Cb} and Qp_{Cr} are derived as follows:

$$qPi_{Cb} = Clip3(-QpBdOffset_C, 57, Qp_Y + slice\_cb\_qp\_offset) \quad (1044)$$

$$qPi_{Cr} = Clip3(-QpBdOffset_C, 57, Qp_Y + slice\_cr\_qp\_offset) \quad (1045)$$

$$Qp_{Cb} = ChromaQpTable[0][qPi_{Cb}] \quad (1046)$$

$$Qp_{Cr} = ChromaQpTable[1][qPi_{Cr}] \quad (1047)$$

— The chroma quantization parameters for the Cb and Cr components, Qp'_{Cb} and Qp'_{Cr}, are derived as follows:

$$Qp'_{Cb} = Qp_{Cb} + QpBdOffset_C \quad (1048)$$

$$Qp'_{Cr} = Qp_{Cr} + QpBdOffset_C \quad (1049)$$

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, and

— 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.

$$qP = Qp'_{Y} \tag{1050}$$

— Otherwise, if cIdx is equal to 1, the following applies.

$$qP = Qp'_{Cb} \tag{1051}$$

— Otherwise (cIdx is equal to 2), the following applies.

$$qP = Qp'_{Cr} \tag{1052}$$

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 block location (xTbY, yTbY), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx, and the quantization parameter qP as inputs, and the output is an (nTbW)x(nTbH) array of scaled transform coefficients d.

2) 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 height nTbH, the colour component variable cIdx, and the (nTbW)x(nTbH) array of scaled transform coefficients d as inputs, and the output is an (nTbW)x(nTbH) array of residual samples r.

3) The variable bdShift is derived as follows:

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

   $$bdShift = ((cIdx == 0) ? 20 - BitDepth_{Y} : 20 - BitDepth_{C}) + 7 \tag{1053}$$

   — Otherwise (sps_iqt_flag is equal to 1), the following applies:

   $$bdShift = (cIdx == 0) ? 20 - BitDepth_{Y} : 20 - BitDepth_{C} \tag{1054}$$

4) The residual sample values r[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are modified as follows:

$$r[x][y] = (r[x][y] + (1 << (bdShift - 1))) >> bdShift \tag{1055}$$

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, and

— 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:

$$bdShift = BitDepth_Y + (((Log2(nTbW) + Log2(nTbH)) \& 1) * 8 + (Log2(nTbW) + Log2(nTbH)) / 2) - 5 \quad (1056)$$

— Otherwise, the following applies:

$$bdShift = BitDepth_C + (((Log2(nTbW) + Log2(nTbH)) \& 1) * 8 + (Log2(nTbW) + Log2(nTbH)) / 2) - 5 \quad (1057)$$

The variable rectNorm is derived as follows:

$$rectNorm = ((Log2(nTbW) + Log2(nTbH)) \& 1) == 1 \ ? \ 181 : 1 \quad (1058)$$

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 (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:

$$d[x][y] = Clip3(-32768, 32767, ((TransCoeffLevel[xTbY][yTbY][cIdx][x][y] * levelScale[qP\%6] << (qP / 6)) * rectNorm + (1 << (bdShift - 1))) >> bdShift) \quad (1059)$$

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, and

— 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 variables x0 and y0 are set equal to xTbY and yTbY.

The variables trTypeHor and trTypeVer are dervied as follows:

— If ats_cu_intra_flag[x0][y0] is equal to 1, trTypeHor and trTypeVer are specifed according to Table 30.

— Otherwise, if ats_cu_intra_flag[x0][y0] is equal to 0 and ats_cu_inter_flag[x0][y0] is equal to 1 and Max(nTbH, nTbW) is less than 64, trTypeHor and trTypeVer are specifed according to Table 31.

— Otherwise, trTypeHor and trTypeVer are set equal to 0.

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 to 0, otherwise 0 as inputs, and the output is the list e[x][y] with y = 0..nTbH − 1.

2) The following applies:

   — If sps_iqt_flag is equal to 1, the following applies:

   $$g[x][y] = Clip3(-32768, 32767, (e[x][y] + 64) >> 7) \quad (1060)$$

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

   $$g[x][y] = e[x][y] \quad (1061)$$

3) 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 to 0, otherwise 0 as inputs, and the output is the list r[x][y] with x = 0..nTbW − 1.

Table 30 — Specification of trTypeHor and trTypeVer depending on ats_hor_mode[x0][y0] and ats_ver_mode[x0][y0], when ats_cu_intra_flag[x0][y0] is equal to 1

ats_hor_mode[x0][y0]	0	1	0	1
ats_ver_mode[x0][y0]	0	0	1	1
trTypeHor	1	2	1	2
trTypeVer	1	1	2	2

Table 31 — Specification of trTypeHor and trTypeVer depending on ats_cu_inter_horizontal_flag[x0][y0] and ats_cu_inter_pos_flag[x0][y0], when Max(nTbH, nTbW) is less than 64 and ats_cu_inter_flag[x0][y0] is equal to 1

ats_cu_inter_horizontal_flag[x0][y0]	0	1	0	1
ats_cu_inter_pos_flag[x0][y0]	0	0	1	1
trTypeHor	2	1	1	1
trTypeVer	1	2	1	1

8.7.4.2 Transformation process

Inputs to this process are:

— a variable nTbS specifying the sample size of scaled transform coefficients, and

— 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 subclause 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.

The list of transformed samples y[i] with i = 0..nTbS − 1 is derived as follows:

$$y[\,i\,] = \sum_{j=0}^{nTbS-1} transMatrix[\,i\,][\,j\,] * x[\,j\,] \quad \text{with } i = 0..nTbS - 1 \tag{1062}$$

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, and

— 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] = (1063)

```
{
{  64,   64 }
{  64,  -64 }
}
```

— Otherwise, if trType is equal to 0 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (1064)

```
{
{  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] = (1065)

```
{
{  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] = (1066)

```
{
{  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 (1067)

transMatrixCol0to15 = (1068)

```
{
{  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 (1069)

transMatrixCol16to31 = (1070)

```
{
{  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 (1071)

transMatrixCol0to15 = (1072)

```
{
{  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 (1073)

transMatrixCol16to31 = (1074)

```
{
{  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] (1075)
with m = 32..47, n = 0..63

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol0to15[63 −m][n] (1076)
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] = (1077)

```
{
{  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] = (1078)

```
{
{  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] = (1079)

```
{
{   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 (1080)

transMatrixCol0to15 = (1081)

```
{
{   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 (1082)

transMatrixCol16to31 = (1083)

```
{
{  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] = (1084)

```
{
{  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] = (1085)

```
{
{  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] = (1086)

```
{
{  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 (1087)

transMatrixCol0to15 = (1088)

```
{
{  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 (1089)

transMatrixCol16to31 = (1090)

```
{
{  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, and

— an (nCbW)x(nCbH) array resSamples specifying the residual samples of the current block.

Output of this process is a modified reconstructed picture.

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 $Clip1_C$, and

— 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]) (1091)
, with i = 0..nCbW − 1, j = 0..nCbH − 1

IsCoded[xCurr + i][yCurr + j] = TRUE (1092)

8.7.6 Post-reconstruction filter process

8.7.6.1 General

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, and

— a luma quantization parameter Qp_Y of the current block.

The output of this process is the modified reconstructed picture sample array S_L.

The process is not applied if one of the following conditions is true:

— nCbW * nCbH < 64.

— Max(nCbW, nCbH) >= 128.

— Min(nCbW, nCbH) >= 32 & & LumaPredMode[xCb][yCb] != MODE_INTRA.

— Qp_Y <= 17.

An (nCbW + 2)x(nCbH + 2) array of padded reconstructed luma samples recSamplesPad is derived according to the padding process as specified in subclause 8.7.6.2, with location (xCb, yCb) specifying the top-left sample of the current luma block, size (nCbW, nCbH) of the current block and array S_L specifying the reconstructed picture sample array as inputs.

The accumulated filtered samples accFlt[x][y] are initialized to be equal to zero and then derived based on padded reconstructed luma samples recSamplesPad, with x = −1..nCbW − 1, y = −1..nCbH − 1 as follows:

— The input array for filtering inFilt including a current sample and three its neighbouring samples according to the scan template scanTmpl with i = 0..3 is derived as follows:

$$inFilt[i] = recSamplesPad[x + scanTmpl[i][1]][y + scanTmpl[i][0]] \quad (1093)$$

, where array scanTmpl is equal to { (0, 0), (0, 1), (1, 0), (1, 1) }

— The Hadamard spectrum components fHad are derived by performing Hadamard transform as follows:

$$fHad[0] = inFilt[0] + inFilt[2] + inFilt[1] + inFilt[3] \quad (1094)$$

$$fHad[1] = inFilt[0] + inFilt[2] - inFilt[1] - inFilt[3] \quad (1095)$$

$$fHad[2] = inFilt[0] - inFilt[2] + inFilt[1] - inFilt[3] \quad (1096)$$

$$fHad[3] = inFilt[0] - inFilt[2] - inFilt[1] + inFilt[3] \quad (1097)$$

— The filtered Hadamard spectrum components fHadFilt are derived using look-up table as follows with i = 1..3, where variables aTHR, tblShift and look-up table array bLUT are derived based on luma quantization parameter Qp_Y according to subclause 8.7.6.3:

— If $BitDepth_Y < 10$, the following applies:

$$bdShift = 10 - BitDepth_Y, \tag{1098}$$

$$fHadFilt[i] = \begin{cases} fHad[i]; \;\; Abs(fHad[i]) \ll bdShift >= aTHR \\ bLUT[(fHad[i] \ll bdShift + (1 \ll (tblShift - 1))) \gg tblShift] \gg bdShift ; \\ \quad fHad[i] > 0, Abs(fHad[i]) \ll bdShift < aTHR \\ -\left(bLUT\left[\left((- fHad[i]) \ll bdShift + \left(1 \ll (tblShift - 1)\right)\right) \gg tblShift\right] \gg bdShift\right); \\ \quad fHad[i] <= 0, Abs(fHad[i]) \ll bdShift < aTHR \end{cases}$$

— Otherwise, the following applies:

$$bdShift = BitDepth_Y - 10, \tag{1099}$$

$$fHadFilt[i] = \begin{cases} fHad[i]; \;\; Abs(fHad[i]) \gg bdShift >= aTHR \\ bLUT[(fHad[i] \gg bdShift + (1 \ll (tblShift - 1))) \gg tblShift] \ll bdShift ; \\ \quad fHad[i] > 0, Abs(fHad[i]) \gg bdShift < aTHR \\ -\left(bLUT\left[\left((- fHad[i]) \gg bdShift + \left(1 \ll (tblShift - 1)\right)\right) \gg tblShift\right] \ll bdShift\right); \\ \quad fHad[i] <= 0, Abs(fHad[i]) \gg bdShift < aTHR \end{cases}$$

— The filtered Hadamard spectrum component fHadFilt[0] is set equal to the Hadamard spectrum component fHad[0].

— The filtered samples invHadFilt are derived by performing inverse Hadamard transform for filtered spectrum components fHadFilt as follows:

$$invHadFilt[0] = fHadFilt[0] + fHadFilt[2] + fHadFilt[1] + fHadFilt[3] \tag{1100}$$

$$invHadFilt[1] = fHadFilt[0] + fHadFilt[2] - fHadFilt[1] - fHadFilt[3] \tag{1101}$$

$$invHadFilt[2] = fHadFilt[0] - fHadFilt[2] + fHadFilt[1] - fHadFilt[3] \tag{1102}$$

$$invHadFilt[3] = fHadFilt[0] - fHadFilt[2] - fHadFilt[1] + fHadFilt[3] \tag{1103}$$

— The filtered samples invHadFilt are accumulated in accumulation buffer accFlt according to the scan template scanTmpl with i = 0..3 as follows:

$$accFlt[x + scanTmpl[i][1]][y + scanTmpl[i][0]] \mathrel{+}= invHadFilt[i] \gg 2 \tag{1104}$$

The reconstructed picture sample array S_L is modified as follows for x = 0..nCbW – 1 and y = 0..nCbH – 1:

$$S_L[xCb + x][yCb + y] = Clip1_Y((accFlt[x][y] + 2) \gg 2) \tag{1105}$$

8.7.6.2 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, and

— an array recSamples specifying the reconstructed picture sample array.

The output of this process is (nCbW + 2)x(nCbH + 2) array of padded reconstructed luma samples of the current block recSamplesPad.

For x = −1..nCbW and y = −1..nCbH, recSamplesPad[x][y] is derived as follows:

— When 0 <= x <= nCbW − 1 and 0 <= y <= nCbH − 1, the recSamplesPad[x][y] is set equal to recSamples[xCb + x][yCb + y].

— Otherwise, the following applies:

  — The derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the neighbouring luma location (xNbY, yNbY) set equal to (xCb + x, yCb + y) as input, and the output is assigned to availableN.

  — The variable dx is set equal to 0 and variable dy is set equal to 0.

  — When x = = −1, and availableN is equal to FALSE, or constrained_intra_pred_flag is equal to 1 and LumaPredMode[xCb + x][yCb + y] is not equal to MODE_INTRA, dx = 1.

  — When x = = nCbW, and availableN is equal to FALSE, or constrained_intra_pred_flag is equal to 1 and LumaPredMode[xCb + x][yCb + y] is not equal to MODE_INTRA, dx = −1.

  — When y = = −1, and availableN is equal to FALSE, or constrained_intra_pred_flag is equal to 1 and LumaPredMode[xCb + x][yCb + y] is not equal to MODE_INTRA, dy = 1.

  — When y = = nCbH, and availableN is equal to FALSE, or constrained_intra_pred_flag is equal to 1 and LumaPredMode[xCb + x][yCb + y] is not equal to MODE_INTRA, dy = −1.

  — recSamplesPad[x][y] is set equal to recSamples[xCb + x + dx][yCb + y + dy].

8.7.6.3 Derivation process for post-reconstruction filter look-up table

Input to this process is a luma quantization parameter Qp_Y of the current block.

Outputs of this process are:

— a look-up table bLUT used for filtering of the block,

— a look-up table access threshold aTHR,

— 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 derived as follows:


```
if( LumaPredMode[ xCb ][ yCb ] = = MODE_INTER && nCbW = = nCbH && Min( nCbW, nCbH ) >= 32)
    qpIdx = Clip3( 0, 4, ( QpY − 28 + ( 1 << 2 ) ) >> 3 )                    (1106)
else
    qpIdx = Clip3( 0, 4, ( QpY − 20 + ( 1 << 2 ) ) >> 3 )
```

The look-up table bLUT used for filtering of the block is derived by selecting array from setOfLUT based on qpIdx:

```
bLUT = setOfLUT[ qpIdx ]                                                     (1107)
```

```
setOfLUT[ 5 ][ 16 ] =                                                        (1108)
        {
        { 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 derived as follows:

```
tblShift = tblThrLog2[ qpIdx ] − 4                                           (1109)
```

```
tblThrLog2[ 5 ] = { 6, 7, 7, 8, 8 }                                         (1110)
```

The look-up table access threshold aTHR is derived as follows:

```
aTHR = ( 1 << tblThrLog2[ qpIdx ] ) − ( 1 << tblShift )                      (1111)
```

8.8 In-loop filter process

8.8.1 General

This subclause 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) When slice_deblocking_filter_flag is equal to 1, the following applies:

   — If sps_addb_flag is equal to 0, the deblocking filter process as specified in subclause 8.8.2.1 is invoked with the reconstructed picture sample arrays S_L, and, when ChromaArrayType is not equal to 0, S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L, and, when ChromaArrayType is not equal to 0, S'_{Cb} and S'_{Cr} after application of deblocking filter as outputs.

   — Otherwise (sps_addb_flag is equal to 1), the advanced deblocking filter process as specified in subclause 8.8.3.1 is invoked with the reconstructed picture sample arrays S_L, and, when ChromaArrayType is not equal to 0, S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L, and, when ChromaArrayType is not equal to 0, S'_{Cb} and S'_{Cr} after application of deblocking filter as outputs.

   — The array S'_L, and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} are assigned to the array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.

2) When slice_alf_enabled_flag is equal to 1, or slice_alf_chroma_idc is larger than 0, the following applies:

— The adaptive loop filter process as specified in subclause 8.8.4.1 is invoked with the reconstructed picture sample arrays S_L, and, when ChromaArrayType is not equal to 0, S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L, and, when ChromaArrayType is not equal to 0, S'_{Cb} and S'_{Cr} after application of adaptive loop filter as outputs.

— The arrays S'_L, and, when ChromaArrayType is not equal to 0, S'_{Cb} and S'_{Cr} are assigned to the arrays S_L, and, when ChromaArrayType is not equal to 0, S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.

8.8.2 Deblocking filter process

8.8.2.1 General

This process is invoked when sps_addb_flag is equal to 0.

Inputs to this process are:

— the array $recPicture_L$ specifying the reconstructed picture sample array prior to deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the reconstructed picture sample array prior to deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the reconstructed picture sample array prior to deblocking for Cr.

Outputs of this process are:

— the array $recPicture_L$ specifying the modified reconstructed picture sample arrays after deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the modified reconstructed picture sample arrays after deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the modified reconstructed picture sample arrays after deblocking for Cr.

Filtering in the horizontal direction (across the vertical block edges in a picture) is conducted first. Then filtering in the vertical direction (across the horizontal block edges in a picture) 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.

NOTE Although the filtering process is specified on a picture basis in this document, the filtering process can be implemented on a coding unit basis with an equivalent result, provided the decoder properly accounts for the processing dependency order so as to produce the same output values.

The deblocking filter process is applied to all the block edges of a picture, except the edges that are at the boundary of the picture,the edges that coincide with tile bounaries when loop_filter_across_tiles_enabled_flag is equal to 0, and the edges that coincide with slice boundaries and slice on either side of the boundary is coded slice_deblocking_filter_flag equal to 0.

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 32.

Table 32 — Name of association to edgeType

edgeType	Name of edgeType
0 (vertical edge)	EDGE_VER
1 (horizontal edge)	EDGE_HOR

The deblocking is applied as follows:

— The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in subclause 8.8.2.2 with the variable edgeType set equal to EDGE_VER, the reconstructed picture prior to deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as inputs, and the modified reconstructed picture after deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— The horizonal edges are filtered by invoking the deblocking filter process for one direction as specified in subclause 8.8.2.2 with the variable edgeType set equal to EDGE_HOR, the modified reconstructed picture after deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as inputs, and the modified reconstructed picture after deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

8.8.2.2 Deblocking filter process for one direction

Inputs to this process are:

— the variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

— the array $recPicture_L$ specifying the reconstructed picture sample array prior to deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the reconstructed picture sample array prior to deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the reconstructed picture sample array prior to deblocking for Cr.

Outputs of this process are:

— the array $recPicture_L$ specifying the modified reconstructed picture sample arrays after deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the modified reconstructed picture sample arrays after deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the modified reconstructed picture sample arrays after deblocking for Cr.

For luma and chroma coding blocks in a picture, following process applies separately with colour component index cIdx set equal to 0 for luma and, when ChromaArrayType is not equal to 0, cIdx set equal to 1 or 2 for Cb and Cr chroma components, respectively, coding block width log2CbWidth and

coding block height log2CbHeight set equal to luma block width and height if cIdx is equal to 0 and set equal to chroma block width and height if cIdx is equal to 1 or 2.

For each coding block with width log2CbWidth, height log2CbHeight, 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 and a variable splitTH specifying block split threshold set equal to MaxTbLog2SizeY if cIdx is equal to 0 and set equal to (MaxTbLog2SizeY − 1) if cIdx is equal to 1 or 2, the deblocking process is conducted as follows:

— If edgeType is equal to EDGE_VER and log2CbWidth <= splitTH, the following applies:

  — The deblocking process of coding block boundary as specified in subclause 8.8.2.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width log2CbWidth, block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— Otherwise, if edgeType is equal to EDGE_VER and log2CbWidth > splitTH, the following applies:

  — The deblocking process of coding block boundary as specified in subclause 8.8.2.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width (log2CbWidth >> 1), block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$ as outputs.

  — The deblocking process of coding block boundary as specified in subclause 8.8.2.3 is invoked with luma location of top-left sample of the coding block (xCb + (1 << splitTH), yCb), block width (log2CbWidth >> 1), block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— Otherwise, if edgeType is equal to EDGE_HOR and log2CbHeight <= splitTH, the following applies:

  — The deblocking process of coding block boundary as specified in subclause 8.8.2.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width log2CbWidth, block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— Otherwise, if edgeType is equal to EDGE_HOR and log2CbHeight > splitTH, the following applies:

  — The deblocking process of coding block boundary as specified in subclause 8.8.2.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width log2CbWidth, block height (log2CbHeight >> 1), and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component

index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— The deblocking process of coding block boundary as specified in subclause 8.8.2.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb + (1 << splitTH)), block width log2CbWidth, block height (log2CbHeight >> 1), and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

8.8.2.3 Deblocking filter process of coding block boundary

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 log2CbWidth and log2CbHeight specifying the width and the height of the current coding block,

— the picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$,

— the edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

— luma motion vector arrays mvL0 and mvL1,

— reference indices arrays refIdxL0 and refIdxL1, and

— a variable cIdx specifying colour component index.

Outputs of this process are the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ after deblocking.

If cIdx is equal to 0, or ChromaArrayType is equal to 3, or ChromaArrayType is equal to 2 and edgeType is equal to EDGE_HOR, the filtering process for edges in the luma coding block, when cIdx is equal to 0, or chroma coding block, when cIdx is equal to 1 or 2, of the current coding unit consists of the following ordered steps:

— The variable recPicture is set equal to $recPicture_L$ when cIdx is equal to 0, or to $recPicture_{Cb}$ when cIdx is equal to 1, or to $recPicture_{Cr}$ when cIdx is equal to 2, respectively.

— The variable bitDepth is set equal to $BitDepth_Y$ when cIdx is equal to 0, or to $BitDepth_C$ when cIdx is equal to 1 or 2, respectively.

— The variable bS is set equal to bS_L when cIdx is equal to 0, or to bS_{Cb} when cIdx is equal to 1, or to bS_{Cr} when cIdx is equal to 2, respectively.

— The variable cBf is derived as follows:

  — If ChromaArrayType is equal to 3, and cIdx is equal to 1, the variable cBf is set equal to cbf_cb, otherwise, if ChromaArrayType is equal to 3, and cIdx is equal to 2, the variable cBf is set equal to cbf_cr.

— Otherwise, variable cBf is set equal to cbf_luma.

— The variable xCb′ is set equal to xCb / (cIdx ? SubWidthC : 1), yCb′ is set equal to yCb / (cIdx ? SubHeightC : 1).

— The variables xP_i, yP_j, xD_i, yD_j, xN and yN are derived as follows:

— xP_i is set equal to i, yP_j is set equal to j, xN is set equal to ((1 << log2CbWidth) – 1), and yN is set equal to ((1 << log2CbHeight) – 1).

— xD_i is set equal to ((xP_i >> 2) << 2), yD_j is set equal to ((yP_j >> 2) << 2), when cIdx is equal to 0 or ChromaArrayType is not equal to 2, otherwise, xD_i is set equal to ((xP_i >> 1) << 2), yD_j is set equal to ((yP_j >> 2) << 2) when ChromaArrayType is equal to 2 and cIdx is not equal to 0.

— The filtering samples in the luma coding block to be filtered are set with the following conditions:

— If edgeType is equal to EDGE_HOR, samples for xP_i with i = 0..xN and yP_j with j = 0 are set to be filtered.

— Otherwise, if edgeType is equal to EDGE_VER, samples for xP_i with i = 0 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:

— If edgeType is equal to EDGE_VER, p_0 is set equal to recPicture[xCb′ + xP_i – 1][yCb′ + yP_j] and q_0 is set equal to recPicture[xCb′ + xP_i][yCb′ + yP_j].

— Otherwise (edgeType is equal to EDGE_HOR), p_0 is set equal to recPicture[xCb′ + xP_i][yCb′ + yP_j – 1] and q_0 is set equal to recPicture[xCb′ + xP_i][yCb′ + yP_j].

2) The variable bS[xD_i][yD_j] is derived as follows:

— If the sample p_0 or q_0 is in the coding block of a coding unit coded with intra prediction mode, bS[xD_i][yD_j] is set equal to 0.

— Otherwise, if the sample p_0 or q_0 is in the coding block of a coding unit coded with cBf equal to 1, bS[xD_i][yD_j] is set equal to 1.

— Otherwise, if the sample p_0 or q_0 is in the coding block of a coding unit coded with ibc prediction mode, bS[xD_i][yD_j] is set equal to 2.

— Otherwise, the following applies:

— The variables mv0L0x, mv0L0y, mv0L1x, mv0L1y, mv1L0x, mv1L0y, mv1L1x, mv1L1y, refIdx0L0, refIdx0L1, refIdx1L0, and refIdx1L1 are derived as follows:

— If edgeType is equal to EDGE_VER, the following applies:

$$mv0L0x = mvL0[xCb + xD_i][yCb + yD_j][0] \quad (1112)$$

$$mv0L0y = mvL0[xCb + xD_i][yCb + yD_j][1] \quad (1113)$$

$$mv0L1x = mvL1[xCb + xD_i][yCb + yD_j][0] \quad (1114)$$

$$mv0L1y = mvL1[xCb + xD_i][yCb + yD_j][1] \quad (1115)$$

$$mv1L0x = mvL0[xCb + xD_i - 1][yCb + yD_j][0] \quad (1116)$$

$$mv1L0y = mvL0[xCb + xD_i - 1][yCb + yD_j][1] \quad (1117)$$

$$mv1L1x = mvL1[xCb + xD_i - 1][yCb + yD_j][0] \quad (1118)$$

$$mv1L1y = mvL1[xCb + xD_i - 1][yCb + yD_j][1] \quad (1119)$$

$$refIdx0L0 = refIdxL0[xCb + xD_i][yCb + yD_j] \quad (1120)$$

$$refIdx0L1 = refIdxL1[xCb + xD_i][yCb + yD_j] \quad (1121)$$

$$refIdx1L0 = refIdxL0[xCb + xD_i - 1][yCb + yD_j] \quad (1122)$$

$$refIdx1L1 = refIdxL1[xCb + xD_i - 1][yCb + yD_j] \quad (1123)$$

— Otherwise, if edgeType is equal to EDGE_HOR, the following applies:

$$mv0L0x = mvL0[xCb + xD_i][yCb + yD_j][0] \quad (1124)$$

$$mv0L0y = mvL0[xCb + xD_i][yCb + yD_j][1] \quad (1125)$$

$$mv0L1x = mvL1[xCb + xD_i][yCb + yD_j][0] \quad (1126)$$

$$mv0L1y = mvL1[xCb + xD_i][yCb + yD_j][1] \quad (1127)$$

$$mv1L0x = mvL0[xCb + xD_i][yCb + yD_j - 1][0] \quad (1128)$$

$$mv1L0y = mvL0[xCb + xD_i][yCb + yD_j - 1][1] \quad (1129)$$

$$mv1L1x = mvL1[xCb + xD_i][yCb + yD_j - 1][0] \quad (1130)$$

$$mv1L1y = mvL1[xCb + xD_i][yCb + yD_j - 1][1] \quad (1131)$$

$$refIdx0L0 = refIdxL0[xCb + xD_i][yCb + yD_j] \quad (1132)$$

$$refIdx0L1 = refIdxL1[xCb + xD_i][yCb + yD_j] \quad (1133)$$

$$refIdx1L0 = refIdxL0[xCb + xD_i][yCb + yD_j - 1] \quad (1134)$$

$$refIdx1L1 = refIdxL1[xCb + xD_i][yCb + yD_j - 1] \quad (1135)$$

— When refIdx0L0 is unavailable, mv0L0x and mv0L0y are set equal to 0.

— When refIdx0L1 is unavailable, mv0L1x and mv0L1y are set equal to 0.

— When refIdx1L0 is unavailable, mv1L0x and mv1L0y are set equal to 0.

— When refIdx1L1 is unavailable, mv1L1x and mv1L1y are set equal to 0.

— If refIdx0L0 is equal to refIdx0L1 and refIdx1L0 is equal to refIdx1L1, the following applies:

$$bS[xD_i][yD_j] = (Abs(mv0L0x - mv1L0x) >= 4 \,||\, Abs(mv0L0y - mv1L0y) >= 4 \,||\, Abs(mv0L1x - mv1L1x) >= 4 \,||\, Abs(mv0L1y - mv1L1y) >= 4) \,?\, 2 : 3 \quad (1136)$$

— Otherwise, if refIdx0L0 is equal to refIdx1L1 and refIdx0L1 is equal to refIdx1L0, the following applies:

$$bS[xD_i][yD_j] = (Abs(mv0L0x - mv1L1x) >= 4 \,||\, Abs(mv0L0y - mv1L1y) >= 4 \,||\, Abs(mv0L1x - mv1L0x) >= 4 \,||\, Abs(mv0L1y - mv1L0y) >= 4) \,?\, 2 : 3 \quad (1137)$$

— Otherwise, the following applies:

$$bS[xD_i][yD_j] = 2 \quad (1138)$$

3) The variable qP is derived as follows:

— If cIdx is equal to 0, qP is set equal to Qp_Y that is used in the CU where sample q_0 is located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— Otherwise, if cIdx is equal to 1, qP is set equal to Qp_{Cb} that are used in the CU where sample q_0 is located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— Otherwise, qP is set equal to Qp_{Cr} that are used in the CU where sample q_0 is located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— If cIdx is not equal to 0, variable qP is updated as follows:

$$qP = Clip3(0, 51, qP) \quad (1139)$$

4) The variable $sT[xD_i][yD_j]$ and $sT'[xD_i][yD_j]$ are derived as follows:

— $sT[xD_i][yD_j]$ is derived from Table 33, depending on qP.

— $sT'[xD_i][yD_j]$ is equal to $sT[xD_i][yD_j] << (bitDepth - 8)$.

5) When $sT'[xD_i][yD_j]$ is greater than 0, the following ordered steps apply:

— If edgeType is equal to EDGE_VER, the sample values sA, sB, sC and sD are derived as follows:

$$sA = recPicture[xCb' + xP_i - 2][yCb' + yP_j] \quad (1140)$$

$$sB = recPicture[xCb' + xP_i - 1][yCb' + yP_j] \quad (1141)$$

$$sC = recPicture[xCb' + xP_i][yCb' + yP_j] \quad (1142)$$

$$sD = recPicture[xCb' + xP_i + 1][yCb' + yP_j] \quad (1143)$$

— Otherwise (edgeType is equal to EDGE_HOR), the sample values sA, sB, sC and sD are derived as follows:

$$sA = recPicture[xCb' + xP_i][yCb' + yP_j - 2] \quad (1144)$$

$$sB = recPicture[xCb' + xP_i][yCb' + yP_j - 1] \quad (1145)$$

$$sC = recPicture[xCb' + xP_i][yCb' + yP_j] \quad (1146)$$

$$sD = recPicture[xCb' + xP_i][yCb' + yP_j + 1] \quad (1147)$$

— The variables iD, absD, iTmp, clipD, iD1, and iD2 are derived as follows:

$$iD = (sA - (sB << 2) + (sC << 2) - sD) / 8 \quad (1148)$$

$$absD = Abs(iD) \quad (1149)$$

$$iTmp = Max(0, (absD - sT'[xD_i][yD_j]) << 1) \quad (1150)$$

$$clipD = Max(0, absD - iTmp) \quad (1151)$$

$$iD1 = Sign(iD) ? -clipD : clipD \quad (1152)$$

$$clipD = clipD >> 1 \quad (1153)$$

$$iD2 = Clip3(-clipD, clipD, (sA - sD) / 4) \quad (1154)$$

$$sA -= iD2 \quad (1155)$$

$$sB += iD1 \quad (1156)$$

$$sC -= iD1 \quad (1157)$$

$$sD += iD2 \quad (1158)$$

6) The modified picture sample array recPicture is derived as follows:

— If edgeType is equal to EDGE_VER, recPicture is derived as follows:

$$recPicture[xCb' + xP_i - 2][yCb' + yP_j] = Clip3(0, (1 << bitDepth) - 1, sA) \quad (1159)$$

$$recPicture[xCb' + xP_i - 1][yCb' + yP_j] = Clip3(0, (1 << bitDepth) - 1, sB) \quad (1160)$$

$$recPicture[xCb' + xP_i][yCb' + yP_j] = Clip3(0, (1 << bitDepth) - 1, sC) \quad (1161)$$

$$recPicture[xCb' + xP_i + 1][yCb' + yP_j] = Clip3(0, (1 << bitDepth) - 1, sD) \quad (1162)$$

— Otherwise (edgeType is equal to EDGE_HOR), $recPicture_L$ is derived as follows:

$$recPicture[xCb' + xP_i][yCb' + yP_j - 2] = Clip3(0, (1 << bitDepth) - 1, sA) \quad (1163)$$

$$recPicture[xCb' + xP_i][yCb' + yP_j - 1] = Clip3(0, (1 << bitDepth) - 1, sB) \quad (1164)$$

$$recPicture[xCb' + xP_i][yCb' + yP_j] = Clip3(0, (1 << bitDepth) - 1, sC) \quad (1165)$$

$$recPicture[xCb' + xP_i][yCb' + yP_j + 1] = Clip3(0, (1 << bitDepth) - 1, sD) \quad (1166)$$

Otherwise, the filtering process for edges in the chroma coding blocks consists of the following ordered steps:

— The variable $recPicture_X$ is set equal to $recPicture_{Cb}$ when cIdx is equal to 1, or to $recPicture_{Cr}$ when cIdx is equal to 2.

— The variable bS_X is set equal to bS_{Cb} when cIdx is equal to 1, or to bS_{Cr} when cIdx is equal to 2.

— The variable xCb′ is set equal to xCb / SubWidthC, yCb′ is set equal to yCb / SubHeightC.

— The variables xP_i, yP_j, xD_i, yD_j, xN and yN are derived as follows:

  — xP_i is set equal to i, yP_j is set equal to j, xN is set equal to (1 << log2CbWidth) − 1, and yN is set equal to (1 << log2CbHeight) − 1.

  — xD_i is set equal to ((xP_i >> 1) << 2), yD_j is set equal to ((yP_j >> 1) << 2), when cIdx is equal to 0 or ChromaArrayType is equal to 1, otherwise, xD_i is set equal to ((xP_i >> 1) << 2), yD_j is set equal to ((yP_j >> 2) << 2) when ChromaArrayType is equal to 2.

— The filtering samples in the chroma coding block to be filtered are set with the following conditions:

  — If edgeType is equal to EDGE_HOR, samples for xP_i with i = 0..xN and yP_j with j = 0 are set to be filtered.

  — Otherwise, if edgeType is equal to EDGE_VER, samples for xP_i with i = 0 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:

    — If edgeType is equal to EDGE_VER, p_0 is set equal to $recPicture_X$[xCb′ + xP_i − 1][yCb′ + yP_j] and q_0 is set equal to $recPicture_X$[xCb′ + xP_i][yCb′ + yP_j].

    — Otherwise (edgeType is equal to EDGE_HOR), p_0 is set equal to $recPicture_X$[xCb′ + xP_i][yCb′ + yP_j − 1] and q_0 is set equal to $recPicture_X$[xCb′ + xP_i][yCb′ + yP_j].

  2) The variable bS_X[xD_i][yD_j] is derived as follows:

    — If the sample p_0 or q_0 is in the coding block coded with intra prediction mode, bS_X[xD_i][yD_j] is set equal to 0.

    — Otherwise, if the sample p_0 or q_0 is in the coding block coded with cbf_luma equal to 1, bS_X[xD_i][yD_j] is set equal to 1.

    — Otherwise, if the sample p_0 or q_0 is in the coding block coded with ibc prediction mode, bS_X[xD_i][yD_j] is set equal to 2.

    — Otherwise, the following applies:

      — The variables mv0L0x, mv0L0y, mv0L1x, mv0L1y, mv1L0x, mv1L0y, mv1L1x, mv1L1y, refIdx0L0, refIdx0L1, refIdx1L0, and refIdx1L1 are derived as follows:

        — If edgeType is equal to EDGE_VER, the following applies:

$$mv0L0x = mvL0[xCb + xD_i][yCb + yD_j][0] \quad (1167)$$

$$mv0L0y = mvL0[xCb + xD_i][yCb + yD_j][1] \quad (1168)$$

$$mv0L1x = mvL1[\ xCb + xD_i\ ][\ yCb + yD_j\ ][\ 0\ ] \quad (1169)$$

$$mv0L1y = mvL1[\ xCb + xD_i\ ][\ yCb + yD_j\ ][\ 1\ ] \quad (1170)$$

$$mv1L0x = mvL0[\ xCb + xD_i - 1\ ][\ yCb + yD_j\ ][\ 0\ ] \quad (1171)$$

$$mv1L0y = mvL0[\ xCb + xD_i - 1\ ][\ yCb + yD_j\ ][\ 1\ ] \quad (1172)$$

$$mv1L1x = mvL1[\ xCb + xD_i - 1\ ][\ yCb + yD_j\ ][\ 0\ ] \quad (1173)$$

$$mv1L1y = mvL1[\ xCb + xD_i - 1\ ][\ yCb + yD_j\ ][\ 1\ ] \quad (1174)$$

$$refIdx0L0 = refIdxL0[\ xCb + xD_i\ ][\ yCb + yD_j\ ] \quad (1175)$$

$$refIdx0L1 = refIdxL1[\ xCb + xD_i\ ][\ yCb + yD_j\ ] \quad (1176)$$

$$refIdx1L0 = refIdxL0[\ xCb + xD_i - 1\ ][\ yCb + yD_j\ ] \quad (1177)$$

$$refIdx1L1 = refIdxL1[\ xCb + xD_i - 1\ ][\ yCb + yD_j\ ] \quad (1178)$$

— Otherwise, if edgeType is equal to EDGE_HOR, the following applies:

$$mv0L0x = mvL0[\ xCb + xD_i\ ][\ yCb + yD_j\ ][\ 0\ ] \quad (1179)$$

$$mv0L0y = mvL0[\ xCb + xD_i\ ][\ yCb + yD_j\ ][\ 1\ ] \quad (1180)$$

$$mv0L1x = mvL1[\ xCb + xD_i\ ][\ yCb + yD_j\ ][\ 0\ ] \quad (1181)$$

$$mv0L1y = mvL1[\ xCb + xD_i\ ][\ yCb + yD_j\ ][\ 1\ ] \quad (1182)$$

$$mv1L0x = mvL0[\ xCb + xD_i\ ][\ yCb + yD_j - 1\ ][\ 0\ ] \quad (1183)$$

$$mv1L0y = mvL0[\ xCb + xD_i\ ][\ yCb + yD_j - 1\ ][\ 1\ ] \quad (1184)$$

$$mv1L1x = mvL1[\ xCb + xD_i\ ][\ yCb + yD_j - 1\ ][\ 0\ ] \quad (1185)$$

$$mv1L1y = mvL1[\ xCb + xD_i\ ][\ yCb + yD_j - 1\ ][\ 1\ ] \quad (1186)$$

$$refIdx0L0 = refIdxL0[\ xCb + xD_i\ ][\ yCb + yD_j\ ] \quad (1187)$$

$$refIdx0L1 = refIdxL1[\ xCb + xD_i\ ][\ yCb + yD_j\ ] \quad (1188)$$

$$refIdx1L0 = refIdxL0[\ xCb + xD_i\ ][\ yCb + yD_j - 1\ ] \quad (1189)$$

$$refIdx1L1 = refIdxL1[\ xCb + xD_i\ ][\ yCb + yD_j - 1\ ] \quad (1190)$$

— When refIdx0L0 is unavailable, mv0L0x and mv0L0y are set equal to 0.

— When refIdx0L1 is unavailable, mv0L1x and mv0L1y are set equal to 0.

— When refIdx1L0 is unavailable, mv1L0x and mv1L0y are set equal to 0.

— When refIdx1L1 is unavailable, mv1L1x and mv1L1y are set equal to 0.

— If refIdx0L0 is equal to refIdx0L1 and refIdx1L0 is equal to refIdx1L1, the following applies:

$$bS_X[\,xD_i\,][\,yD_j\,] = (\,Abs(\,mv0L0x - mv1L0x\,) >= 4\;||\;Abs(\,mv0L0y - mv1L0y\,) >= 4\;||\;Abs(\,mv0L1x - mv1L1x\,) >= 4\;||\;Abs(\,mv0L1y - mv1L1y\,) >= 4\,)\;?\;2 : 3 \quad (1191)$$

— Otherwise, if refIdx0L0 is equal to refIdx1L1 and refIdx0L1 is equal to refIdx1L0, the following applies:

$$bS_X[\,xD_i\,][\,yD_j\,] = (\,Abs(\,mv0L0x - mv1L1x\,) >= 4\;||\;Abs(\,mv0L0y - mv1L1y\,) >= 4\;||\;Abs(\,mv0L1x - mv1L1x\,) >= 4\;||\;Abs(\,mv0L1y - mv1L0y\,) >= 4\,)\;?\;2 : 3 \quad (1192)$$

— Otherwise, the following applies:

$$bS_X[\,xD_i\,][\,yD_j\,] = 2 \quad (1193)$$

3) The variable $sT[\,xD_i\,][\,yD_j\,]$ and $sT'[\,xD_i\,][\,yD_j\,]$ are derived as follows:

— If cIdx is equal to 1, Qp_C is set equal to Qp_{Cb} that are used in the CU where sample are located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— Otherwise, Qp_C is set equal to Qp_{Cr} that are used in the CU where sample are located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— $clippedQp_C$ value is derived as follows:

$$clippedQp_C = Clip3(\,0, 51, Qp_C\,) \quad (1194)$$

— $sT[\,xD_i\,][\,yD_j\,]$ depending on $clippedQp_C$ is derived from Table 33.

— $sT'[\,xD_i\,][\,yD_j\,]$ is equal to $sT[\,xD_i\,][\,yD_j\,] << (\,BitDepth_C - 8\,)$.

4) When $sT'[\,xD_i\,][\,yD_j\,]$ is greater than 0, the following ordered steps apply:

— If edgeType is equal to EDGE_VER, the sample values sA, sB, sC, and sD are derived as follows:

$$sA = recPicture_X[\,xCb' + xP_i - 2\,][\,yCb' + yP_j\,] \quad (1195)$$

$$sB = recPicture_X[\,xCb' + xP_i - 1\,][\,yCb' + yP_j\,] \quad (1196)$$

$$sC = recPicture_X[\,xCb' + xP_i\,][\,yCb' + yP_j\,] \quad (1197)$$

$$sD = recPicture_X[\,xCb' + xP_i + 1\,][\,yCb' + yP_j\,] \quad (1198)$$

— Otherwise (edgeType is equal to EDGE_HOR), the sample values sA, sB, sC, and sD are derived as follows:

$$sA = recPicture_X[\,xCb' + xP_i\,][\,yCb' + yP_j - 2\,] \quad (1199)$$

$$sB = recPicture_X[\,xCb' + xP_i\,][\,yCb' + yP_j - 1\,] \quad (1200)$$

$$sC = recPicture_X[\,xCb' + xP_i\,][\,yCb' + yP_j\,] \quad (1201)$$

$$sD = recPicture_X[\ xCb' + xP_i\ ][\ yCb' + yP_j + 1\ ] \quad (1202)$$

— The variables iD, absD, iTmp, clipD, iD1, and iD2 are derived as follows:

$$iD = (\ sA - (\ sB << 2\ ) + (\ sC << 2\ ) - sD\ ) / 8 \quad (1203)$$

$$absD = Abs(\ iD\ ) \quad (1204)$$

$$iTmp = Max(\ 0, (\ absD - sT'[\ xD_i\ ][\ yD_j\ ]\ ) << 1\ ) \quad (1205)$$

$$clipD = Max(\ 0, absD - iTmp\ ) \quad (1206)$$

$$iD1 = Sign(\ iD\ )\ ?\ -clipD : clipD \quad (1207)$$

$$sB\ +=\ iD1 \quad (1208)$$

$$sC\ -=\ iD1 \quad (1209)$$

5) The modified chroma picture sample array $recPicture_X$ is derived as follows:

— If edgeType is equal to EDGE_VER, $recPicture_X$ is derived as follows:

$$recPicture_X[\ xCb' + xP_i - 1\ ][\ yCb' + yP_j\ ] = Clip3(\ 0, (\ 1 << BitDepth_C\ ) - 1, sB\ ) \quad (1210)$$

$$recPicture_X[\ xCb' + xP_i\ ][\ yCb' + yP_j\ ] = Clip3(\ 0, (\ 1 << BitDepth_C\ ) - 1, sC\ ) \quad (1211)$$

— Otherwise (edgeType is equal to EDGE_HOR), $recPicture_X$ is derived as follows:

$$recPicture_X[\ xCb' + xP_i\ ][\ yCb' + yP_j - 1\ ] = Clip3(\ 0, (\ 1 << BitDepth_C\ ) - 1, sB\ ) \quad (1212)$$

$$recPicture_X[\ xCb' + xP_i\ ][\ yCb' + yP_j\ ] = Clip3(\ 0, (\ 1 << BitDepth_C\ ) - 1, sC\ ) \quad (1213)$$

Table 33 — Derivation of $sT[\ xD_i\ ][\ yD_j\ ]$ from QP and $bS_X[\ xD_i\ ][\ yD_j\ ]$ where X is colour component

QP	$bS_X[\ xD_i\ ][\ yD_j\ ] = 1$	$bS_X[\ xD_i\ ][\ yD_j\ ] = 2$	$bS_X[\ xD_i\ ][\ yD_j\ ] = 3$
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

QP	$bS_X[xD_i][yD_j] = 1$	$bS_X[xD_i][yD_j] = 2$	$bS_X[xD_i][yD_j] = 3$
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 Advanced deblocking filter process

8.8.3.1 General

This process is invoked when sps_addb_flag is equal to 1.

Inputs to this process are:

— the array $recPicture_L$ specifying the reconstructed picture sample array prior to deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the reconstructed picture sample array prior to deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the reconstructed picture sample array prior to deblocking for Cr.

Outputs of this process are:

— the array $recPicture_L$ specifying the modified reconstructed picture sample arrays after deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the modified reconstructed picture sample arrays after deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the modified reconstructed picture sample arrays after deblocking for Cr.

Filtering in the horizontal direction (across the vertical block edges in a picture) is conducted first. Then filtering in the vertical direction (across the horizontal block edges in a picture) is conducted with samples modified by the filtering in the horizontal direction as input. The vertical and horizontal edges in the coding tree blocks of each coding tree unit are processed separately on a coding unit basis. The vertical edges of the coding blocks in a coding unit are filtered starting with the edge on the left-hand side of the coding blocks proceeding through the edges towards the right-hand side of the coding blocks in their geometrical order. The horizontal edges of the coding blocks in a coding unit are filtered starting with the edge on the top of the coding blocks proceeding through the edges towards the bottom of the coding blocks in their geometrical order.

NOTE Although the filtering process is specified on a picture basis in this document, the filtering process can be implemented on a coding unit basis with an equivalent result, provided the decoder properly accounts for the processing dependency order so as to produce the same output values.

The deblocking filter process is applied to all the block edges of a picture, except of the following conditions:

— The edges that are at the boundary of the picture and luma and chroma edges that do not correspond to 8x8 sample grid boundaries of the corresponded luma component,

— The edges that belong to the slice for which the deblocking filter process is disabled by slice_deblocking_filter_flag,

— The edges that correspond to tiles boundaries if syntax element loop_filter_across_tiles_enabled_flag is equal to 0,

— The edges that correspond to the slice boundaries and if the neighbouring samples, comprising to the filtering process, belongs to the neighbouring slice 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 32.

The deblocking is applied as follows:

— The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in subclause 8.8.3.2 with the variable edgeType set equal to EDGE_VER, the reconstructed picture prior to deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$, and $recPicture_{Cr}$ as inputs, and the modified reconstructed picture after deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$, and $recPicture_{Cr}$ as outputs.

— The horizonal edges are filtered by invoking the deblocking filter process for one direction as specified in subclause 8.8.3.2 with the variable edgeType set equal to EDGE_HOR, the modified reconstructed picture after deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$, and $recPicture_{Cr}$ as inputs, and the modified reconstructed picture after deblocking, i.e., $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$, and $recPicture_{Cr}$ as outputs.

8.8.3.2 Deblocking filter process for one direction

Inputs to this process are:

— the variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

— the array $recPicture_L$ specifying the reconstructed picture sample array prior to deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the reconstructed picture sample array prior to deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the reconstructed picture sample array prior to deblocking for Cr.

Outputs of this process are:

— the array $recPicture_L$ specifying the modified reconstructed picture sample arrays after deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the modified reconstructed picture sample arrays after deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the modified reconstructed picture sample arrays after deblocking for Cr.

For luma and chroma coding blocks in a picture, following process applies separately with colour component index cIdx set equal to 0 for luma and, when ChromaArrayType is not equal to 0, cIdx set equal to 1 or 2 for Cb and Cr chroma components respectively, coding block width log2CbWidth and coding block height log2CbHeight set equal to luma block width and height if cIdx is equal to 0 and set equal to chroma block width and height if cIdx is equal to 1 or 2.

For each coding block with width log2CbWidth, height log2CbHeight, 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 and a variable splitTH specifying block split threshold set equal to MaxTbLog2SizeY if cIdx is equal to 0 and set equal to (MaxTbLog2SizeY − 1) if cIdx is equal to 1 or 2, the deblocking process is conducted as follows:

— If edgeType is equal to EDGE_VER and log2CbWidth <= splitTH, the following applies:

  — The deblocking process of coding block boundary as specified in subclause 8.8.3.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width log2CbWidth, block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— Otherwise, if edgeType is equal to EDGE_VER and log2CbWidth > splitTH, the following applies:

  — The deblocking process of coding block boundary as specified in subclause 8.8.3.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width (log2CbWidth >> 1), block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour

component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— The deblocking process of coding block boundary as specified in subclause 8.8.3.3 is invoked with luma location of top-left sample of the coding block (xCb + (1 << splitTH), yCb), block width (log2CbWidth >> 1), block height log2CbHeight, and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— Otherwise, if edgeType is equal to EDGE_HOR and log2CbHeight <= splitTH, the following applies:

— The deblocking process of coding block boundary as specified in subclause 8.8.3.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb), block width log2CbWidth, block height log2CbHeight, and picture arrays $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$ the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— Otherwise, if edgeType is equal to EDGE_HOR and log2CbHeight > splitTH, the following applies:

— The deblocking process of coding block boundary as specified in subclause 8.8.3.3 is invoked with lumalocation of top-left sample of the coding block (xCb, yCb), block width log2CbWidth, block height (log2CbHeight >> 1), and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1,the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

— The deblocking process of coding block boundary as specified in subclause 8.8.3.3 is invoked with luma location of top-left sample of the coding block (xCb, yCb + (1 << splitTH)), block width log2CbWidth, block height (log2CbHeight >> 1), and picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, the edgeType, the luma motion vectors arrays MvL0 and MvL1, the reference indices arrays RefIdxL0 and RefIdxL1 and colour component index cIdx as inputs, and the modified arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

8.8.3.3 Deblocking filter process of coding block boundary

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 log2CbWidth and log2CbHeight specifying the width and the height of the current coding block,

— the picture arrays $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$, and $recPicture_{Cr}$,

— the edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

— luma motion vector arrays mvL0 and mvL1,

— reference indices arrays refIdxL0 and refIdxL1, and

— a variable cIdx specifying colour component index.

Output of this process are:

— the array $recPicture_L$ specifying the modified picture array after deblocking for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the modified picture array after deblocking for Cb, and

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cr}$ specifying the modified picture array after deblocking for Cr.

If cIdx is equal to 0, or ChromaArrayType is equal to 3, or ChromaArrayType is equal to 2 and edgeType is equal to EDGE_HOR, the filtering process for edges in the luma coding block, when cIdx is equal to 0, or chroma coding block, when cIdx is equal to 1 or 2, of the current coding unit consists of the following steps:

— The variable chromaStyleFilteringFlag is set equal to 0.

— The variable recPicture is set equal to $recPicture_L$ when cIdx is equal to 0, or to $recPicture_{Cb}$ when cIdx is equal to 1, or to $recPicture_{Cr}$ when cIdx is equal to 2, respectively.

— The variable xCb′ is set equal to xCb / (cIdx ? SubWidthC : 1), and yCb′ is set equal to yCb / (cIdx ? SubHeightC : 1).

— The variables xN and yN, are derived as follows:

  — If cIdx is equal to 0, or ChromaArrayType is equal to 3, the variable xN is set equal to (1 << (log2CbWidth − 2)) − 1.

  — Otherwise, the variable xN is set equal to (1 << (log2CbWidth − 1)) − 1.

  — The variable yN is set equal to (1 << (log2CbHeight − 2)) − 1.

— If edgeType is equal to EDGE_HOR, the variables xD_i, yD_j, xD_i', and yD_j' are derived as follows:

  — xD_i is set equal to i << 2 with i = 0..xN and yD_j is set equal to 0.

  — If cIdx is not equal to 0 and ChromaArrayType is equal to 2, xD_i' is set equal to i << 1 with i = 0..xN.

  — Otherwise, xD_i' is set equal to i << 2 with i = 0..xN.

  — yD_j' is set equal to 0.

— Otherwise (edgeType is equal to EDGE_VER), the variables xD_i, yD_j, xD_i', and yD_j' are derived as follows:

  — xD_i is set equal to 0, and yD_j is set equal to j << 2 with j = 0..yN.

  — xD_i' is set equal to 0, and yD_j' is set equal to j << 2 with j = 0..yN.

— For xD_i and yD_j, the following ordered steps apply:

1) Two luma locations (xCbP, yCbP) and (xCbQ, yCbQ) are derived as follows:

   — If edgeType is equal to EDGE_VER, (xCbP, yCbP) is set equal to (xCb + xD_i – 1, yCb + yD_j) and (xCbQ, yCbQ) is set equal to (xCb + xD_i, yCb + yD_j).

   — Otherwise (edgeType is equal to EDGE_HOR), (xCbP, yCbP) is set equal to (xCb + xD_i, yCb + yD_j – 1) and (xCbQ, yCbQ) is set equal to (xCb + xD_i, yCb + yD_j).

2) Two block locations (xCbP′, yCbP′) and (xCbQ′, yCbQ′) are derived as follows:

   — If edgeType is equal to EDGE_VER, (xCbP′, yCbP′) is set equal to (xCb′ + xD_i' – 1, yCb′ + yD_j') and (xCbQ′, yCbQ′) is set equal to (xCb′ + xD_i', yCb′ + yD_j').

   — Otherwise (edgeType is equal to EDGE_HOR), (xCbP′, yCbP′) is set equal to (xCb′ + xD_i', yCb′ + yD_j' – 1) and (xCbQ′, yCbQ′) is set equal to (xCb′ + xD_i', yCb′ + yD_j').

3) The variable bS[xD_i][yD_j] is derived by invoking the process as specified in subclause 8.8.3.4 with (xCbP, yCbP), (xCbQ, yCbQ), refIdxL0, refIdxL1, mvL0, mvL1 when cIdx is equal to 0 or ChromaArrayType equal to 3.

4) The sample values p_k and q_k with k = 0..3 are derived as follows:

   — If edgeType is equal to EDGE_VER, p_k is set equal to recPicture[xCbP′ – k][yCbP′ + d] and q_k is set equal to recPicture[xCbQ′ + k][yCbQ′ + d] with offset d = 0..3.

   — Otherwise (edgeType is equal to EDGE_HOR), p_k is set equal to recPicture[xCbP′ + d][yCbP′ – k] and q_k is set equal to recPicture[xCbQ′ + d][yCbQ′ + k] with offset d = 0..3 when cIdx is equal to 0 or ChromaArrayType is not equal to 2, otherwise with offset d = 0..1 when ChromaArrayType is equal to 2 and cIdx is not equal to 0.

   — The thresholds α, β, the variables filterSamplesFlag and indexA for each block edge are derived by invoking the process as specified in subclause 8.8.3.5 with p_0, p_1, q_0, q_1, the variable cIdx specifying colour component index, and bS[xD_i][yD_j] as inputs.

   — Depending on the variable filterSamplesFlag, the following applies:

      — If filterSamplesFlag is equal to 1, the following applies:

         — If bS[xD_i][yD_j] is less than 4, the process as specified in subclause 8.8.3.6 is invoked with p_i, q_i (i = 0..2), chromaStyleFilteringFlag, cIdx, bS[xD_i][yD_j], β and indexA given as inputs, and the output is assigned to p'_i and q'_i (i = 0..2).

         — Otherwise (bS is equal to 4), the process as specified in subclause 8.8.3.7 is invoked with p_i and q_i (i = 0..3), chromaStyleFilteringFlag, cIdx, α, and β given as inputs, and the output is assigned to p'_i and q'_i (i = 0..2).

      — Otherwise (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 as follows:

$$p'_i = p_i, \text{ for } i = 0..2 \tag{1214}$$

$$q'_i = q_i, \text{ for } i = 0..2 \tag{1215}$$

   — The filtered reconstructed sample p'_k and q'_k are used to replace the reconstructed samples in the reconstructed picture $recPicture_L$, when cIdx is equal to 0, or in $recPicture_{Cb}$ when cIdx is

equal to 1, or to $recPicture_{Cr}$ when cIdx is equal to 2, respectively, in the corresponding sample positions.

Otherwise, the filtering process for edges in the chroma coding blocks of current coding unit consists of the following steps:

— The variable chromaStyleFilteringFlag is set equal to 1.

— The variable $recPicture_X$ is set equal to $recPicture_{Cb}$ when cIdx is equal to 1, or to $recPicture_{Cr}$ when cIdx is equal to 2, with X being replaced by Cb or Cr.

— The variable xCb′ is set equal to xCb / SubWidthC, and yCb′ is set equal to yCb / SubHeightC.

— The variables xN and yN, are derived as follows:

  — The variable xN is set equal to (1 << (log2CbWidth − 1)) − 1.

  — If ChromaArrayType is equal to 1, the variable yN is set equal to (1 << (log2CbHeight − 1)) − 1.

  — Otherwise, the variable yN is set equal to (1 << (log2CbHeight − 2)) − 1.

— If edgeType is equal to EDGE_HOR, the variables xDi, yDj, xDi′, and yDj′ are derived as follows:

  — xD_i is set equal to i << 2 with i = 0..xN and yD_j is set equal to 0.

  — xD_i' is set equal to i << 1 with i = 0..xN and yD_j' is set equal to 0.

— Otherwise (edgeType is equal to EDGE_VER), the variables xD_i, yD_j, xD_i', and yD_j' are derived as follows:

  — xD_i is set equal to 0, and yD_j is set equal to j << 2 with j = 0..yN.

  — xD_i' is set equal to 0.

  — if ChromaArrayType is equal to 1, yD_j' is set equal to j << 1 with j = 0..yN.

  — Otherwise, yD_j' is set equal to j << 2 with j = 0..yN.

— For xD_i and yD_j, the following ordered steps apply:

  1) Two block locations (xCbP, yCbP) and (xCbQ, yCbQ) are derived as follows:

     — If edgeType is equal to EDGE_VER, (xCbP, yCbP) is set equal to (xCb′ + xD_i' − 1, yCb′ + yD_j') and (xCbQ, yCbQ) is set equal to (xCb′ + xD_i', yCb′ + yD_j').

     — Otherwise (edgeType is equal to EDGE_HOR), (xCbP, yCbP) is set equal to (xCb′ + xD_i', yCb′ + yD_j' − 1) and (xCbQ, yCbQ) is set equal to (xCb′ + xD_i', yCb′ + yD_j').

  2) The sample values p_k and q_k with k = 0..3 are derived as follows:

     — If edgeType is equal to EDGE_VER, p_k is set equal to $recPicture_X$[xCbP − k][yCbP + d] and q_k is set equal to $recPicture_X$[xCbQ + k][yCbQ + d] with offset d = 0..1 when ChromaArrayType is equal to 1 and with offset d = 0..3 when ChromaArrayType is equal to 2.

— Otherwise (edgeType is equal to EDGE_HOR), p_k is set equal to $recPicture_X[xCbP + d][yCbP - k]$ and q_k is set equal to $recPicture_X[xCbQ + d][yCbQ + k]$ with offset d = 0..1.

— The thresholds α, β, the variables filterSamplesFlag and indexA for each block edge are derived by invoking the process as specified in subclause 8.8.3.5 with p_0, p_1, q_0, q_1, a variable cIdx specifying colour component index, and $bS[xD_i][yD_j]$ as inputs.

— Depending on the variable filterSamplesFlag, the following applies:

  — If filterSamplesFlag is equal to 1, the following applies:

    — If $bS[xD_i][yD_j]$ is less than 4, the process as specified in subclause 8.8.3.6 is invoked with p_i, q_i (i = 0..2), chromaStyleFilteringFlag, cIdx, $bS[xD_i][yD_j]$, β and indexA given as inputs, and the output is assigned to p'_i and q'_i (i = 0..2).

    — Otherwise (bS is equal to 4), the process as specified in subclause 8.8.3.7 is invoked with p_i and q_i (i = 0..3), chromaStyleFilteringFlag, cIdx, α, and β given as inputs, and the output is assigned to p'_i and q'_i (i = 0..2).

  — Otherwise (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 as follows:

$$p'_i = p_i, \text{ for } i = 0..2 \tag{1216}$$

$$q'_i = q_i, \text{ for } i = 0..2 \tag{1217}$$

— The filtered reconstructed sample p'_k and q'_k are used to replace the reconstructed samples in the reconstructed picture $recPicture_X$ in the corresponding sample positions.

8.8.3.4 Derivation process for boundary filtering strength

Inputs to this process are:

— two luma locations (xCbP, yCbP) and (xCbQ, yCbQ),

— reference indices refIdxL0 and refIdxL1, and

— motion information mvL0 and mvL1.

Output of this process is the variable $bS[xD_i][yD_j]$ at location (xD_i, yD_j).

The variable cBf is derived as follows:

— If ChromaArrayType is equal to 3, and cIdx is equal to 1, the variable cBf is set equal to cbf_cb, otherwise, if ChromaArrayType is equal to 3, and cIdx is equal to 2, the variable cBf is set equal to cbf_cr.

— Otherwise, variable cBf is set equal to cbf_luma.

The variable $bS[xD_i][yD_j]$ is derived as follows:

— If the sample at location (xCbP, yCbP) or (xCbQ, yCbQ) is in the luma coding block of a coding unit coded with intra prediction mode and the block at location (xCbP, yCbP) and (xCbQ, yCbQ) are from two coding units that are located in different CTU, $bS[xD_i][yD_j]$ is set equal to 4.

— Otherwise, if the sample at location (xCbP, yCbP) or (xCbQ, yCbQ) is in the luma coding block of a coding unit coded with intra prediction mode or ibc prediction mode, bS[xD_i][yD_j] is set equal to 3.

— Otherwise, if the sample at location (xCbP, yCbP) or (xCbQ, yCbQ) is in the luma coding block of a coding unit coded with cBf equal to 1, or coded with ats_cu_inter_flag equal to 1, bS[xD_i][yD_j] is set equal to 2.

— Otherwise, the following applies:

— The variables mv0L0x, mv0L0y, mv1L0x, and mv1L0y are derived as follows:

refIdx0L0 = refIdxL0[xCbQ][yCbQ] (1218)

refIdx0L1 = refIdxL1[xCbQ][yCbQ] (1219)

refIdx1L0 = refIdxL0[xCbP][yCbP] (1220)

refIdx1L1 = refIdxL1[xCbP][yCbP] (1221)

mv0L0x = refIdx0L0 != −1 ? mvL0[xCbQ][yCbQ][0] : 0 (1222)

mv0L0y = refIdx0L0 != −1 ? mvL0[xCbQ][yCbQ][1] : 0 (1223)

mv0L1x = refIdx0L1 != −1 ? mvL1[xCbQ][yCbQ][0] : 0 (1224)

mv0L1y = refIdx0L1 != −1 ? mvL1[xCbQ][yCbQ][1] : 0 (1225)

mv1L0x = refIdx1L0 != −1 ? mvL0[xCbP][yCbP][0] : 0 (1226)

mv1L0y = refIdx1L0 != −1 ? mvL0[xCbP][yCbP][1] : 0 (1227)

mv1L1x = refIdx1L1 != −1 ? mvL1[xCbP][yCbP][0] : 0 (1228)

mv1L1y = refIdx1L1 != −1 ? mvL1[xCbP][yCbP][1] : 0 (1229)

— If refIdx0L0 is equal to refIdx1L0 and refIdx0L1 is equal to refIdx1L1 or if refIdx0L0 is equal to refIdx1L1 and refIdx0L1 is equal to refIdx1L0, the variable bS is derived as follows:

— If refIdx0L0 is equal to refIdx0L1, the variable bS is derived by:

bS[xD_i][yD_j] = (Abs(mv0L0x − mv1L0x) >= 4 | | Abs(mv0L0y − mv1L0y) >= 4 | |
Abs(mv0L1x − mv1L1x) >= 4 | | Abs(mv0L1y − mv1L1y) >= 4 | |
Abs(mv0L0x − mv1L1x) >= 4 | | Abs(mv0L0y − mv1L1y) >= 4 | |
Abs(mv0L1x − mv1L0x) >= 4 | | Abs(mv0L1y − mv1L0y) >= 4) ? 1 : 0 (1230)

— Otherwise, if refIdx0L0 is equal to refIdx1L0 and refIdx0L1 is equal to refIdx1L1, the variable bS is derived by:

bS[xD_i][yD_j] = (Abs(mv0L0x − mv1L0x) >= 4 | | Abs(mv0L0y − mv1L0y) >= 4 | |
Abs(mv0L1x − mv1L1x) >= 4 | | Abs(mv0L1y − mv1L1y) >= 4) ? 1 : 0 (1231)

— Otherwise, if refIdx0L0 is equal to refIdx1L1 and refIdx0L1 is equal to refIdx1L0, the variable bS is derived by:

bS[xD_i][yD_j] = (Abs(mv0L0x − mv1L1x) >= 4 | | Abs(mv0L0y − mv1L1y) >= 4 | |
Abs(mv0L1x − mv1L0x) >= 4 | | Abs(mv0L1y − mv1L0y) >= 4) ? 1 : 0 (1232)

— Otherwise, $bS[xD_i][yD_j]$ is set equal to 1.

8.8.3.5 Derivation process for the thresholds for each block edge

Inputs to this process are:

— the variables p_0, p_1, q_0 and q_1 specifying the values of a single set of samples across an edge that is to be filtered,

— the variable cIdx specifying colour component index, and

— the variables $bS[xD_i][yD_j]$.

Outputs of this process are:

— the variable filterSamplesFlag specifying the indicates whether the input samples are filtered,

— the variable indexA, and

— the variables α and β specifying the values of the threshold.

The variable qP_p and qP_q are derived as follows:

— If cIdx is equal to 0, qP_p and qP_q are set equal to Qp_Y that are used in the CU where sample p_0 and q_0 are located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— Otherwise, if cIdx is equal to 1, qP_p and qP_q are set equal to Qp_{Cb} that are used in the CU where sample p_0 and q_0 are located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

— Otherwise, if cIdx is equal to 2, qP_p and qP_q are set equal to Qp_{Cr} that are used in the CU where sample p_0 and q_0 are located, and is obtained according to derivation process for quantization parameters as specified in subclause 8.7.1.

The variable bitDepth is set equal to $BitDepth_Y$ if cIdx is equal to 0, otherwise bitDepth is set equal to $BitDepth_C$.

Let qP_{av} be a variable specifying an average quantization parameter. It is derived as follows:

$$qP_{av} = (qP_p + qP_q + 1) >> 1 \quad (1233)$$

Let indexA be a variable that is used to access Table 34 as well as Table 35, which is used in filtering of edges with $bS[xD_i][yD_j]$ less than 4 as derived in subclause 8.8.3.6, and let indexB be a variable that is used to access Table 34. The variables indexA and indexB are derived as follows:

$$indexA = Clip3(0, 51, qP_{av} + FilterOffsetA) \quad (1234)$$

$$indexB = Clip3(0, 51, qP_{av} + FilterOffsetB) \quad (1235)$$

The variables α′ and β′ depending on the values of indexA and indexB are specified in Table 34. The corresponding threshold variables α and β are derived as follows:

$$α = α' * (1 << (bitDepth − 8)) \quad (1236)$$

$$β = β' * (1 << (bitDepth − 8)) \quad (1237)$$

The variable filterSamplesFlag is derived as follows:

filterSamplesFlag =
(bS[xD_i][yD_j] != 0 && Abs($p_0 - q_0$) < α && Abs($p_1 - p_0$) < β && Abs($q_1 - q_0$) < β) (1238)

Table 34 — 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

8.8.3.6 Derivation process for edges with bS less than 4

Inputs to this process are:

— input sample values p_i and q_i (i = 0..2) of a single set of samples across an edge that is to be filtered,

— the variable chromaStyleFilteringFlag,

— the variable cIdx specifying colour component index, and

— the variables bS, β, and indexA for the set of input samples.

Outputs of this process are the filted result sample values p'_i and q'_i (i = 0..2) for the set of input sample values.

The variable bitDepth is set equal to $BitDepth_Y$ if cIdx is equal to 0, otherwise BitDepth is set equal to $BitDepth_C$.

Depending on the values of indexA and bS, the variable t'_{C0} is specified in Table 35.

The threshold variable t_C is derived as follows:

$t_C = t'_{C0} * (1 << (Max(0, bitDepth - 9)))$ (1239)

The threshold variables a_p and a_q are derived as follows:

$a_p = Abs(p_2 - p_0)$ (1240)

$a_q = Abs(q_2 - q_0)$ (1241)

The threshold variable t_{C0} is determined as follows:

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

tcIncP = ((a_p < β) ? 1 : 0) (1242)

$$tcIncQ = ((a_q < \beta) \ ? \ 1 : \ 0) \quad (1243)$$

$$t_{C0} = (t'_{C0} + tcIncP + tcIncQ) * (1 \ << \ (Max(0, bitDepth - 9))) \quad (1244)$$

— Otherwise (chromaStyleFilteringFlag is equal to 1), the following applies:

$$t_{C0} = (t'_{C0} + 1) * (1 \ << \ (Max(0, bitDepth - 9))) \quad (1245)$$

The filtered result samples p'_0 and q'_0 are derived as follows:

$$\Delta = Clip3(- t_{C0}, t_{C0}, (((q_0 - p_0) \ << \ 2) + (p_1 - q_1) + 4) \ >> \ 3)) \quad (1246)$$

$$p'_0 = p_0 + \Delta \quad (1247)$$

$$q'_0 = q_0 - \Delta \quad (1248)$$

The filtered result sample p'_1 is derived as follows:

— If chromaStyleFilteringFlag is equal to 0 and a_p is less than β, the following applies:

$$p'_1 = p_1 + Clip3(- t_c, t_c, ((p_2 + p_0 + q_0) * 3 - (p_1 << 3) - q_1) \ >> \ 4) \quad (1249)$$

— Otherwise (chromaStyleFilteringFlag is equal to 1 or a_p is greater than or equal to β), the following applies:

$$p'_1 = p_1 \quad (1250)$$

The filtered result sample q'_1 is derived as follows:

— If chromaStyleFilteringFlag is equal to 0 and a_q is less than β, the following applies

$$q'_1 = q_1 + Clip3(- t_c, t_c, ((q_2 + q_0 + p_0) * 3 - (q_1 << 3) - p_1) \ >> \ 4) \quad (1251)$$

— Otherwise (chromaStyleFilteringFlag is equal to 1 or a_q is greater than or equal to β), the following applies

$$q'_1 = q_1 \quad (1252)$$

The filtered result samples p'_2 and q'_2 are always set equal to the input samples p_2 and q_2 as follows:

$$p'_2 = p_2 \quad (1253)$$

$$q'_2 = q_2 \quad (1254)$$

Let Clip1( ) be a function that is replaced by $Clip1_Y( )$ when cIdx is equal to 0 and by $Clip1_C( )$ when cIdx is equal to 1 or 2.

$$p'_0 = Clip1(p'_0) \quad (1255)$$

$$p'_1 = Clip1(p'_1) \quad (1256)$$

$$p'_2 = Clip1(p'_2) \quad (1257)$$

$$q'_0 = Clip1(q'_0) \quad (1258)$$

$q'_1 = Clip1(q'_1)$ (1259)

$q'_2 = Clip1(q'_2)$ (1260)

Table 35 — 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

8.8.3.7 Derivation process for edges with bS equal to 4

Inputs to this process are:

— input sample values p_i and q_i (i = 0..3) of a single set of samples across an edge that is to be filtered,

— the variable chromaStyleFilteringFlag,

— the variable cIdx specifying colour component index, and

— the value of threshold variable β.

Outputs of this process are the filtered result sample values p'_i and q'_i (i = 0..2) for the set of input sample values.

Let a_p and a_q be two threshold variables as specified in Formulae 1240 and 1241, respectively in subclause 8.8.3.6.

The filtered result samples p'_i (i = 0..2) are derived as follows:

— If chromaStyleFilteringFlag is equal to 0 and the following condition holds,

$a_p < \beta \;\&\&\; Abs(p_0 - q_0) < ((\alpha >> 2) + 2)$ (1261)

then the variables p'_0, p'_1, and p'_2 are derived as follows:

$p'_0 = (p_2 + 2 * p_1 + 2 * p_0 + 2 * q_0 + q_1 + 4) >> 3$ (1262)

$p'_1 = (p_2 + p_1 + p_0 + q_0 + 2) >> 2$ (1263)

$p'_2 = (2 * p_3 + 3 * p_2 + p_1 + p_0 + q_0 + 4) >> 3$ (1264)

— Otherwise (chromaStyleFilteringFlag is equal to 1 or the condition in Formula 1261 does not hold), the variables p'_0, p'_1, and p'_2 are derived as follows:

$$p'_0 = (2 * p_1 + p_0 + q_1 + 2) >> 2 \quad (1265)$$

$$p'_1 = p_1 \quad (1266)$$

$$p'_2 = p_2 \quad (1267)$$

The filtered result samples q'_i (i = 0..2) are derived as follows:

— If chromaStyleFilteringFlag is equal to 0 and the following condition holds,

$$a_q < \beta \ \&\& \ Abs(p_0 - q_0) < ((\alpha >> 2) + 2) \quad (1268)$$

then the variables q′0, q′1, and q′2 are derived as follows:

$$q'_0 = (p_1 + 2 * p_0 + 2 * q_0 + 2 * q_1 + q_2 + 4) >> 3 \quad (1269)$$

$$q'_1 = (p_0 + q_0 + q_1 + q_2 + 2) >> 2 \quad (1270)$$

$$q'_2 = (2 * q_3 + 3 * q_2 + q_1 + q_0 + p_0 + 4) >> 3 \quad (1271)$$

— Otherwise (chromaStyleFilteringFlag is equal to 1 or the condition in Formula 1268 does not hold), the variables q'_0, q'_1, and q'_2 are derived as follows:

$$q'_0 = (2 * q_1 + q_0 + p_1 + 2) >> 2 \quad (1272)$$

$$q'_1 = q_1 \quad (1273)$$

$$q'_2 = q_2 \quad (1274)$$

Let Clip1( ) be a function that is replaced by $Clip1_Y( )$ when cIdx is equal to 0 and by $Clip1_C( )$ when cIdx is equal to 1 or 2.

$$p'_0 = Clip1(p'_0) \quad (1275)$$

$$p'_1 = Clip1(p'_1) \quad (1276)$$

$$p'_2 = Clip1(p'_2) \quad (1277)$$

$$q'_0 = Clip1(q'_0) \quad (1278)$$

$$q'_1 = Clip1(q'_1) \quad (1279)$$

$$q'_2 = Clip1(q'_2) \quad (1280)$$

8.8.4 Adaptive Loop Filter

8.8.4.1 General

Inputs to this process are:

— the array $recPicture_L$ specifying the reconstructed picture sample array prior to adaptive loop filter for luma,

— when ChromaArrayType is not equal to 0, the array $recPicture_{Cb}$ specifying the reconstructed picture sample array prior to adaptive loop filter for Cb, and

— when ChromaArrayType is not equal to 0, the array$recPicture_{Cr}$ specifying the reconstructed picture sample array prior to adaptive loop filter for Cr.

Outputs of this process are:

— the array $alfPicture_L$ specifying the modified reconstructed picture sample array after adaptive loop filter for luma,

— when ChromaArrayType is not equal to 0, $alfPicture_{Cb}$ specifying the modified reconstructed picture sample array after adaptive loop filter for Cb, and

— when ChromaArrayType is not equal to 0, $alfPicture_{Cr}$ specifying the modified reconstructed picture sample array after adaptive loop filter for Cr.

The sample values in the modified reconstructed picture sample arrays $alfPicture_L$, and, when ChromaArrayType is not equal to 0, $alfPicture_{Cb}$ and $alfPicture_{Cr}$ are initially set equal to the sample values in the reconstructed picture sample arrays prior to adaptive loop filter $recPicture_L$, and, when ChromaArrayType is not equal to 0, $recPicture_{Cb}$ and $recPicture_{Cr}$, respectively.

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:

— If (rx << CtbLog2SizeY) + CtbSizeY is smaller than or equal to pic_width_in_luma_samples, a variable blkWidth is set equal to CtbSizeY, otherwise, a variable blkWidth is set equal to pic_width_in_luma_samples – (rx << CtbLog2SizeY).

— If (ry << CtbLog2SizeY) + CtbSizeY is smaller than or equal to pic_height_in_luma_samples, a variable blkHeight is set equal to CtbSizeY, otherwise blkHeight is set equal to pic_height_in_luma_samples – (ry << CtbLog2SizeY).

— When alf_ctb_flag[rx][ry] is equal to 1, the coding tree block luma type filtering process as specified in subclause 8.8.4.2 is invoked with $recPicture_L$, $alfPicture_L$, referred APS identifier slice_alf_luma_aps_id, the luma coding tree block location (xCtb, yCtb) set equal to (rx << CtbLog2SizeY, ry << CtbLog2SizeY), and the coding tree block width blkWidth and height blkHeight as inputs, and the output is the modified filtered picture $alfPicture_L$.

— If ChromaArrayType is equal to 3, the coding tree block luma type filtering process for chroma samples of Cb and Cr chroma components are invoked as follows:

  — When alf_ctb_chroma_flag[rx][ry] is equal to 1, the coding tree block luma type filtering process as specified in subclause 8.8.4.2 is invoked with $recPicture_{Cb}$, $alfPicture_{Cb}$, referred APS identifier slice_alf_chroma_aps_id, the chroma coding tree block location (xCtb, yCtb) set equal to (rx << CtbLog2SizeY, ry << CtbLog2SizeY), and the coding tree block width blkWidth and height blkHeight as inputs, and the output is the modified filtered picture $alfPicture_{Cb}$.

  — When alf_ctb_chroma2_flag[rx][ry] is equal to 1, the coding tree block luma type filtering process as specified in subclause 8.8.4.2 is invoked with $recPicture_{Cr}$, $alfPicture_{Cr}$, referred APS identifier slice_alf_chroma2_aps_id, the chroma coding tree block location (xCtb, yCtb) set equal to (rx << CtbLog2SizeY, ry << CtbLog2SizeY), and the coding tree block width blkWidth and height blkHeight as inputs, and the output is the modified filtered picture $alfPicture_{Cr}$.

— Otherwise, when ChromaArrayType is in the range from 1 to 2, inclusive, and slice_alf_chroma_idc is larger than 0, the following applies:

  — When sliceChromaAlfEnabledFlag is equal to 1, the coding tree block chroma type filtering process as specified in subclause 8.8.4.4 is invoked with recPicture set equal to $recPicture_{Cb}$, alfPicture set equal to $alfPicture_{Cb}$, referred APS identifier slice_alf_chroma_aps_id, the chroma coding tree block location (xCtbC, yCtbC) set equal to ((rx << CtbLog2SizeY) / SubWidthC, (ry << CtbLog2SizeY) / SubHeightC), and the coding tree block width blkWidth / SubWidthC and height blkHeight / SubHeightC as inputs, and the output is the modified filtered picture $alfPicture_{Cb}$.

  — When sliceChroma2AlfEnabledFlag is equal to 1, the coding tree block chroma type filtering process as specified in subclause 8.8.4.4 is invoked with recPicture set equal to $recPicture_{Cr}$, alfPicture set equal to alfPictureCr, referred APS identifier slice_alf_chroma_aps_id, the chroma coding tree block location (xCtbC, yCtbC) set equal to ((rx << CtbLog2SizeY) / SubWidthC, (ry << CtbLog2SizeY) / SubHeightC), and the coding tree block width blkWidth / SubWidthC and height blkHeight / SubHeightC as inputs, and the output is the modified filtered picture alfPictureCr.

8.8.4.2 Coding tree block luma type filtering process

Inputs to this process are:

— a reconstructed picture sample array recPictureRec prior to the adaptive loop filtering process,

— a filtered reconstructed luma picture sample array alfPicture,

— an identifier of the referred APS, apsId,

— a luma location (xCtb, yCtb) specifying the top-left sample of the current coding tree block relative to the top-left sample of the current picture, and

— variables blkWidth and blkHeight specifying the width and the height of the current coding tree block.

Output of this process is the modified filtered reconstructed picture sample array alfPicture.

The sample values in the picture sample arrays recPicture are derived by invoking subclause 8.8.4.5 with recPictureRec, the coding tree block location (xCtb, yCtb), and the coding tree block width blkWidth and the coding tree block height blkHeight as inputs, and the output is the modified filtered picture recPicture.

The derivation process for ALF transpose and filter index subclause 8.8.4.3 is invoked with the location (xCtb, yCtb), the reconstructed picture sample array recPicture, and the coding tree block width blkWidth and the coding tree block height blkHeight as inputs, and filtIdx[x][y] and transposeIdx[x][y] with x = 0..blkWidth – 1, y = 0..blkHeight – 1 as outputs.

For the derivation of the filtered reconstructed samples alfPicture[x][y], each reconstructed sample inside the current coding tree block recPicture[x][y] is filtered as follows with x = 0..blkWidth – 1, y = 0..blkHeight – 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:

$$f[j] = AlfCoeff_L[apsId][filtIdx[x][y]][j] \quad (1281)$$

The luma filter coefficients filterCoeff are derived depending on transposeIdx[x][y] as follows:

— If transposeIdx[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] } (1282)

— Otherwise, if transposeIdx[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] } (1283)

— Otherwise, if transposeIdx[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] } (1284)

— 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] } (1285)

The variable sum is derived as follows:

```
sum = filterCoeff[ 0 ]  * ( recPicture[ x, y + 3 ] + recPicture[ x, y − 3 ] ) +
    filterCoeff[ 1 ]  * ( recPicture[ x + 1, y + 2 ] + recPicture[ x − 1, y − 2 ] ) +
    filterCoeff[ 2 ]  * ( recPicture[ x, y + 2 ] + recPicture[ x, y − 2 ] ) +
    filterCoeff[ 3 ]  * ( recPicture[ x − 1, y + 2 ] + recPicture[ x + 1, y − 2 ] ) +
    filterCoeff[ 4 ]  * ( recPicture[ x + 2, y + 1 ] + recPicture[ x − 2, y − 1 ] ) +
    filterCoeff[ 5 ]  * ( recPicture[ x + 1, y + 1 ] + recPicture[ x − 1, y − 1 ] ) +
    filterCoeff[ 6 ]  * ( recPicture[ x, y + 1 ] + recPicture[ x, y − 1 ] ) +
    filterCoeff[ 7 ]  * ( recPicture[ x − 1, y + 1 ] + recPicture[ x + 1, y − 1 ] ) +
    filterCoeff[ 8 ]  * ( recPicture[ x − 2, y + 1 ] + recPicture[ x + 2, y − 1 ] ) +
    filterCoeff[ 9 ]  * ( recPicture[ x + 3, y ] + recPicture[ x − 3, y ] ) +
    filterCoeff[ 10 ] * ( recPicture[ x + 2, y ] + recPicture[ x − 2, y ] ) +
    filterCoeff[ 11 ] * ( recPicture[ x + 1, y ] + recPicture[ x − 1, y ] ) +
    filterCoeff[ 12 ] *   recPicture[ x, y ]
```

(1286)

sum = (sum + 256) >> 9 (1287)

The modified filtered reconstructed picture sample alfPicture[xCtb + x][yCtb + y] is derived as follows:

alfPicture[xCtb + x][yCtb + y] = Clip3(0, (1 << $BitDepth_Y$) − 1, sum) (1288)

8.8.4.3 Derivation process for ALF transpose and filter index

Inputs to this process are:

— a location (xCtb, yCtb) specifying the top-left sample of the current coding tree block relative to the top-left sample of the current picture,

— a reconstructed picture sample array recPicture prior to the adaptive loop filtering process, and

— variables blkWidth and blkHeight specifying the width and the height of the current coding tree block.

Outputs of this process are:

— the classification filter index array filtIdx[x][y] with x = 0..blkWidth − 1, y = 0..blkHeight − 1,

— the transpose index array transposeIdx[x][y] with x = 0..blkWidth − 1, y = 0..blkHeight − 1.

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 = −2..blkWidth + 1, y = −2..blkHeight + 1 are derived as follows:

filtH[x][y] = Abs((recPicture[xCtb + x, yCtb + y] << 1) − recPicture[xCtb + x − 1, yCtb + y] − recPicture[xCtb + x + 1, yCtb + y]) (1289)

filtV[x][y] = Abs((recPicture[xCtb + x, yCtb + y] << 1) − recPicture[xCtb + x, yCtb + y − 1] − recPicture[xCtb + x, yCtb + y + 1]) (1290)

filtD0[x][y] = Abs((recPicture[xCtb + x, yCtb + y] << 1) − recPicture[xCtb + x − 1, yCtb + y − 1] − recPicture[xCtb + x + 1, yCtb + y + 1]) (1291)

filtD1[x][y] = Abs((recPicture[xCtb + x, yCtb + y] << 1) − recPicture[xCtb + x + 1, yCtb + y − 1] − recPicture[xCtb + x − 1, yCtb + y + 1]) (1292)

— The variables sumH[x][y], sumV[x][y], sumD0[x][y], sumD1[x][y] and sumOfHV[x][y] with x = 0..(blkWidth − 1) >> 2, y = 0..(blkHeight − 1) >> 2 are derived as follows:

sumH[x][y] = ΣiΣj filtH[(x << 2) + i][(y << 2) + j] with i, j = −2..5 (1293)

sumV[x][y] = ΣiΣj filtV[(x << 2) + i][(y << 2) + j] with i, j = −2..5 (1294)

sumD0[x][y] = ΣiΣj filtD0[(x << 2) + i][(y << 2) + j] with i, j = −2..5 (1295)

sumD1[x][y] = ΣiΣj filtD1[(x << 2) + i][(y << 2) + j] with i, j = −2..5 (1296)

sumOfHV[x][y] = sumH[x][y] + sumV[x][y] (1297)

— The variables dir1[x][y], dir2[x][y] and dirS[x][y] with x = 0..blkWidth − 1, y = 0..blkHeight − 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] (1298)

      hv0 = sumH[x >> 2][y >> 2] (1299)

      dirHV = 1 (1300)

    — Otherwise, the following applies:

      hv1 = sumH[x >> 2][y >> 2] (1301)

      hv0 = sumV[x >> 2][y >> 2] (1302)

      dirHV = 3 (1303)

  — 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] (1304)

d0 = sumD1[x >> 2][y >> 2] (1305)

dirD = 0 (1306)

— Otherwise, the following applies:

d1 = sumD1[x >> 2][y >> 2] (1307)

d0 = sumD0[x >> 2][y >> 2] (1308)

dirD = 2 (1309)

— The variables hvd1 and hvd0 are derived as follows:

hvd1 = (d1 * hv0 > hv1 * d0) ? d1 : hv1 (1310)

hvd0 = (d1 * hv0 > hv1 * d0) ? d0 : hv0 (1311)

— The variables dirS[x][y], dir1[x][y] and dir2[x][y] are derived as follows:

dir1[x][y] = (d1 * hv0 > hv1 * d0) ? dirD : dirHV (1312)

dir2[x][y] = (d1 * hv0 > hv1 * d0) ? dirHV : dirD (1313)

dirS[x][y] = (hvd1 > 2 * hvd0) ? 1 : ((hvd1 * 2 > 9 * hvd0) ? 2 : 0) (1314)

— The variable avgVar[x][y] with x = 0..blkWidth – 1, y = 0..blkHeight – 1 is derived as follows:

varTab[ ] = { 0, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4 } (1315)

avgVar[x][y] = varTab[Clip3(0, 15, (sumOfHV[x >> 2][y >> 2]) >> ($BitDepth_Y$ – 2))] (1316)

— The classification filter index array filtIdx[x][y] and the transpose index array transposeIdx[x][y] with x = 0..blkWidth – 1, y = 0..blkHeight – 1 are derived as follows:

transposeTable[ ] = { 0, 1, 0, 2, 2, 3, 1, 3 } (1317)

transposeIdx[x][y] = transposeTable[dir1[x][y] * 2 + (dir2[x][y] >> 1)] (1318)

filtIdx[x][y] = avgVar[x][y] (1319)

— When dirS[x][y] is not equal to 0, filtIdx[x][y] is modified as follows:

filtIdx[x][y] += (((dir1[x][y] & 0x1) << 1) + dirS[x][y]) * 5 (1320)

8.8.4.4 Coding tree block chroma type filtering process

Inputs to this process are:

— a reconstructed chroma picture sample array recPictureRec prior to the adaptive loop filtering process,

— a filtered reconstructed chroma picture sample array alfPicture,

— an identifier of the referred APS, apsId,

— 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, and

— variables blkWidth and blkHeight specifying the width and the height of the current coding tree block.

Output of this process is the modified filtered reconstructed chroma picture sample array alfPicture.

The sample values in the picture sample arrays recPicture are derived by invoking subclause 8.8.4.6 with recPictureRec, the chroma coding tree block location (xCtbC, yCtbC), and the coding tree block width blkWidth and block height blkHeight as inputs, and the output is the modified filtered picture recPicture.

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 = 0..blkWidth − 1, y = 0..blkHeight − 1:

The variable sum is derived as follows:

$$\begin{aligned} sum = {} & AlfCoeff_C[\ apsId\ ][\ 0\ ] * (recPicture[\ x, y + 2\ ] + recPicture[\ x, y - 2\ ]) + \\ & AlfCoeff_C[\ apsId\ ][\ 1\ ] * (recPicture[\ x + 1, y + 1\ ] + recPicture[\ x - 1, y - 1\ ]) + \\ & AlfCoeff_C[\ apsId\ ][\ 2\ ] * (recPicture[\ x, y + 1\ ] + recPicture[\ x, y - 1\ ]) + \\ & AlfCoeff_C[\ apsId\ ][\ 3\ ] * (recPicture[\ x - 1, y + 1\ ] + recPicture[\ x + 1, y - 1\ ]) + \\ & AlfCoeff_C[\ apsId\ ][\ 4\ ] * (recPicture[\ x + 2, y\ ] + recPicture[\ x - 2, y\ ]) + \\ & AlfCoeff_C[\ apsId\ ][\ 5\ ] * (recPicture[\ x + 1, y\ ] + recPicture[\ x - 1, y\ ]) + \\ & AlfCoeff_C[\ apsId\ ][\ 6\ ] * recPicture[\ x, y\ ] \end{aligned} \tag{1321}$$

$$sum = (sum + 256) >> 9 \tag{1322}$$

The modified filtered reconstructed chroma picture sample alfPicture[xCtbC + x][yCtbC + y] is derived as follows:

$$alfPicture[\ xCtbC + x\ ][\ yCtbC + y\ ] = Clip3(0, (1 << BitDepth_C) - 1, sum) \tag{1323}$$

8.8.4.5 Derivation process for luma type ALF input samples

Inputs to this process are:

— a reconstructed picture sample array recPicture prior to the adaptive loop filtering process,

— a luma location (xCtb, yCtb) specifying the top-left sample of the current coding tree block relative to the top-left sample of the current picture, and

— variables blkWidth and blkHeight specifying the width and the height of the current coding tree block.

Output of this process is sample array recPictureOut.

Variables availableL, availableT, availableR and availableB are set equal to TRUE.

For x = 0..blkWidth − 1 and for y = 0..blkHeight − 1, variable recPictureOut is initialized as follows:

$$recPictureOut[\ xCtb + x, yCtb + y\ ] = recPicture[\ xCtb + x, yCtb + y\ ] \tag{1324}$$

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (xCtb – 1, yCtb) as

input, and the output is assigned to the variable availableL. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (xCtb – 1, yCtb) as input, and the output is assigned to the variable availableL.

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (xCtb, yCtb – 1) as input, and the output is assigned to the variable availableT. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (xCtb, yCtb – 1) as input, and the output is assigned to the variable availableT.

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (xCtb + blkWidth, yCtb) as input, and the output is assigned to the variable availableR. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (xCtb + blkWidth, yCtb) as input, and the output is assigned to the variable availableR.

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (xCtb, yCtb + blkHeight) as input, and the output is assigned to the variable availableB. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (xCtb, yCtb + blkHeight) as input, and the output is assigned to the variable availableB.

If availableL is equal to FALSE, for y = 0..blkHeight − 1, samples recPictureOut are derived as follows:

recPictureOut[xCtb − 3, yCtb + y] = recPicture[xCtb + 3, yCtb + y] (1325)

recPictureOut[xCtb − 2, yCtb + y] = recPicture[xCtb + 2, yCtb + y] (1326)

recPictureOut[xCtb − 1, yCtb + y] = recPicture[xCtb + 1, yCtb + y] (1327)

Otherwise, (availableL is equal to TRUE), samples recPictureOut are derived as follows:

recPictureOut[xCtb − 3, yCtb + y] = recPicture[xCtb − 3, yCtb + y] (1328)

recPictureOut[xCtb − 2, yCtb + y] = recPicture[xCtb − 2, yCtb + y] (1329)

recPictureOut[xCtb − 1, yCtb + y] = recPicture[xCtb − 1, yCtb + y] (1330)

If availableR is equal to FALSE, for y = 0..blkHeight − 1, samples recPictureOut are derived as follows:

recPictureOut[xCtb + blkWidth + 2, yCtb + y] = recPicture[xCtb+ blkWidth − 4, yCtb + y] (1331)

recPictureOut[xCtb + blkWidth + 1, yCtb + y] = recPicture[xCtb+ blkWidth − 3, yCtb + y] (1332)

recPictureOut[xCtb + blkWidth, yCtb + y] = recPicture[xCtb + blkWidth − 2, yCtb + y] (1333)

Otherwise (availableR is equal to TRUE), samples recPictureOut are derived as follows:

recPictureOut[xCtb + blkWidth + 2, yCtb + y] = recPicture[xCtb + blkWidth + 2, yCtb + y] (1334)

recPictureOut[xCtb + blkWidth + 1, yCtb + y] = recPicture[xCtb + blkWidth + 1, yCtb + y] (1335)

recPictureOut[xCtb + blkWidth, yCtb + y] = recPicture[xCtb + blkWidth, yCtb + y] (1336)

If availableT is equal to FALSE, for x = −3..blkWidth + 2 and, samples recPictureOut are derived as follows:

recPictureOut[xCtb + x, yCtb − 3] = recPicture[xCtb + x, yCtb + 3] (1337)

recPictureOut[xCtb + x, yCtb − 2] = recPicture[xCtb + x, yCtb + 2] (1338)

recPictureOut[xCtb + x, yCtb − 1] = recPicture[xCtb + x, yCtb + 1] (1339)

Otherwise (availableT is equal to TRUE), samples recPictureOut are derived as follows:

recPictureOut[xCtb + x, yCtb − 3] = recPicture[xCtb + x, yCtb − 3] (1340)

recPictureOut[xCtb + x, yCtb − 2] = recPicture[xCtb + x, yCtb − 2] (1341)

recPictureOut[xCtb + x, yCtb − 1] = recPicture[xCtb + x, yCtb − 1] (1342)

If availableB is equal to FALSE, for x = −3..blkWidth + 2 and, samples recPictureOut are derived as follows:

recPictureOut[xCtb + x, yCtb + blkHeight + 2] = recPicture[xCtb + x, yCtb + blkHeight − 4](1343)

recPictureOut[xCtb + x, yCtb + blkHeight + 1] = recPicture[xCtb + x, yCtb + blkHeight − 3](1344)

recPictureOut[xCtb + x, yCtb + blkHeight] = recPicture[xCtb + x, yCtb + blkHeight − 2] (1345)

Otherwise (availableB is equal to TRUE), samples recPictureOut are derived as follows:

recPictureOut[xCtb + x, yCtb + blkHeight + 2] = recPicture[xCtb + x, yCtb + blkHeight + 2](1346)

recPictureOut[xCtb + x, yCtb + blkHeight + 1] = recPicture[xCtb + x, yCtb + blkHeight + 1](1347)

recPictureOut[xCtb + x, yCtb + blkHeight] = recPicture[xCtb + x, yCtb + blkHeight] (1348)

8.8.4.6 Derivation process for chroma type ALF input samples

Inputs to this process are:

— a reconstructed chroma sample array recPictureCh prior to the adaptive loop filtering process,

— a luma location (xCtb, yCtb) specifying the top-left sample of the current chroma coding tree block relative to the top-left sample of the current picture, and

— variables blkWidth and blkHeight specifying the width and the height of the current coding tree block.

Output of this process is sample array recPictureChOut.

Variables availableL, availableT, availableR and availableB are set equal to TRUE.

For x = 0..blkWidth − 1 and for y = 0..blkHeight − 1, variable recPictureChOut is initialized as follows:

recPictureChOut[xCtb + x, yCtb + y] = recPictureCh[xCtb + x, yCtb + y] (1349)

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location

(SubWidthC * xCtb – 1, SubWidthC * yCtb) as input, and the output is assigned to the variable availableL. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (SubWidthC * xCtb – 1, SubWidthC * yCtb) as input, and the output is assigned to the variable availableL.

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (SubWidthC * xCtb, SubWidthC * yCtb – 1) as input, and the output is assigned to the variable availableT. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (SubWidthC * xCtb, SubWidthC * yCtb – 1) as input, and the output is assigned to the variable availableT.

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (SubWidthC * (xCtb + blkWidth), SubWidthC * yCtb) as input, and the output is assigned to the variable availableR. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (SubWidthC * (xCtb + blkWidth), SubWidthC * yCtb) as input, and the output is assigned to the variable availableR.

When loop_filter_across_tiles_enabled_flag is equal to FALSE, the derivation process for neighbouring block availability as specified in subclause 6.4.1 is invoked with the luma location (SubWidthC * xCtb, SubWidthC * (yCtb + blkHeight)) as input, and the output is assigned to the variable availableB. Otherwise, when loop_filter_across_tiles_enabled_flag is equal to TRUE, the derivation process for ALF neighbouring block availability as specified in subclause 6.4.4 is invoked with the luma location (SubWidthC * xCtb, SubWidthC * (yCtb + blkHeight)) as input, and the output is assigned to the variable availableB.

If availableL is equal to FALSE, for y = 0..blkHeight – 1, samples recPictureChOut are derived as follows:

recPictureChOut[xCtb – 3, yCtb + y] = recPictureCh[xCtb + 3, yCtb + y] (1350)

recPictureChOut[xCtb – 2, yCtb + y] = recPictureCh[xCtb + 2, yCtb + y] (1351)

recPictureChOut[xCtb – 1, yCtb + y] = recPictureCh[xCtb+1, yCtb + y] (1352)

Otherwise (availableL is equal to TRUE), the following applies:

recPictureChOut[xCtb – 3, yCtb + y] = recPictureCh[xCtb – 3, yCtb + y] (1353)

recPictureChOut[xCtb – 2, yCtb + y] = recPictureCh[xCtb – 2, yCtb + y] (1354)

recPictureChOut[xCtb – 1, yCtb + y] = recPictureCh[xCtb – 1, yCtb + y] (1355)

If availableR is equal to FALSE, for y = 0..blkHeight – 1, samples recPictureChOut are derived as follows:

recPictureChOut[xCtb + blkWidth + 2, yCtb + y] = recPictureCh[xCtb+ blkWidth – 4, yCtb + y] (1356)

recPictureChOut[xCtb + blkWidth + 1, yCtb + y] = recPictureCh[xCtb+ blkWidth – 3, yCtb + y] (1357)

recPictureChOut[xCtb + blkWidth, yCtb + y] = recPictureCh[xCtb + blkWidth – 2, yCtb + y] (1358)

Otherwise (availableR is equal to TRUE), the following applies:

recPictureChOut[xCtb + blkWidth + 2, yCtb + y] = recPictureCh[xCtb + blkWidth + 2, yCtb + y] (1359)

recPictureChOut[xCtb + blkWidth + 1, yCtb + y] = recPictureCh[xCtb + blkWidth + 1, yCtb + y] (1360)

recPictureChOut[xCtb + blkWidth, yCtb + y] = recPictureCh[xCtb + blkWidth, yCtb + y] (1361)

If availableT is equal to FALSE, and loop_filter_across_tiles_enabled_flag is equal to FALSE, for x = −3..blkWidth + 2 and, samples recPictureChOut are derived as follows:

recPictureChOut[xCtb + x, yCtb − 3] = recPictureCh[xCtb + x, yCtb + 3] (1362)

recPictureChOut[xCtb + x, yCtb − 2] = recPictureCh[xCtb + x, yCtb + 2] (1363)

recPictureChOut[xCtb + x, yCtb − 1] = recPictureCh[xCtb + x, yCtb + 1] (1364)

Otherwise (availableT is equal to TRUE), the following applies:

recPictureChOut[xCtb + x, yCtb − 3] = recPictureCh[xCtb + x, yCtb − 3] (1365)

recPictureChOut[xCtb + x, yCtb − 2] = recPictureCh[xCtb + x, yCtb − 2] (1366)

recPictureChOut[xCtb + x, yCtb − 1] = recPictureCh[xCtb + x, yCtb − 1] (1367)

If availableB is equal to FALSE, for x = −3..blkWidth + 2 and, samples recPictureChOut are derived as follows:

recPictureChOut[xCtb + x, yCtb + blkHeight + 2] = recPictureCh[xCtb + x, yCtb + blkHeight − 4] (1368)

recPictureChOut[xCtb + x, yCtb + blkHeight + 1] = recPictureCh[xCtb + x, yCtb + blkHeight − 3] (1369)

recPictureChOut[xCtb + x, yCtb + blkHeight] = recPictureCh[xCtb + x, yCtb + blkHeight − 2] (1370)

Otherwise (availableB is equal to TRUE), the following applies:

recPictureChOut[xCtb + x, yCtb + blkHeight + 2] = recPictureCh[xCtb + x, yCtb + blkHeight + 2] (1371)

recPictureChOut[xCtb + x, yCtb + blkHeight + 1] = recPictureCh[xCtb + x, yCtb + blkHeight + 1] (1372)

recPictureChOut[xCtb + x, yCtb + blkHeight] = recPictureCh[xCtb + x, yCtb + blkHeight] (1373)

8.9 DRA process

8.9.1 General

When pic_dra_enabled_flag is equal to 1, the DRA mapping process as specified in subclause 8.9.2 is applied prior to the cropping and output of the decoded picture as specified in subclause C.5.2.

NOTE PPS syntax elements referred in this subclause are referring to PPS referred to by the picture that is being mapped by the DRA process.

8.9.2 Derivation of samples of output decoded picture by DRA process

Inputs to this process are:

— decoded picture luma samples array decPictureL,

— decoded picture chroma sample arrays decPictureCb and decPictureCr, when ChromaTypeArray is not equal to 0, and

— identifier of the referred APS of the DRA_APS type pic_dra_aps_id.

Outputs to this process are:

— mapped picture luma samples array draPictureL, and

— mapped picture chroma sample arrays draPictureCb and draPictureCr, when ChromaTypeArray is not equal to 0.

The sample values in the mapped picture sample arrays draPictureL, and, when ChromaArrayType is not equal to 0, draPictureCb and draPictureCr are initially set equal to the sample values in the decoded picture sample arrays decPictureL, decPictureCb and decPictureCr, respectively.

When pic_dra_enabled_flag is equal to 1, the following applies:

— When ChromaArrayType is not equal to 0, for each Cb sample in the picture with location (x, y) with x in the range of 0 to PicWidthInSamplesC – 1, inclusive, and y in the range of 0 to PicHeightInSamplesC – 1, inclusive, the DRA process for chroma samples as specified in subclause 8.9.4 is invoked with decPictureL[x * SubWidthC, y * SubHeightC], decPictureCb[x, y], chroma index 0 and pic_dra_aps_id specifying the APS id of the utilized DRA function as inputs, and the output is draPictureCb[x, y].

— When ChromaArrayType is not equal to 0, for each Cr sample in the picture with location (x, y) with x in the range of 0 to PicWidthInSamplesC – 1, inclusive, and y in the range of 0 to PicHeightInSamplesC – 1, inclusive, the DRA process for chroma samples as specified in subclause 8.9.4 is invoked with decPictureL[x * SubWidthC, y * SubHeightC], decPictureCr[x, y], chroma index 1 and pic_dra_aps_id specifying the APS id of the utilized DRA function as inputs, and the output is draPictureCr[x, y].

— For each luma sample in the picture with location (x, y) with x in the range of 0 to pic_width_in_luma_samples – 1, inclusive, and y in the range of 0 to pic_height_in_luma_samples – 1, inclusive, the DRA process for luma samples as specified in subclause 8.9.3 is invoked with decPictureL[x, y] and pic_dra_aps_id specifying the APS id of the utilized DRA function as inputs, and the output is draPictureL[x, y].

8.9.3 Inverse mapping process for a luma sample

Inputs to this process are:

— a luma sample lumaSample, and

— identifier of the referred APS of the DRA_APS type apsId.

Output of this process is a modified luma sample invLumaSample.

The value of invLumaSample is derived as follows:

— The following ordered steps apply:

1) The variable rangeIdx is derived by invoking subclause 8.9.5 for a luma sample lumaSample, outputRangesL array and numOutRangesL as the inputs and rangeIdx as the output.

2) The variable mappedSample is derived as follows:

$$incrValue = InvLumaScales[apsId][rangeIdx] * lumaSample \quad (1374)$$

$$mappedSample = (DraOffsets[apsId][rangeIdx] + incrValue + (1 << 8)) >> 9 \quad (1375)$$

3) The inverse mapped luma sample invLumaSample is derived as follows:

$$invLumaSample = Clip1_Y(mappedSample) \quad (1376)$$

8.9.4 Inverse mapping process for a chroma sample

Inputs to this process are:

— luma sample lumaSample,

— chroma sample chromaSample,

— identifier of the referred APS of the DRA_APS type apsId, and

— chroma index cIdx.

Output of this process is a modified chroma sample invChromaSample.

The value of invChromaSample is derived as follows:

— The following ordered steps apply:

1) The variable chromaScale is derived by invoking process as specified in subclause 8.9.6 with lumaSample, chromaSample, identifier of DRA APS apsId, and chroma index cIdx as the input and chromaScale as the output.

2) The variable invSample is derived as follows:

$$signedValue = chromaSample - 2^{BitDepth_C - 1} \quad (1377)$$

$$unsignedValue = (signedValue < 0) ? (-1) * signedValue * chromaScale : signedValue * chromaScale \quad (1378)$$

$$scaledSample = (unsignedValue + (1 << 8)) >> 9 \quad (1379)$$

$$scaledSample = (signedValue < 0) ? (-1) * scaledSample : scaledSample \quad (1380)$$

$$scaledSample = scaledSample + 2^{BitDepth_C - 1} \quad (1381)$$

3) The inverse mapped chroma sample invChromaSample is derived as follows:

$$invChromaSample = Clip1_C(scaledSample) \quad (1382)$$

8.9.5 Identification of the range index of piecewise function

Inputs to this process are:

— input sample inputSample,

— array of range boundaries rangesArray, and

— size of the rangesArray numRanges.

Output of this process is an index rangeIdx identifying the range to which the input sample inputSample belongs.

The variable rangeIdx is derived as follows:

```
rangeFound = 0
for( rangeIdx = 0; rangeIdx < numRanges; rangeIdx++ )
    if( inputSample < rangesArray[ rangeIdx + 1 ] )
    {
        rangeFound = 1                                                    (1383)
        break
    }
rangeIdx = ( rangeFound  = =  1 ) ? rangeIdx : numRanges – 1
rangeIdx = Min( rangeIdx, numRanges – 1 )
```

8.9.6 DRA chroma scale value derivaton process

Inputs to this process are:

— a luma sample lumaSample,

— identifier of DRA APS apsId, and

— chroma index cIdx.

Output of this process is a reconstructed chroma scale value chromaScale.

The variable chromaScale is derived by the following ordered steps:

1) The variable rangeIdx is derived by invoking the process as specified in subclause 8.9.5 with the variable lumaSample, OutRangesC and dra_number_ranges_minus1 + 2 as the input and rangeIdx as the output.

2) The variable chromaScale is derived as follows:

$$incValue = lumaSample - OutRangesC[cIdx][rangeIdx] \quad (1384)$$

$$chromaScale = OutOffsetsC[apsId][cIdx][rangeIdx] + (OutScalesC[apsId][cIdx][rangeIdx] * incValue + (1 << 9)) >> 10 \quad (1385)$$

8.9.7 Derivation of output chroma DRA parameters

Inputs to this process are:

— array of luma scales, lumaScales[ ],

— array of output luma ranges OutRangesL[ ], and

— chroma component index cIdx.

Outputs of this process are:

— array of output chroma scales, OutScalesC[ ][ ],

— array of output chroma offset, OutOffsetsC[ ][ ], and

— array of output chroma ranges OutRangesC[ ][ ].

The variables defining the number of ranges for numRangesL and for numRangesC are set equal to dra_number_ranges_minus1 + 1 and dra_number_ranges_minus1 + 2, respectively.

For every i being in range from 0 to numRangesL – 1, inclusive, variable chromaScales[cIdx][i] is derived by invoking subclause 8.9.8 with lumaScales[i] and cIdx as inputs.

For every i being in range from 0 to numRangesL – 1, inclusive, variable invChromaScales[cIdx][i] is derived as follows:

$$invChromaScales[cIdx][i] = ((1 << 18) + (chromaScales[cIdx][i] >> 1)) / chromaScales[cIdx][i] \quad (1386)$$

Variables OutScalesC[cIdx][0], OutOffsetsC[cIdx][0] and OutRangesC[0] are set equal to 0, invChromaScales[cIdx][0] and OutRangesL[0], respectively.

For every range index i being in range from 1 to numRangesC – 1 inclusive, variable OutRangesC[i] is derived as follows:

$$OutRangesC[i] = (OutRangesL[i] + OutRangesL[i - 1]) >> 1 \quad (1387)$$

For every range index i being in range from 1 to dra_number_ranges_minus1 inclusive, variable OutScalesC[cIdx][i] is derived as follows:

$$deltaRange = OutRangesC[i + 1] - OutRangesC[i] \quad (1388)$$

$$OutOffsetsC[cIdx][i] = invChromaScales[cIdx][i - 1] \quad (1389)$$

$$deltaScale = invChromaScales[cIdx][i] - invChromaScales[cIdx][i - 1] \quad (1390)$$

$$OutScalesC[cIdx][i] = ((deltaScale << 10) + (deltaRange >> 1)) / deltaRange \quad (1391)$$

The variables OutOffsetsC[cIdx][numRangesL] and OutScalesC[cIdx][numRangesL] are derived as follows:

$$OutOffsetsC[cIdx][numRangesL] = invChromaScales[cIdx][numRangesL - 1] \quad (1392)$$

$$OutScalesC[cIdx][numRangesL] = 0 \quad (1393)$$

8.9.8 Derivation of adjusted chroma DRA scales

Inputs to this process are:

— variable denoting luma scales, lumaScale, and

— chroma component index cIdx.

Output of this process is variable denoting chroma scale chromaScale.

When DraJoinedScaleFlag is equal to 0, output value chromaScale is derived as follows:

chromaScale = (cIdx == 0) ? dra_cb_scale_value : dra_cr_scale_value (1394)

Otherwise, output value chromaScale is derived as follows:

— Variables scaleDra and scaleDraNorm are derived as follows:

scaleDra = lumaScale * ((cIdx == 0) ? dra_cb_scale_value : dra_cr_scale_value) (1395)

scaleDraNorm = (scaleDra + (1 << 8)) >> 9 (1396)

— The variable IndexScaleQP is derived by invoking subclause 8.9.5 for input value inValue set equal to scaleDraNorm, ScaleQP array and the size of the ScaleQP array, set equal to 54 as input.

— The variable qpDraInt is derived as follows:

qpDraInt = 2 * IndexScaleQP – 60 (1397)

— The variables qpDraInt, qpDraInt and qpDraFrac are derived as follows:

tableNum = scaleDraNorm – ScaleQP[IndexScaleQP] (1398)

tableDelta = ScaleQP[IndexScaleQP + 1] – ScaleQP[IndexScaleQP] (1399)

— If tableNum is equal to 0, the variable qpDraFrac is set equal to 0, and the variable qpDraInt is decreased by 1, otherwise the variables qpDraInt, qpDraFrac and draChromaQpShift are derived as follows:

qpDraFrac = (tableNum << 10) / tableDelta (1400)

qpDraInt += (qpDraFrac >> 9) (1401)

qpDraFrac = (1 << 9) – qpDraFrac % (1 << 9) (1402)

idx0 = Clip3(−QpBdOffset$_C$, 57, dra_table_idx – qpDraInt) (1403)

idx1 = Clip3(−QpBdOffset$_C$, 57, dra_table_idx – qpDraInt + 1) (1404)

qp0 = ChromaQpTable[cIdx][idx0] (1405)

qp1 = ChromaQpTable[cIdx][idx1] (1406)

qpDraIntAdj = ((qp1 – qp0) * qpDraFrac) >> 9 (1407)

qpDraFracAdj = qpDraFrac – (((qp1 – qp0) * qpDraFrac) % (1 << 9)) (1408)

draChromaQpShift = ChromaQpTable[cIdx][dra_table_idx] – qp0 – qpDraIntAdj – qpDraInt (1409)

— When qpDraFracAdj is smaller than 0, the variables draChromaQpShift and qpDraFracAdj are derived as follows:

draChromaQpShift = draChromaQpShift – 1 (1410)

qpDraFracAdj = (1 << 9) + qpDraFracAdj (1411)

— The variable draChromaScaleShift is derived as follows:

idx0 = Clip3(0, 24, draChromaQpShift + 12) (1412)

idx1 = Clip3(0, 24, draChromaQpShift + 12 – 1) (1413)

idx2 = Clip3(0, 24, draChromaQpShift + 12 + 1) (1414)

draChromaScaleShift = QpScale[idx0] (1415)

— If draChromaQpShift is less than 0, variable draChromaScaleShiftFrac is derived as follows:

draChromaScaleShiftFrac = QpScale[idx0] – QpScale[idx1] (1416)

— Otherwise, variable draChromaScaleShiftFrac is derived as follows:

draChromaScaleShiftFrac = QpScale[idx2] – QpScale[idx0] (1417)

— The variable draChromaScaleShift is modified as follows:

draChromaScaleShift =
draChromaScaleShift + (draChromaScaleShiftFrac * qpDraFracAdj + (1 << 8)) >> 9 (1418)

— Output variable chromaScale is derived as follows:

chromaScale = ((scaleDra * draChromaScaleShift) + (1 << 17)) >> 18 (1419)

— The entries of ScaleQP and QpScale tables are initialized as follows:

ScaleQP = (1420)

```
{   0,    1,     1,     1,     1,     1,     2,     2,     3,     4,     4,     6,     7,     9,
   11,   14,    18,    23,    29,    36,    45,    57,    72,    91,   114,   144,   181,   228,
  287,  362,   456,   575,   724,   912,  1149,  1448,  1825,  2299,  2896,  3649,  4598, 5793,
 7298, 9195, 11585, 14596, 18390, 23170, 29193, 36781, 46341, 58386, 73562, 92682, 116772}
```

QpScale = (1421)

```
{  128,  144,  161,  181,  203,  228,  256,  287, 322, 362, 406, 456, 512, 574, 645, 724, 812,
   912, 1024, 1149, 1290, 1448, 1625, 1825, 2048}
```

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), uek(v) (see subclause 9.2), or ae(v) (see subclause 9.3).

9.2 Parsing process for k-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), uek(v), or se(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v) or se(v) are Exp-Golomb-coded with order k equal to 0 and syntax elements coded as uek(v) are Exp-Golomb-coded with order k. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

```
leadingZeroBits = −1
for( b = 0; !b; leadingZeroBits++ )                                    (1422)
    b = read_bits( 1 )
```

The variable codeNum is then assigned as follows:

$$codeNum = (2^{leadingZeroBits} - 1) * 2^{k} + read\_bits(leadingZeroBits + k) \quad (1423)$$

where the value returned from read_bits(leadingZeroBits) is interpreted as a binary representation of an unsigned integer with the most significant bit written first.

Table 36 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 36. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as x_i in Table 36, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each x_i is equal to either 0 or 1.

Table 36 — Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges

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 37 illustrates explicitly the assignment of bit strings to codeNum values.

Table 37 — Exp-Golomb bit strings and codeNum in explicit form and used as ue(v)

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 subclause 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 subclause 9.2.1.

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 38 provides the assignment rule.

Table 38 — 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 subclauses 7.3.8.1 through 7.3.8.8.

Inputs to this process are:

— a request for a value of a syntax element, and

— the values of prior parsed syntax elements.

Output of this process is the value of the syntax element.

The initialization process as specified in subclause 9.3.2 is invoked when starting the parsing of one or more of the following:

1) The slice data syntax specified in subclause 7.3.8.1,

2) The CTU syntax specified in subclause 7.3.8.2 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 subclause 9.3.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 9.3.4.

9.3.2 Initialization process

9.3.2.1 General

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 subclause 9.3.2.2.

— The initialization process for the arithmetic decoding engine is invoked as specified in subclause 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.

If sps_cm_init_flag is equal to 0, the following applies:

— The initValue is set equal to 256.

— For each context variable, the two variables valState and valMps are initialized as follows:

```
valState = initValue
valMps = 0                                                    (1424)
```

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

— Table 40 to Table 90 contain the values of the 10-bit variable initValue used in the initialization of context variables that are assigned to all syntax elements in subclauses 7.3.8.1 through 7.3.8.8, 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 & 14) << 4
valSlope = ( initValue & 1 ) ? –valSlope : valSlope
valOffset = ( ( initValue >> 4 ) & 62 ) << 7                  (1425)
valOffset = ( ( initValue >> 4 ) & 1 ) ? –valOffset : valOffset
valOffset += 4096
```

— The two values assinged to valState and valMps for initialization 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 ) >> 4 )
valMps = prevalState > 256 ? 0 : 1                            (1426)
valState = valMps ? preValState : ( 512 – preValState )
```

— In Table 39, 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 39 — 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
coding_tree_unit( )	alf_ctb_flag[ ][ ]	na	0	1	Table 40	0	1
	alf_ctb_chroma_flag[ ][ ]	na	0	1	Table 40	0	1
	alf_ctb_chroma2_flag[ ][ ]	na	0	1	Table 40	0	1
split_unit( )	split_cu_flag[ ][ ]	na	0	1	Table 41	0	1
	btt_split_flag[ ][ ]	na	0	1	Table 42	0..14	15..29
	btt_split_dir[ ][ ]	na	0	1	Table 43	0..4	5..9
	btt_split_type[ ][ ]	na	0	1	Table 44	0	1
	split_unit_coding_order_flag[ ][ ]	na	0	1	Table 45	0..11	12..23
	pred_mode_constraint_type_flag[ ][ ]	na	0	1	Table 46	0	1
coding_unit( )	cu_skip_flag[ ][ ]	na	0	1	Table 47	0..1	2..3
	mvp_idx_l0[ ][ ] mvp_idx_l1[ ][ ]	na	0..2	3..5	Table 48	0..2	3..5
	merge_idx[ ][ ]	na	0..4	5..9	Table 49	0..4	5..9
	mmvd_flag[ ][ ]	na	0	1	Table 50	0	1
	mmvd_group_idx[ ][ ]	na	0..1	2..3	Table 51	0..1	2..3
	mmvd_merge_idx[ ][ ]	na	0..2	3..5	Table 52	0..2	3..5
	mmvd_distance_idx[ ][ ]	na	0..6	7..13	Table 53	0..6	7..13
	mmvd_direction_idx[ ][ ]	na	0..1	2..3	Table 54	0..1	2..3
	affine_flag[ ][ ]	na	0	1	Table 55	0..1	2..3
	affine_merge_idx[ ][ ]	na	0..4	5..9	Table 56	0..4	5..9
	affine_mode_flag[ ][ ]	na	0	1	Table 57	0	1
	affine_mvp_flag_l0[ ][ ] affine_mvp_flag_l1[ ][ ]	na	0	1	Table 58	0	1
	affine_mvd_flag_l0[ ][ ]	na	0	1	Table 59	0	1
	affine_mvd_flag_l1[ ][ ]	na	0	1	Table 60	0	1
	pred_mode_flag[ ][ ]	na	0	1	Table 61	0..2	3..5
	intra_pred_mode[ ][ ]	na	0..1	2..3	Table 62	0..1	2..3
	intra_luma_pred_mpm_flag[ ][ ]	na	0	1	Table 63	0	1
	intra_luma_pred_mpm_idx[ ][ ]	na	0	1	Table 64	0	1
	intra_chroma_pred_mode[ ][ ]	na	0	1	Table 65	0	1
	ibc_flag[ ][ ]	na	0	1	Table 66	0..1	2..3
	amvr_idx[ ][ ]	na	0..3	4..7	Table 67	0..3	4..7
	direct_mode_flag[ ][ ]	na	0	1	Table 68	0	1
	inter_pred_idc[ ][ ]	na	0..1	2..3	Table 69	0..1	2..3
	merge_mode_flag[ ][ ]	na	0	1	Table 70	0	1
	bi_pred_idx[ ][ ]	na	0..1	2..3	Table 71	0..1	2..3
	ref_idx_l0[ ][ ] ref_idx_l1[ ][ ]	na	0..1	2..3	Table 72	0..1	2..3
	abs_mvd_l0[ ][ ][ ] abs_mvd_l1[ ][ ][ ]	na	0	1	Table 73	0	1

Syntax structure	Syntax element	sps_cm_init_flag = = 0			sps_cm_init_flag = = 1		
		ctxTable	initType		ctxTable	initType	
			0	1		0	1
	cbf_all[ ][ ]	na	0	1	Table 74	0	1
transform_unit( )	cbf_luma	na	0	1	Table 75	0	1
	cbf_cb	na	0	1	Table 76	0	1
	cbf_cr	na	0	1	Table 77	0	1
	cu_qp_delta_abs	na	0	1	Table 78	0	1
	ats_hor_mode[ ][ ] ats_ver_mode[ ][ ]	na	0	1	Table 79	0	1
	ats_cu_inter_flag[ ][ ]	na	0	1	Table 80	0..1	2..3
	ats_cu_inter_quad_flag[ ][ ]	na	0	1	Table 81	0	1
	ats_cu_inter_horizontal_flag[ ][ ]	na	0	1	Table 82	0..2	3..5
	ats_cu_inter_pos_flag[ ][ ]	na	0	1	Table 83	0	1
residual_coding_rle( )	coeff_zero_run	na	0..3	4..7	Table 84	0..23	24..47
	coeff_abs_level_minus1	na	0..3	4..7	Table 85	0..23	24..47
	coeff_last_flag	na	0..1	2..3	Table 86	0..1	2..3
residual_coding_adv( )	last_sig_coeff_x_prefix	na	0	1	Table 87	0..20	21..41
	last_sig_coeff_y_prefix	na	0	1	Table 88	0..20	21..41
	sig_coeff_flag	na	0	1	Table 89	0..46	47..93
	coeff_abs_level_greaterA_flag coeff_abs_level_greaterB_flag	na	0	1	Table 90	0..17	18..35

Table 40 — Values of initValue for ctxIdx of alf_ctb_flag, alf_ctb_chroma_flag and alf_ctb_chroma2_flag

Initialization variable	ctxIdx of alf_ctb_flag	
	0	1
initValue	0	0

Table 41 — Values of initValue for ctxIdx of split_cu_flag

Initialization variable	ctxIdx of split_cu_flag	
	0	1
initValue	0	0

Table 42 — 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 43 — 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 44 — Values of initValue for ctxIdx of btt_split_type

Initialization variable	ctxIdx of btt_split_type	
	0	**1**
initValue	257	225

Table 45 — 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 46 — Values of initValue for ctxIdx of pred_mode_constraint_type_flag

Initialization variable	ctxIdx of pred_mode_constraint_type_flag	
	0	1
initValue	0	481

Table 47 — Values of initValue for ctxIdx of cu_skip_flag

Initialization variable	ctxIdx of cu_skip_flag			
	0	1	2	3
initValue	0	0	711	233

Table 48 — 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	0	0	0	0	0	0

Table 49 — Values of initValue for ctxIdx of merge_idx

Initialization variable	ctxIdx of merge_idx				
	0	1	2	3	4
initValue	0	0	0	496	496
	5	6	7	8	9
initValue	18	128	146	37	69

Table 50 — Values of initValue for ctxIdx of mmvd_flag

Initialization variable	ctxIdx of mmvd_flag	
	0	1
initValue	0	194

Table 51 — 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 52 — 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 53 — 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 54 — 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 55 — Values of initValue for ctxIdx of affine_flag

Initialization variable	ctxIdx of affine_flag			
	0	**1**	**2**	**3**
initValue	0	0	320	210

Table 56 — 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 57 — Values of initValue for ctxIdx of affine_mode_flag

Initialization variable	ctxIdx of affine_mode_flag	
	0	1
initValue	0	225

Table 58 — Values of initValue for ctxIdx of affine_mvp_flag_l0 and affine_mvp_flag_l1

Initialization variable	ctxIdx of affine_mvp_flag_l0 and affine_mvd_flag_l1	
	0	1
initValue	0	161

Table 59 — Values of initValue for ctxIdx of affine_mvd_flag_l0

Initialization variable	ctxIdx of affine_mvd_flag_l0	
	0	1
initValue	0	547

Table 60 — Values of initValue for ctxIdx of affine_mvd_flag_l1

Initialization variable	ctxIdx of affine_mvd_flag_l1	
	0	1
initValue	0	645

Table 61 — Values of initValue for ctxIdx of pred_mode_flag

Initialization variable	ctxIdx of pred_mode_flag					
	0	1	2	3	4	5
initValue	64	0	0	481	16	368

Table 62 — Values of initValue for ctxIdx of intra_pred_mode

Initialization variable	ctxIdx of intra_pred_mode			
	0	1	2	3
initValue	0	0	0	0

Table 63 — 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 64 — 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 65 — Values of initValue for ctxIdx of intra_chroma_pred_mode

Initialization variable	ctxIdx of intra_chroma_pred_mode	
	0	1
initValue	465	560

Table 66 — Values of initValue for ctxIdx of ibc_flag

Initialization variable	ctxIdx of ibc_flag			
	0	1	2	3
initValue	0	0	711	233

Table 67 — 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 68 — Values of initValue for ctxIdx of direct_mode_flag

Initialization variable	ctxIdx of direct_mode_flag	
	0	1
initValue	0	0

Table 69 — Values of initValue for ctxIdx of inter_pred_idc

Initialization variable	ctxIdx of inter_pred_idc			
	0	1	2	3
initValue	0	0	242	80

Table 70 — Values of initValue for ctxIdx of merge_mode_flag

Initialization variable	ctxIdx of merge_mode_flag	
	0	1
initValue	0	464

Table 71 — 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 72 — 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	0	0	288	0

Table 73 — 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	0	18

Table 74 — Values of initValue for ctxIdx of cbf_all

Initialization variable	ctxIdx of cbf_all	
	0	1
initValue	0	794

Table 75 — Values of initValue for ctxIdx of cbf_luma

Initialization variable	ctxIdx of cbf_luma	
	0	1
initValue	664	368

Table 76 — Values of initValue for ctxIdx of cbf_cb

Initialization variable	ctxIdx of cbf_cb	
	0	1
initValue	384	416

Table 77 — Values of initValue for ctxIdx of cbf_cr

Initialization variable	ctxIdx of cbf_cr	
	0	1
initValue	320	288

Table 78 — Values of initValue for ctxIdx of cu_qp_delta_abs

Initialization variable	ctxIdx of cu_qp_delta_abs	
	0	1
initValue	4	4

Table 79 — Values of initValue for ctxIdx of ats_hor_mode and ats_ver_mode

Initialization variable	ctxIdx of ats_hor_mode and ats_ver_mode	
	0	1
initValue	512	673

Table 80 — Values of initValue for ctxIdx of ats_cu_inter_flag

Initialization variable	ctxIdx of ats_cu_inter_flag	
	0	1
initValue	0	0

Table 81 — Values of initValue for ctxIdx of ats_cu_inter_quad_flag

Initialization variable	ctxIdx of ats_cu_inter_quad_flag	
	0	**1**
initValue	0	0

Table 82 — Values of initValue for ctxIdx of ats_cu_inter_horizontal_flag

Initialization variable	ctxIdx of ats_cu_inter_horizontal_flag					
	0	**1**	**2**	**3**	**4**	**5**
initValue	0	0	0	0	0	0

Table 83 — Values of initValue for ctxIdx of ats_cu_inter_pos_flag

Initialization variable	ctxIdx of ats_cu_inter_pos_flag	
	0	**1**
initValue	0	0

Table 84 — Values of initValue for ctxIdx of coeff_zero_run

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 85 — 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**
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 86 — Values of initValue for ctxIdx of coeff_last_flag

Initialization variable	ctxIdx of coeff_last_flag			
	0	**1**	**2**	**3**
initValue	421	337	33	790

Table 87 — Values of initValue for ctxIdx of last_sig_coeff_x_prefix

Initialization variable	ctxIdx of last_sig_coeff_x_prefix							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	762	310	288	828	342	451	502	51
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	97	416	662	890	340	146	20	337
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	468	216	66	54	216	892	84	581
	24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	600	278	419	372	568	408	485	338
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**
initValue	632	666	732	16	178	180	585	581
	40	**41**						
initValue	34	257						

Table 88 — Values of initValue for ctxIdx of last_sig_coeff_y_prefix

Initialization variable	ctxIdx of last_sig_coeff_y_prefix							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	81	440	4	534	406	226	370	370
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	259	38	598	792	860	312	88	662
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	924	161	248	20	54	470	376	323
	24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	276	602	52	340	600	376	378	598
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**
initValue	502	730	538	17	195	504	378	320
	40	**41**						
initValue	160	572						

Table 89 — Values of initValue for ctxIdx of sig_coeff_flag

Initialization variable	ctxIdx of sig_coeff_flag							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	387	98	233	346	717	306	233	37
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	321	293	244	37	329	645	408	493
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	164	781	101	179	369	871	585	244
b	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	361	147	416	408	628	352	406	502
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**
initValue	566	466	54	97	521	113	147	519
	40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	36	297	132	457	308	231	534	66
	48	**49**	**50**	**51**	**52**	**53**	**54**	**55**
initValue	34	241	321	293	113	35	83	226
	56	**57**	**58**	**59**	**60**	**61**	**62**	**63**
initValue	519	553	229	751	224	129	133	162
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**
initValue	227	178	165	532	417	357	33	489
	72	**73**	**74**	**75**	**76**	**77**	**78**	**79**
initValue	199	387	939	133	515	32	131	3
	80	**81**	**82**	**83**	**84**	**85**	**86**	**87**
initValue	305	579	323	65	99	425	453	291
	88	**89**	**90**	**91**	**92**	**93**		
initValue	329	679	683	391	751	51		

Table 90 — Values of initValue for ctxIdx of coeff_abs_level_greaterA_flag and coeff_abs_level_greaterB_flag

Initialization variable	ctxIdx of coeff_abs_level_greaterA_flag and coeff_abs_level_greaterB_flag							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
initValue	40	225	306	272	85	120	389	664
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	209	322	291	536	338	709	54	244
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
initValue	19	566	38	352	340	19	305	258
	24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	18	33	209	773	517	406	719	741
	32	**33**	**34**	**35**				
initValue	613	295	37	498				

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 91 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the unary binarization process, the TR binarization process, the k-th order EGK binarization process, the FL binarization process and the TB binarization process are given in subclauses 9.3.3.2 through 9.3.3.6, respectively. Other binarization are specified in subclauses 9.3.3.7 and 9.3.3.8.

Table 91 — Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	**Input parameters**
slice_data( )	end_of_tile_one_bit	FL	cMax = 1
coding_tree_unit( )	alf_ctb_flag[ ][ ]	FL	cMax = 1
	alf_ctb_chroma_flag[ ][ ]	FL	cMax = 1
	alf_ctb_chroma2_flag[ ][ ]	FL	cMax = 1

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
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
	pred_mode_constraint_type_flag[ ][ ]	FL	cMax = 1
coding_unit( )	cu_skip_flag[ ][ ]	FL	cMax = 1
	mvp_idx_l0[ ][ ]	TR	cMax = 3, cRiceParam = 0
	mvp_idx_l1[ ][ ]	TR	cMax = 3, cRiceParam = 0
	merge_idx[ ][ ]	TR	cMax = (nCbW * nCbH <= 32) ? 3 : 5, cRiceParam = 0
	mmvd_flag[ ][ ]	FL	cMax = 1
	mmvd_group_idx[ ][ ]	TR	cMax = 2, cRiceParam = 0
	mmvd_merge_idx[ ][ ]	TR	cMax = 3, cRiceParam = 0
	mmvd_distance_idx[ ][ ]	TR	cMax = 7, cRiceParam = 0
	mmvd_direction_idx[ ][ ]	FL	cMax = 3
	affine_flag[ ][ ]	FL	cMax = 1
	affine_merge_idx[ ][ ]	TR	cMax = 5, cRiceParam = 0
	affine_mode_flag[ ][ ]	FL	cMax = 1
	affine_mvp_flag_l0[ ][ ]	FL	cMax = 1
	affine_mvp_flag_l1[ ][ ]	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[ ][ ]	see 9.3.3.7	-
	direct_mode_flag[ ][ ]	FL	cMax = 1
	ibc_flag[ ][ ]	FL	cMax = 1
	amvr_idx[ ][ ]	TR	cMax = 4, cRiceParam = 0
	merge_mode_flag[ ][ ]	FL	cMax = 1
	inter_pred_idc[ ][ ]	TR	cMax = (!sps_admvp_flag || nCbW + nCbH > 12) ? 2 : 1, cRiceParam = 0
	bi_pred_idx[ ][ ]	TR	cMax = 2, cRiceParam = 0

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
	ref_idx_l0[ ][ ]	TR	cMax = num_ref_idx_active_minus1[0], cRiceParam = 0
	abs_mvd_l0[ ][ ][ ]	EG0	-
	mvd_l0_sign_flag[ ][ ][ ]	FL	cMax = 1
	affine_mvd_flag_l0[ ][ ]	FL	cMax = 1
	ref_idx_l1[ ][ ]	TR	cMax = num_ref_idx_active_minus1[1], cRiceParam = 0
	abs_mvd_l1[ ][ ][ ]	EG0	-
	mvd_l1_sign_flag[ ][ ][ ]	FL	cMax = 1
	affine_mvd_flag_l1[ ][ ]	FL	cMax = 1
	cbf_all[ ][ ]	FL	cMax = 1
transform_unit( )	cbf_luma	FL	cMax = 1
	cbf_cb	FL	cMax = 1
	cbf_cr	FL	cMax = 1
	cu_qp_delta_abs	U	-
	cu_qp_delta_sign_flag	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_rle( )	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_adv( )	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_x_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[ ]	see 9.3.3.8	baseLevel, cIdx, (xC, yC), log2TbWidth, log2TbHeight
	coeff_signs_group	FL	cMax = (1 << 16) – 1

9.3.3.2 U binarization process

Input 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 92 illustrates the bin strings of the unary binarization for a syntax element.

Table 92 — Bin string of the unary binarization

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 TR binarization process

Inputs to this process are:

— a request for a TR binarization for a syntax element with value synVal,

— a variable cMax specifying the largest possible value of the syntax element being decoded, and

— a variable cRiceParam specifying the rice parameter.

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:

$$prefixVal = synVal \gg cRiceParam \tag{1427}$$

— 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) << cRiceParam) (1428)

— The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as in subclause 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 EGK binarization process

Input 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 )
        while( k− − )
            put( ( absV >> k ) & 1 )
        stopLoop = 1
    }
} while( !stopLoop )
```
(1429)

NOTE The specification for the k-th order Exp-Golomb (EGk) code uses 1s and 0s in reverse meaning for the unary part of the Exp-Golomb code of 0-th order as specified in subclause 9.2.2.

9.3.3.5 FL binarization process

Inputs to this process are:

— a request for an FL binarization for a syntax element, and

— a variable cMax specifying the largest possible value of the syntax element being decoded.

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 TB binarization process

Inputs to this process are:

— a request for a TB binarization for a syntax element with value synVal, and

— a variable cMax specifying the largest possible value of the syntax element being decoded.

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:

$$\begin{aligned} n &= cMax + 1 \\ k &= Floor(Log2(n)) \\ u &= (1 \ll (k + 1)) - n \end{aligned} \tag{1430}$$

— If synVal is less than u, the TB bin string is derived by invoking the FL binarization process as specified in subclause 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 as specified in subclause 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 93.

Table 93 — Binarization for intra_chroma_pred_mode

Value of intra_chroma_pred_mode	Bin string
0	1
1	00
2	010
3	0110
4	0111

9.3.3.8 Binarization process for coeff_abs_level_remaining

Inputs to this process are:

— a request for a binarization for the syntax element coeff_abs_level_remaining[n],

— a variable baseLevel,

— a variable cIdx specifying the colour component,

— a location (xC, yC) specifying the current coefficient scan location, and

— two variables log2TbWidth and log2TbHeight specifying the binary logarithm of the transform block width and height.

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 subclause 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 94.

Table 94 — Specification of cRiceParam based on locSumAbs

cRiceParam	0	1	2	3
numBinRem	6	5	6	3

The variable cMax is derived from cRiceParam as:

$$cMax = numBinRem << cRiceParam \quad (1431)$$

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:

$$prefixVal = Min(cMax, coeff\_abs\_level\_remaining[n]) \quad (1432)$$

— The prefix bin string is specified by invoking the TR binarization process as specified in subclause 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:

$$suffixVal = coeff\_abs\_level\_remaining[n] - cMax \quad (1433)$$

— The suffix bin string is specified by invoking the limited k-th order EGk binarization process as specified in subclause 9.3.3.4 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1 and cRiceParam as inputs.