INTERNATIONAL STANDARD

ISO/IEC 14496-33

First edition
2019-02

Information technology — Coding of audio-visual objects —

Part 33:
Internet video coding

Technologies de l'information — Codage des objets audiovisuels —

Partie 33: Codage vidéo Internet

Reference number
ISO/IEC 14496-33:2019(E)

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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of 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).

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

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

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

This document specifies Internet video coding, a video compression technology that is intended to be suitable for video distribution models currently adopted on the Internet.

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 patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patent rights are registered with ISO and IEC. Information may be obtained from:

Nokia Technologies Oy

Joensuunkatu 7E

FIN-24100 Salo

FINLAND

Telephone : +358 50 366 2022

Apple Inc.

Intellectual Property and Licensing

1 Infinite Loop, MS 169-3IPL

Cupertino, CA 95014

USA

Telephone: +1(408) 974-0015

Industry-University Cooperation Foundation Hanyang University

222 Wangsimni-ro, Seongdong-gu

Seoul 04763

REPUBLIC OF KOREA

Telephone: +82-2-2220-2212

Mitsubishi Electric Corporation
Corporate Licensing Division
2-7-3 Marunouchi, Chiyoda-ku
Tokyo 100-8310
JAPAN
Telephone: +81-3-3218-3465

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121
USA
Telephone: +1 (858) 587-1121

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

Information technology — Coding of audio-visual objects —

Part 33: Internet video coding

1 Scope

This document specifies MPEG-4 Internet video coding.

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.

Rec. ITU-T H.262 | ISO/IEC 13818-2: 2013, *Information technology — Generic coding of moving pictures and associated audio information — Part 2: Video*

IEC 60461, *Time and control code*

3 Terms and definitions

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

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

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

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

3.1
B frame
bidirectional frame
frame (3.28) that is coded using motion compensated prediction from past or future *reference frames* (3.53) in *output order* (3.40)

3.2
backward prediction
process of predicting the current *frame* (3.28) by using future *frames* in an *output order* (3.40) as *reference frames* (3.53)

3.3
bin
bit of a *bin string* (3.4)

3.4
bin string
intermediate binary representation of values of *syntax elements* (3.65) resulting from the *binarization* (3.5) of the *syntax element*

3.5
binarization
set of *bin strings* (3.4) for all possible values of a *syntax element* (3.65)

3.6
binarization process
unique mapping process of all possible values of a *syntax element* (3.65) onto a set of *bin strings* (3.4)

3.7
bitstream
ordered series of bits that forms the *coded representation* (3.14) of the data

3.8
block
MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients* (3.66)

3.9
byte
sequence of 8 bits, written and read with the most significant bit on the left and the least significant bit on the right, such that when represented in a sequence of data bits, the most significant bit of a byte is first

3.10
byte-aligned
positioning of a bit or *byte* (3.9) or *syntax element* (3.65) when the position at which it appears in a *bitstream* (3.7) is an integer multiple of 8 bits from the position of the first bit in the *bitstream*

3.11
byte stream
ordered series of bytes that forms the *coded representation* (3.14) of the data

3.12
chroma
sample array or single sample, identified symbolically by Cb or Cr, representing one of the two colour difference signals related to the primary colours

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 is often associated with the term chrominance.

3.13
coded frame
coded representation (3.14) of a *frame* (3.28)

3.14
coded representation
series of data elements as represented in coded form in the *bitstream* (3.7)

3.15
component
array or single sample from one of the three arrays (*luma* (3.37) and two *chroma* (3.12)) that make up a *frame* (3.28) in 4:2:0 colour format

3.16
DC coefficient
transform coefficient (3.66) for which the *frequency index* (3.27) is zero in all dimensions

3.17
decoded frame
frame (3.28) derived by decoding a *coded frame* (3.13)

3.18
decoder
embodiment of the *decoding process* (3.20)

3.19
decoding order
order in which syntax elements are processed by the *decoding process* (3.20)

3.20
decoding process
process that derives *decoded frames* (3.17) from the syntax elements in the *bitstream* (3.7)

3.21
dequantization
process of *scaling* (3.57) the *quantized transform coefficients* (3.49) after their representation in the *bitstream* (3.7) has been *parsed* (3.42) and before they are presented to the *inverse transform* (3.34) part of the *decoding process* (3.20)

3.22
encoder
embodiment of an *encoding process* (3.23)

3.23
encoding process
process that produces a *bitstream* (3.7)

Note 1 to entry: This document does not specify an encoding process.

3.24
forbidden
specification that a value shall never be used

Note 1 to entry: This is usually to avoid emulation of a *start code* (3.63) pattern.

3.25
forward prediction
process of predicting the current frame by the past *reference frames* (3.53) in output order

3.26
flag
binary variable that can take one of the two possible values, 0 and 1

3.27
frequency index
one-dimensional or two-dimensional index associated with a *transform coefficient* (3.66) prior to an *inverse transform* (3.34) part of a *decoding process* (3.20)

3.28
frame
successive lines, numbered from the top-most line to the bottom-most line, containing samples numbered from the left-most sample to the right-most sample, representing the spatial information of a video signal from a single time instant

3.29
I frame
intra frame
frame (3.28) coded using information only from itself

3.30
inter macroblock
macroblock (3.38) which is coded using *inter prediction* (3.31)

3.31
inter prediction
prediction (3.44) derived from data elements (e.g. sample value or *motion vector* (3.39)) of *reference frames* (3.53) other than the current frame

3.32
intra macroblock
macroblock (3.38) which is coded using *intra prediction* (3.33)

3.33
intra prediction
prediction (3.44) derived from only data elements (e.g. sample values) of the same decoded *slice* (3.60)

3.34
inverse transform
part of the *decoding process* (3.20) by which a set of *transform coefficients* (3.66) are converted into spatial-domain values, or by which a set of *transform coefficients* are converted into *DC coefficients* (3.16)

3.35
layer
one of a set of syntactical structures in a non-branching hierarchical relationship, such that higher layers contain lower layers, with such coded layers being the *coded frame* (3.13), *slice* (3.60), *macroblock* (3.38) and *block* (3.8)

3.36
level
defined set of constraints on the values that may be taken by *syntax elements* (3.65) and variables; or in a different context, the value of a *transform coefficient* (3.66) prior to *scaling* (3.57)

Note 1 to entry: The same set of levels is defined for all *profiles* (3.47), with most aspects of the definition of each level being in common across different *profiles.* Individual implementations may, within specified constraints, support a different level for each supported *profile*.

3.37
luma
sample array or single sample, identified symbolically by Y or L, ordinarily representing the brightness signal related to the primary colours

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 is often associated with the term luminance. The symbol L is sometimes used instead of the symbol Y to avoid confusion with the symbol y as used for vertical location.

3.38
macroblock
16 × 16 *luma* (3.37) sample value block and its corresponding two *chroma* (3.12) sample value blocks

3.39
motion vector
two-dimensional vector used for *inter prediction* (3.31) that provides an offset from the coordinates in the *decoded frame* (3.17) to the coordinates in a *reference frame* (3.53)

3.40
output order
order in which the *decoded frames* (3.17) are output from the *decoded frame* buffer in case the *decoded frames* are to be output from the *decoded frame* buffer

3.41
P frame
predictive frame
frame (3.28) that is coded using motion compensated prediction from past *reference frames* (3.53) in *output order* (3.40)

3.42
parse
procedure of obtaining the value of a *syntax element* (3.65) from a *bitstream* (3.7)

3.43
partitioning
division of a set into subsets such that each element of a set is in exactly one of the subsets

3.44
prediction
embodiment of a *prediction process* (3.45)

3.45
prediction process
use of a *predictor* (3.46) to provide an estimate of a data element (e.g. sample value or *motion vector* (3.39)) currently being decoded

3.46
predictor
combination of specified values or previously decoded data elements (e.g. sample value or *motion vector* (3.39)) used in the *decoding process* (3.20) of subsequent data elements

3.47
profile
specified subset of the syntax

3.48
quantization parameter
variable used by the *decoding process* (3.20) for *scaling* (3.57) of *transform coefficient levels* (3.67)

3.49
quantized transform coefficients
transform coefficients (3.66) before *dequantization* (3.21)

3.50
random access
starting the *decoding process* (3.20) for part of a *bitstream* at some point other than the beginning of the *bitstream* (3.7)

3.51
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 top-most row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc., top-most rows of the pattern (proceeding downwards), with each row scanned from left to right

3.52
reference index
order indication of the *reference frames* (3.53) in the frame buffer in the *decoding process* (3.20)

3.53
reference frame
frame that contains samples that may be used for *inter prediction* (3.31) in the *decoding process* (3.20) of subsequent *frames* (3.28) in *decoding order* (3.19)

3.54
reserved
specification that some values of a particular *syntax element* (3.65) are for future use by ISO/IEC, such that these values shall not be used in *bitstreams* (3.7), but may be specified for use in future extensions by ISO/IEC

3.55
residual
decoded difference between a *prediction* (3.44) of a sample or data element and its decoded value

3.56
run
number of data elements with the same value or the same treatment in the *decoding process* (3.20)

Note 1 to entry: In one context, it means the number of zero coefficients before a non-zero coefficient in the block scan, and in another context, it means the number of consecutive *skipped macroblocks* (3.59).

3.57
scaling
process of multiplying *transform coefficient levels* (3.67) by a factor, resulting in *transform coefficients* (3.66)

3.58
sequence
highest layer syntax structure of the *bitstream* (3.7), including one or more consecutive *coded frames* (3.13)

3.59
skipped macroblock
macroblock (3.38) for which no syntax elements are present in the *bitstream* (3.7) except for the indication that the *macroblock* is a *skipped macroblock* (3.59)

3.60
slice
integer number of consecutive *macroblock* (3.38) rows in the *raster scan* (3.51) order that is associated with the same header data

3.61
slice header
part of a coded *slice* (3.60) containing the data elements pertaining to the first or all *macroblocks* (3.58) represented in a *slice*

3.62
source
video material or some of its attributes before operation of an *encoding process* (3.23)

3.63
start code
32-bit codeword pattern which is unique in the whole *bitstream* (3.7)

Note 1 to entry: Start codes can be used to identify the starting point of a syntax structure in the *bitstream* (e.g. to enable *random access* (3.50)).

3.64
stuffing bits
bit string having a prescribed pattern of fixed values at a particular position in the *bitstream* (3.7)

3.65
syntax element
element of data represented in the *bitstream* (3.7)

3.66
transform coefficient
scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* (3.27) in an *inverse transform* (3.34) part of the *decoding process* (3.20)

3.67
transform coefficient level
integer quantity representing the value associated with a particular two-dimensional frequency index in the *decoding process* (3.20) prior to *scaling* (3.57) for computation of a *transform coefficient* value (3.66)

3.68
video buffering verifier
hypothetical reference *decoder* (3.18) that operates on the *bitstream* (3.7) to perform the *decoding process* (3.20) with a specified timing and with a specified limited capacity for buffering the coded data and *decoded frames* (3.17)

Note 1 to entry: Its purpose is to provide a constraint on the variability of the data rate that an *encoder* (3.22) or editing process may produce.

4 Abbreviations

LSB	least significant bit
MB	macroblock
MSB	most significant bit
VBV	video buffering verifier

5 Conventions

NOTE The mathematical operators and their precedence rules used in this document are similar to those used in the C programming language. However, operators of integer divisions with truncation and of rounding are specifically defined. If not specifically explained, numbering and counting begin from zero.

5.1 Arithmetic operators

+	Addition
−	Subtraction (as a binary operator) or negation (as a unary prefix operator)
*	Multiplication
a^b	Exponential operation: a is raised to power of b. (May alternatively represent a superscript.)
/	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 formulae where no truncation or rounding is intended.
$\frac{a}{b}$	Division in mathematical formulae where no truncation or rounding is intended.
$\sum_{i=a}^{b} f(i)$	The summation of *f*(i) with i taking integral values from a up to and including b.
a % b	Remainder of a divided by b, defined only for a >= 0 and b > 0.

5.2 Logical operators

a && b	Logical AND operation between a and b
a || b	Logical OR operation between a and b
!	Logical NOT operation
a ? b : c	If a is TRUE or not equal to 0, evaluates to b; otherwise, evaluates to c.

5.3 Relational operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

5.4 Bitwise operators

&	AND operation
|	OR operation
~	Negation operation
a >> b	Shift a in 2's complement binary integer representation format to the right by b bit positions. This operator is only defined with b, a positive integer.
a << b	Shift a in 2's complement binary integer representation format to the left by b bit positions. This operator is only defined with b, a positive integer.

5.5 Assignment

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

5.6 Order of operation precedence

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

— operations of a higher precedence are evaluated before any operation of a lower precedence;

— operations of the same precedence are evaluated sequentially from left to right.

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

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

Table 1 — Operation precedence from highest (at top of table) to lowest (at bottom of table)

operations (with operands x, y, and z)
"x++", "x−−"
"!x", "−x" (as a unary prefix operator)
"x^y"
"x * y", "x / y", "x ÷ y", "$\frac{x}{y}$", "x % y"
"x + y", "x − y" (as a two-argument operator), "$\sum_{i=a}^{b} f(i)$"
"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"

5.7 Mathematical functions

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

ceil(x) Takes the smallest integer not smaller than x

clip1(x) = clip3(0, 255, x)

clip3(a, b, c) = $\begin{cases} a & ; \ c < a \\ b & ; \ c > b \\ c & ; \ \text{else} \end{cases}$

floor(x) Takes the biggest integer not bigger than x

log2(x)	Logarithm number of x with base 2
log10(x)	Logarithm number of x with base 10
median(x, y, z) =	x + y + z − min(x, min(y, z)) − max(x, max(y, z))
min(x, y) =	$\begin{cases} \text{x} & ; \; x <= y \\ \text{y} & ; \; x > y \end{cases}$
max(x, y) =	$\begin{cases} \text{x} & ; \; x >= y \\ \text{y} & ; \; x < y \end{cases}$
round(x) =	sign(x) * floor(abs(x) + 0.5)
sign(x) =	$\begin{cases} 0 & ; \; x >= 0 \\ 1 & ; \; x < 0 \end{cases}$
InverseRasterScan (a, b, c, d, e)=	$\begin{cases} \left(a\%\left(d/b\right)\right)*b & ; \; e == 0 \\ \left(a/\left(d/b\right)\right)*b & ; \; e == 1 \end{cases}$

5.8 Variables, syntax elements and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lowercase letters with underscore characters), its one or two syntax categories, and one or two descriptors 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 element values. Such variables appear in the syntax tables, or text, named by a mixture of lowercase and uppercase letter and without any underscore characters. Variables starting with an uppercase letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an uppercase letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lowercase letter are only used within the subclause 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 uppercase letter and may contain more uppercase 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 5.11.2 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 5.7) are described by their names, which start with an uppercase letter, contain a mixture of lower and uppercase 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}.

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.

5.9 Text description of logical operations

In the text, a statement of logical operations as would be described in pseudo-code as:

```
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 in pseudo-code as:

```
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 any 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 in pseudo-code as:

```
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

5.10 Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. All syntax elements and uppercase 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 lowercase variable explicitly specified as the input. Each process specification has explicitly specified an output. The output is a variable that can either be an uppercase variable or a lowercase 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 lowercase input or output variables of the process specification.

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

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

5.11 Description of bitsteam syntax parsing process and decoding process

5.11.1 Method of describing bitstream syntax

The description style of the syntax is similar to C programming language.

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

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

Table 2 lists examples of pseudo code used to describe the syntax. 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 2 — Examples of pseudo code

	Descriptor
/* A statement can be a syntax element with associated descriptor or can be an expression used to specify its existence, type, and value, as in the following examples */	
syntax_element	ue(v)
conditioning statement	
/* A group of statements enclosed in brackets is a compound statement and is treated functionally as a single statement. */	
{	
statement	
statement	
…	
}	
/* A "while" structure specifies that the statement is to be evaluated repeatedly while the condition remains true. */	
while (condition)	
statement	
/* A "do … while" structure executes the statement once, and then tests the condition. It repeatedly evaluates the statement while the condition remains true. */	
do	
statement	
while (condition)	
/* An "if … else" structure tests the condition first. If it is true, the primary statement is evaluated. Otherwise, the alternative statement is evaluated. If the alternative statement is unnecessary to be evaluated, the "else" and corresponding alternative statement can be omitted. */	
if(condition)	
primary statement	
else	
alternative statement	
/* A "for" structure evaluates the initial statement at the beginning then tests the condition. If it is true, the primary and subsequent statements are evaluated until the condition becomes false. */	
for (initial statement; condition; subsequent statement)	
primary statement	

Parsing and decoding process are described using text and C-like pseudo language.

5.11.2 Syntax functions

Functions used for syntax description are explained in this section. It is assumed that the decoder has a bitstream position indicator. This bitstream position indicator locates the position of the bit that is going to be read right next. A function consists of its name and a sequence of parameters inside of parentheses. A function may not have any parameters.

byte_aligned( )

The function byte_aligned () returns TRUE if the current position is on a byte boundary. Otherwise, it returns FALSE.

next_bits(*n*)

The function returns the next n bits from the bitstream, MSB first. The current bitstream position indicator is not changed. If the remaining number of bits to be read are less than n, it returns 0.

byte_aligned_next_bits(*n*)

If the current position of the bitstream is not byte-aligned, returns n bits beginning from the next byte-aligned position, MSB first. The current bitstream position indicator is not changed. If the current position of the bitstream is byte-aligned, returns n bits from the current position, MSB first. The current bitstream position is not changed. If the remaining number of bits to be read is less than n, it returns 0.

next_start_code( )

The next_start_code() function locates the next start code. It is defined in Table 3.

Table 3 — next_start_code() function

next_start_code() {	Descriptor
stuffing_bit	'1'
stuffing_bit	'1'
while (! byte_aligned())	
stuffing_bit	'0'
while (next_bits(24) != '0000 0000 0000 0000 0000 0001')	
stuffing_byte	'0000 0000'
}	

is_end_of_slice( )

This function tests if the current position is at the end of the slice. The function's definition is shown in Table 4.

Table 4 — Function's definition of the end of the slice

is_end_of_slice () {	Descriptor
if(byte_aligned ( )) {	
if(next_bits(32) ==0xc0000001)	
return TRUE /* end of slice */	
}	
else {	
if(((byte_aligned_next_bits(24) == 0x000001) || (byte_aligned_next_bits(32) == 0x80000001)) && is_stuffing_pattern())	
return TRUE /* end of slice */	

Table 4 *(continued)*

}	
return FALSE	
}	

is_stuffing_pattern( )

This function tests whether the remaining bits of the current byte or the next byte (in case the current position is byte-aligned), are stuffing bits. The function's definition is shown in Table 5.

Table 5 — Function's definition of stuffing bits

is_stuffing_pattern () {	descriptor
if((n == 7) && (next_bits(1) == 1)) /* n, in the range 0..7, is the bitstream position indicator in the current byte, when n is 0, the bitstream position indicator indicates the MSB of the current byte. */	
return TRUE	
else if((n <7) && (next_bits(8−n) == ((1<< (7−n)) + (1<< (6−n)))))	
return TRUE	
else	
return FALSE	
}	

read_bits(*n*)

This function returns n bits of the bitstream from the current position, MSB first. The bitstream position indicator advances n bits. If n is equal to 0, the function returns 0, and the bitstream position indicator does not move.

Syntax functions can be also used for describing parsing process and decoding process.

5.11.3 Syntax descriptors

The descriptors below specify the parsing process of syntax elements.

b(8)

A byte with arbitrary value (8 bits). The parsing process for this descriptor is specified by the return value of read_bits(8).

f(n)

A bit string with n bits. The parsing process is specified by the returned value of read_bits(n).

i(n)

Signed integer with n bits. In syntax table, if n is 'v', the number of bits is determined by other syntax elements. The parsing process is specified by the return value of read_bits(n), interpreted as two's complement representation with MSB first.

r(n)

A bit string with n bits equal to '0'. The parsing process is specified by the returned value of read_bits(n).

u(n)

Unsigned integer with n bits. In syntax table, if n is 'v', the number of bits is determined by other syntax elements. The parsing process is specified by the returned value of read_bits(n), interpreted as two's complement representation with MSB first.

ue(v)

Unsigned integer Exp-Golomb(Exponential Golomb) coded syntax element with the first bit on left. The parsing process is specified in subclause 9.2.

5.11.4 Reserved, forbidden and marker bit

In the bitstream syntax defined by this document, the value of some syntax elements is marked as 'reserved' or 'forbidden'.

The term 'reserved', when used in the clauses specifying some values of a particular syntax element, is for future uses. These values shall not be used in the bitstreams conforming to this document, but may be used in future extensions or revisions of this document.

The term 'forbidden' specifies some values of syntax elements that shall not be used in the bitstreams conforming to this document. marker_bit specifies a bit with value '1'. reserved_bits specifies that some particular syntax elements are used for future extension of this document. The decoding process shall ignore these bits.

6 Source, coded, decoded and output data formats

6.1 Source

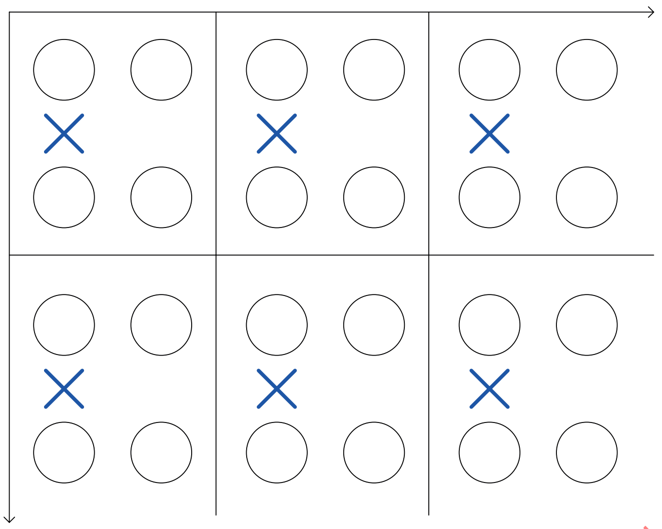
This document only deals with coding of progressive-scanned video sequences, and each picture in the video sequence is a frame. The sequence, at the output of the decoding process, consists of a series of reconstructed frames that are separated in time by a frame period which is the inverse of a specified frame rate.

A frame consists of three matrices of integer samples: a luma sample matrix (Y), and two chroma sample matrices (Cb and Cr).

Each element of each colour component matrix has an integer value.

6.2 Colour format

This document only deals with 4:2:0 colour format, in which the two chroma colour component matrices have half the number of samples of the corresponding luma colour component matrix both horizontally and vertically. The luma and chroma samples are positioned as shown in Figure 1.



Key

○	luma sample
×	chroma sample

Figure 1 — Position of luma and chroma samples in 4:2:0 format

6.3 Coded bitstream format

The highest syntactic structure of the coded video bitstream is the video sequence. A video sequence commences with a sequence header which is followed by one or more coded frames. In front of each frame, a frame header is present. The order of the coded frames in the coded bitstream is the bitstream order. The bitstream order is same as the decoding order. The decoding order is not necessarily same as the output order. The video sequence is terminated by a sequence_end_code.

6.4 Sequence header

A video sequence header commences with sequence header start code and is followed by a series of coded frame data. A sequence header is allowed to be repeatedly present in the bitstream. This sequence header is called repeat sequence header. The main purpose of repeat sequence header is providing with random access functionality. The first coded frame after a sequence header shall be an I frame. The P frames after a sequence header only refer to frames appeared after the sequence header. It is a requirement of bitstream conformance that if the bitstream is modified by removing all of the data preceding any of the repeat sequence headers, then the resulting bitstream shall be a legal bitstream that conforms to this document.

6.5 Frame

A reconstructed frame is obtained by decoding a coded frame, i.e. a frame header, the optional extensions immediately following it, and the frame data.

6.6 Frame types

There are three types of frames that use different coding methods:

— an **Intra-coded (I) frame** is coded using information only from itself;

— a **Predictive-coded (P) frame** is a frame which is coded using motion compensated prediction from past reference frames;

— a **Bidirectionally predictive-coded (B) frame** is a frame which is coded using motion compensated prediction from past or future reference frames.

This document defines three sub-types of P frames, which can be used for P frame coding in low delay cases as shown in the Table 6. A non-reference P frame is not used as a reference frame for motion compensated inter-frame prediction. A non-reference P frame with reference frame buffer (RPB) swapping is referred as a non-reference P frame accompanied with the operation of RPB swapping. After decoding a non-reference P frame with RPB swapping, the last two decoded frames placed in RPB shall exchange their positions in the buffer.

Table 6 — P frame sub-types

Name	Value
P frame	1
Non-reference P frame	2
Non-reference P frame with RPB swapping	3

6.7 Slice

A slice is a series of an arbitrary number of consecutive macroblocks. The first and last macroblocks of a slice shall not be skipped macroblocks, it is a requirement of bitstream conformance that mb_part_type is not equal to zero when the macroblock is at the first or last position in the slice. Every slice contains at least one macroblock. Slices shall not overlap. The position of slices may change from frame to frame. Slices shall occur in the bitstream in the order in which they are encountered, starting at the upper-left of the frame and proceeding by raster-scan order from left to right and top to bottom.

6.8 Macroblock

A slice is partitioned into macroblocks. A macroblock contains a section of the luma component and the spatially corresponding chroma components. The term macroblock can either refer to source and decoded data or to the corresponding coded data elements. A macroblock consists of 6 8x8 blocks. This structure holds 4 Y, 1 Cb and 1 Cr blocks and the block order is depicted in Figure 2.

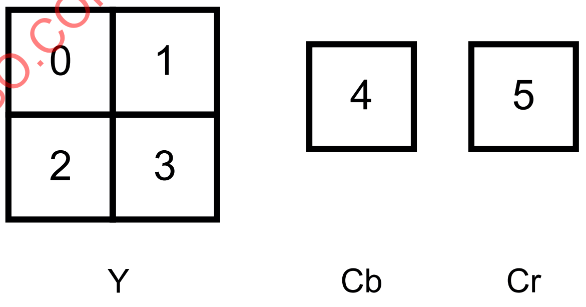


Figure 2 — Partitioning of a macroblock into 8x8 blocks (4:2:0 format)

6.9 Block

The term “block” can refer either to source and reconstructed data or to the transform coefficients or to the corresponding coded data elements.

When “block” refers to source and reconstructed data, it refers to an orthogonal section of a luma or chroma component with the same number of lines and samples.The size of a block can be either 4x4, 8x8 or 16x16.

6.10 Frame re-ordering

When the sequence contains coded B frames, the number of consecutive coded B frames is variable and shall be less than 127. The first coded frame after a sequence header shall not be a B frame.

The order of the coded frames in the bitstream, also called coded order, is the order in which a decoder reconstructs them. The order of the reconstructed frames at the output of the decoding process, also called the output order, is not always the same as the coded order and this subclause defines the rules of frame re-ordering between the decoder input and decoder output.

When the sequence contains no coded B frames, the coded order is the same as the output order. This is true in particular always when low_delay is one. When B frames are present in the sequence, re-ordering shall be performed to produce the output order according to the following rules:

— If the current frame in coded order is a B frame, the output frame is the frame reconstructed from that B frame;

— If the current frame in coded order is an I frame or P frame, the output frame is the frame reconstructed from the previous I frame or P frame if one exists. If none exists, at the start of the sequence, no frame is output.

The frame reconstructed from the final I frame or P frame is output immediately after the frame reconstructed when the last coded frame in the sequence was removed from the VBV buffer.

The following Figure 3 is an example for explaining re-ordering: there are two coded B frames between successive coded P frames. The P frame with only intra coded blocks is marked as "I". Frame '1I' is used to form a prediction for frame '4P'. Frames '4P' and '1I' are both used to form predictions for frames '2B' and '3B'. Therefore the order of coded frames in the coded sequence is '1I', '4P', '2B', '3B'. However, the decoder outputs them in the order '1I', '2B', '3B', '4P'.

At the encoder input:

1	2	3	4	5	6	7	8	9	10	11	12	13
I	B	B	P	B	B	P	B	B	I	B	B	P

At the encoder output, in the coded bitstream, and at the decoder input:

1	4	2	3	7	5	6	10	8	9	13	11	12
I	P	B	B	P	B	B	I	B	B	P	B	B

At the decoder output:

1	2	3	4	5	6	7	8	9	10	11	12	13
I	B	B	P	B	B	P	B	B	I	B	B	P

Figure 3 — Frame re-ordering example

6.11 Reference frames

P frame can use up to eight (at maximum) past frames as reference. B frame can refer up to one forward reference frame or one backward reference frame.

6.12 Inverse scanning processes and derivation processes for neighbours

6.12.1 General

This subclause specifies inverse scanning processes, i.e., the mapping of indices to locations, and derivation processes for neighbours.

6.12.2 Inverse macroblock scanning process

Input to this process is a macroblock address mbAddr.

Output of this process is the location (x, y) of the upper-left luma sample for the macroblock with address mbAddr relative to the upper-left sample of the frame.

The inverse macroblock scanning process is specified as follows:

— x = InverseRasterScan(mbAddr, 16, 16, PicWidth, 0);

— y = InverseRasterScan(mbAddr, 16, 16, PicWidth, 1).

6.12.3 Inverse macroblock partition scanning process

Macroblocks may be partitioned, and the partitions are scanned for inter prediction as shown in Figure 4. The outer rectangles refer to the samples in a macroblock. The rectangles refer to the partitions. The number in each rectangle specifies the index of the inverse macroblock partition scan.

The functions MbPartWidth(), and MbPartHeight() describing the width and height of macroblock partitions are specified in Table 16, and Table 17. MbPartWidth() and MbPartHeight() are set to appropriate values for each macroblock partition, depending on the macroblock partition type (denoted by mb_part_type).

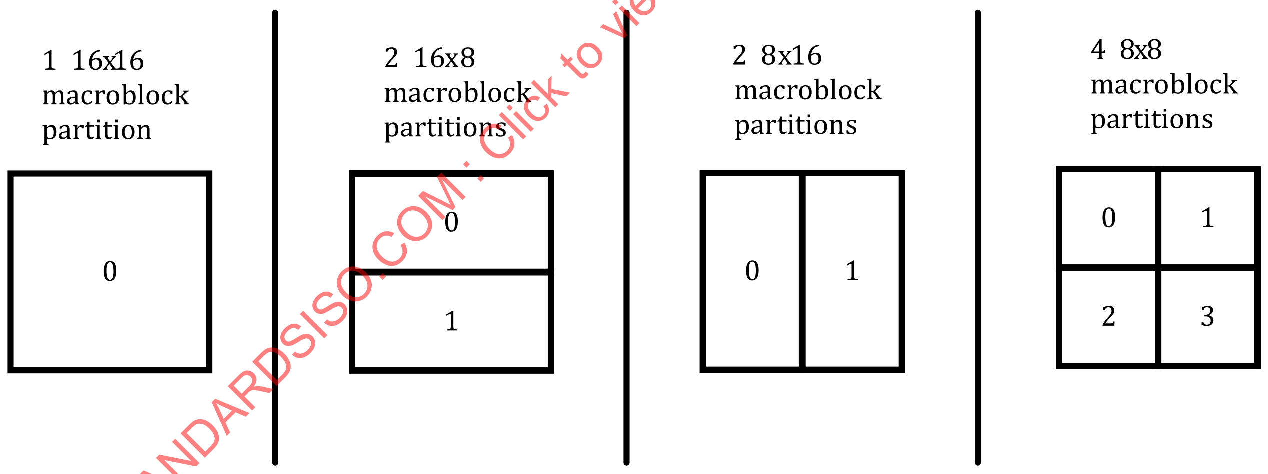

1 16x16 macroblock partition
0
2 16x8 macroblock partitions
0
1
2 8x16 macroblock partitions
0
1
4 8x8 macroblock partitions
0
1
2
3

Figure 4 — Macroblock partitions

Input to this process is the index of a macroblock partition mbPartIdx.

Output of this process is the location (x, y) of the upper-left luma sample for the macroblock partition mbPartIdx relative to the upper-left sample of the macroblock.

The inverse macroblock partition scanning process is specified by:

— x = InverseRasterScan(mbPartIdx, MbPartWidth(mb_part_type), MbPartHeight(mb_part_type), 16, 0);

— y = InverseRasterScan(mbPartIdx, MbPartWidth(mb_part_type), MbPartHeight(mb_part_type), 16, 1).

6.12.4 Inverse 8x8 luma block scanning process

Input to this process is the index of an 8x8 luma block luma8x8BlkIdx within a 16x16 luma block.

Output of this process is the location (x, y) of the upper-left luma sample for the 8x8 luma block with index luma8x8BlkIdx relative to the upper-left luma sample of the16x16 luma block.

Figure 5 shows the scan order for the 8x8 luma blocks.

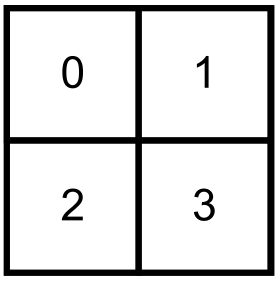


Figure 5 — Scan order for 8x8 luma blocks

The inverse 8x8 luma block scanning process is specified as follows.

— x = InverseRasterScan(luma8x8BlkIdx, 8, 8, 16, 0);

— y = InverseRasterScan(luma8x8BlkIdx, 8, 8, 16, 1).

6.12.5 Inverse 4x4 luma block scanning process

Input to this process is the index of a 4x4 luma block luma4x4BlkIdx within an 8x8 luma block.

Output of this process is the location (x, y) of the upper-left luma sample for the 4x4 luma block with index luma4x4BlkIdx relative to the upper-left luma sample of the 8x8 block.

Figure 6 shows the scan order for the 4x4 luma blocks.

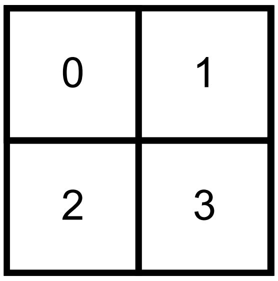


Figure 6 — Scan order for 4x4 luma blocks

The inverse 4x4 luma block scanning process is specified as follows.

— x = InverseRasterScan(luma4x4BlkIdx, 4, 4, 8, 0)

— y = InverseRasterScan(luma4x4BlkIdx, 4, 4, 8, 1)

6.12.6 Derivation process of the availability for macroblock addresses

Input to this process is a macroblock address mbAddr.

Output of this process is the availability of the macroblock mbAddr.

NOTE The meaning of availability is determined when this process is invoked.

The macroblock is marked as available, unless one of the following conditions is true in which case the macroblock is marked as not available:

— mbAddr < 0;

— mbAddr > CurrMbAddr;

— the macroblock with address mbAddr belongs to a different slice than the current slice.

6.12.7 Derivation process for neighbouring macroblock addresses and their availability

The outputs of this process are:

— mbAddrA: the address and availability status of the macroblock to the left of the current macroblock;

— mbAddrB: the address and availability status of the macroblock above the current macroblock;

— mbAddrC: the address and availability status of the macroblock above-right of the current macroblock;

— mbAddrD: the address and availability status of the macroblock above-left of the current macroblock.

Figure 7 shows the relative spatial locations of the macroblocks with mbAddrA, mbAddrB, mbAddrC, and mbAddrD relative to the current macroblock with CurrMbAddr.

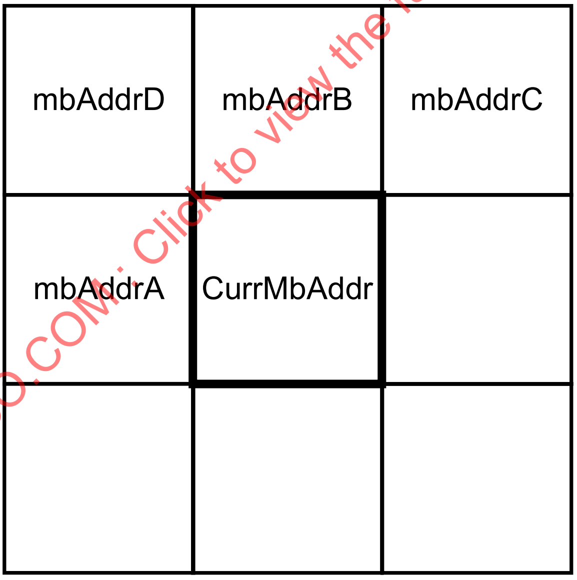


Figure 7 — Neighbouring macroblocks for a given macroblock

Input to the process in subclause 6.12.6 is mbAddrA = CurrMbAddr − 1 and the output is whether the macroblock mbAddrA is available. In addition, mbAddrA is marked as not available when CurrMbAddr % PicWidthInMbs is equal to 0.

Input to the process in subclause 6.12.6 is mbAddrB = CurrMbAddr − PicWidthInMbs and the output is whether the macroblock mbAddrB is available.

Input to the process in subclause 6.12.6 is mbAddrC = CurrMbAddr − PicWidthInMbs + 1 and the output is whether the macroblock mbAddrC is available. In addition, mbAddrC is marked as not available when (CurrMbAddr + 1) % PicWidthInMbs is equal to 0.

Input to the process in subclause 6.12.6 is mbAddrD = CurrMbAddr − PicWidthInMbs − 1 and the output is whether the macroblock mbAddrD is available. In addition, mbAddrD is marked as not available when CurrMbAddr % PicWidthInMbs is equal to 0.

6.12.8 Derivation processes for neighbouring macroblocks, blocks, and partitions

6.12.8.1 General

Subclause 6.12.8.2 specifies the derivation process for neighbouring macroblocks.

Subclause 6.12.8.3 specifies the derivation process for neighbouring 8x8 luma blocks.

Subclause 6.12.8.4 specifies the derivation process for neighbouring partitions.

Table 8 specifies the values for the difference of luma location (xD, yD) for the input and the replacement for N in mbAddrN, mbPartIdxN, and luma8x8BlkIdxN for the output.

These input and output assignments are used in subclauses 6.12.8.2 to 6.12.8.4. The variable predPartWidth is specified when Table 7 is referred to.

Table 7 — Specification of input and output assignments for subclauses 6.12.8.2 to 6.12.8.4

N	xD	yD
A	−1	0
B	0	−1
C	predPartWidth	−1
D	−1	−1

Figure 8 illustrates the relative location of the neighbouring macroblocks, blocks, or partitions A, B, C, and D to the current macroblock, partition, or block.

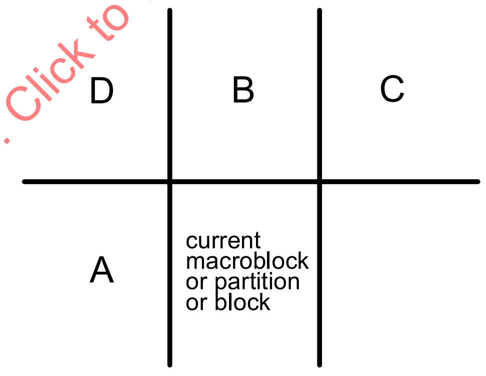


Figure 8 — Determination of the neighbouring macroblock, blocks, and partitions (informative)

6.12.8.2 Derivation process for neighbouring macroblocks

Outputs of this process are:

— mbAddrA: the address of the macroblock to the left of the current macroblock and its availability status; and

— mbAddrB: the address of the macroblock above the current macroblock and its availability status.

mbAddrN (with N being A or B) is derived as follows.

— The difference of luma location (xD, yD) is set according to Table 7.

— The derivation process for neighbouring locations as specified in subclause 6.12.9 is invoked for luma locations with (xN, yN) equal to (xD, yD), and the output is assigned to mbAddrN.

6.12.8.3 Derivation process for neighbouring 8x8 luma block

Input to this process is an 8x8 luma block index luma8x8BlkIdx.

The luma8x8BlkIdx specifies the 8x8 luma blocks of a macroblock in a raster scan.

Outputs of this process are:

— mbAddrA: either equal to CurrMbAddr or the address of the macroblock to the left of the current macroblock and its availability status;

— luma8x8BlkIdxA: the index of the 8x8 luma block to the left of the 8x8 block with index luma8x8BlkIdx and its availability status;

— mbAddrB: either equal to CurrMbAddr or the address of the macroblock above the current macroblock and its availability status;

— luma8x8BlkIdxB: the index of the 8x8 luma block above the 8x8 block with index luma8x8BlkIdx and its availability status.

mbAddrN and luma8x8BlkIdxN (with N being A or B) are derived as follows.

— The difference of luma location (xD, yD) is set according to Table 7.

— The luma location (xN, yN) is specified by:

  — xN = (luma8x8BlkIdx % 2) * 8 + xD;

  — yN = (luma8x8BlkIdx / 2) * 8 + yD.

— The derivation process for neighbouring locations as specified in subclause 6.12.9 is invoked for luma locations with (xN, yN) as the input and the output is assigned to mbAddrN and (xW, yW).

— The variable luma8x8BlkIdxN is derived as follows.

  — If mbAddrN is not available, luma8x8BlkIdxN is marked as not available.

  — Otherwise (mbAddrN is available), the 8x8 luma block in the macroblock mbAddrN covering the luma location (xW, yW) is assigned to luma8x8BlkIdxN.

6.12.8.4 Derivation process for neighbouring partitions

Inputs to this process are

— a macroblock partition index mbPartIdx.

Outputs of this process are:

— mbAddrA\mbPartIdxA: specifying the macroblock partition to the left of the current macroblock and its availability status;

— mbAddrB\mbPartIdxB: specifying the macroblock partition above the current macroblock and its availability status;

— mbAddrC\mbPartIdxC: specifying the macroblock partition to the right-above of the current macroblock and its availability status;

— mbAddrD\mbPartIdxD: specifying the macroblock partition to the left-above of the current macroblock and its availability status.

mbAddrN and mbPartIdxN (with N being A, B, C, or D) are derived as follows.

— The inverse macroblock partition scanning process as described in subclause 6.12.3 is invoked with mbPartIdx as the input and (x, y) as the output.

— The variable predPartWidth in Table 7 is specified as follows.

predPartWidth = MbPartWidth(mb_part_type).

— The difference of luma location (xD, yD) is set according to Table 7.

— The neighbouring luma location (xN, yN) is specified by:

  — xN = x + xD;

  — yN = y + yD.

— The derivation process for neighbouring locations as specified in subclause 6.12.9 is invoked for luma locations with (xN, yN) as the input and the output is assigned to mbAddrN and (xW, yW).

— Depending on mbAddrN, the following applies.

  — If mbAddrN is not available, the macroblock partition mbAddrN\mbPartIdxN is marked as not available.

  — Otherwise (mbAddrN is available), the following applies.

    — The macroblock partition in the macroblock mbAddrN covering the luma location (xW, yW) is assigned to mbPartIdxN.

    — When the partition given by mbPartIdxN is not yet decoded, the macroblock partition mbPartIdxN is marked as not available.

6.12.9 Derivation process for neighbouring locations

Input to this process is a luma or chroma location (xN, yN) expressed relative to the upper left corner of the current macroblock.

Outputs of this process are:

— mbAddrN: either equal to CurrMbAddr or to the address of neighbouring macroblock that contains (xN, yN) and its availability status;

— (xW, yW): the location (xN, yN) expressed relative to the upper-left corner of the macroblock mbAddrN (rather than relative to the upper-left corner of the current macroblock).

Let maxWH be a variable specifying a maximum value of the location components xN, yN, xW, and yW. maxWH is derived as follows.

— If this process is invoked for neighbouring luma locations:

  maxWH = 16

— Otherwise (this process is invoked for neighbouring chroma locations):

  maxWH = 8

The derivation process for neighbouring macroblock addresses and their availability in subclause 6.12.7 is invoked with mbAddrA, mbAddrB, mbAddrC, and mbAddrD as well as their availability status as the output.

Table 8 specifies mbAddrN depending on (xN, yN).

Table 8 — Specification of mbAddrN

xN	yN	mbAddrN
< 0	< 0	mbAddrD
< 0	0..maxWH − 1	mbAddrA
0..maxWH − 1	< 0	mbAddrB
0..maxWH − 1	0..maxWH − 1	CurrMbAddr
> maxWH − 1	< 0	mbAddrC
> maxWH − 1	0..maxWH − 1	not available
	> maxWH − 1	not available

The neighbouring luma location (xW, yW) relative to the upper-left corner of the macroblock mbAddrN is derived as:

— xW = (xN + maxWH) % maxWH;

— yW = (yN + maxWH) % maxWH.

7 Syntax and semantics

7.1 Bitstream syntax

7.1.1 Start codes

Start codes are specific bit patterns that do not otherwise occur in the video bitstream.

Each start code consists of a start code prefix followed by a start code suffix. The start code prefix is a byte-aligned string of twenty-three bits with the value zero followed by a single bit with the value one. The start code prefix is thus the the byte-aligned bit string ‘0000 0000 0000 0000 0000 0001’.

The start code suffix is an eight-bit integer which denotes the type of start code. Most types of start code have just one associated value of the start code suffix. However, a slice_start_code may have a start code suffix value in the range of 0x00 to 0xAF; in this case the start code suffix value is interpreted as the slice_vertical_position syntax element for the slice.

Table 9 specifies the start code suffix values that are allowed in the video elementary bitstream.

Table 9 — Start codes and start code suffix values

Start code name	Suffix value (Hexadecimal)
slice_start_code	00..AF
video_sequence_start_code	B0
video_sequence_end_code	B1
user_data_start_code	B2
i_frame_start_code	B3
reserved	B4
reserved	B5
pb_frame_start_code	B6
video_edit_code	B7
reserved	B8

When considering the values that may occur in the syntax elements of the bitstream, it is important to ensure that the pattern of bits that represents other syntax elements cannot emulate the start code prefix pattern. It is thus a requirement of bitstream conformance that the bit string pattern of a start code prefix (i.e., the bit string ‘0000 0000 0000 0000 0000 0001’) shall not occur in the bitstream in any byte-aligned position other than the positions in which start codes are specified to appear in the bitstream syntax specification that follows.

At the beginning of the decoding process, the decoder initializes its current position in the byte stream to the beginning of the bitstream. It then extracts and discards each zero byte (if present), moving the current position in the bitstream forward one byte at a time, until the current position in the byte stream is such that the next three bytes in the bitstream form the bit string ‘0000 0000 0000 0000 0000 0001’.

The decoder then performs the following process repeatedly to extract and decode the byte stream until the end of byte stream has been encountered.

— The next three-byte sequence in the byte stream is extracted and the current position in the byte stream is set equal to the position of the byte following this three-byte sequence.

— If the current position in the byte stream is such that the next three bytes in the bitstream form the bit string ‘0000 0000 0000 0000 0000 0010’, these three bytes are extracted, the two LSBs of these three bytes are dropped, and the current position in the byte stream is set equal to the position of the byte following this three-byte sequence; otherwise, the current byte is extracted and the current position is moved forward to the position following this byte.

— When one of the following conditions is met, the extracted bitstream segment is decoded by the decoding process:

  1) A subsequent byte-aligned three-byte sequence equal to 0x000000, or

  2) A subsequent byte-aligned three-byte sequence equal to 0x000001, or

  3) The end of the byte stream, as determined by unspecified means.

— When the current position in the byte stream is not at the end of the byte stream (as determined by unspecified means) and the next bytes in the byte stream do not start with a three-byte sequence equal to 0x000001, the decoder extracts and discards each zero byte syntax element, moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next bytes in the byte stream form the three-byte sequence 0x000001 or the end of the byte stream has been encountered (as determined by unspecified means).

7.1.2 Video sequence

7.1.2.1 Sequence

video_sequence() {	**descriptor**
do {	
start_code_prefix	f(24)
start_code_type	f(8)
if(start_code_type != video_sequence_end_code && start_code_type != video_edit_code) {	
if(start_code_type == video_sequence_start_code)	
sequence_header( )	
else if(start_code_type == user_data_start_code)	
user_data()	
else if(start_code_type == i_frame_start_code)	
i_frame_header()	
else if(start_code_type == pb_frame_start_code)	

pb_frame_header()	
else if(start_code_type == slice_start_code)	
slice()	
}	
} while ((next_bits(24) == '0000 0000 0000 0000 0000 0001'))	

7.1.2.2 Sequence header

sequence_header() {	**descriptor**
profile_id	u(8)
level_id	u(8)
horizontal_size	u(14)
vertical_size	u(14)
chroma_format	u(2)
sample_precision	u(3)
aspect_ratio	u(4)
frame_rate_code	u(4)
bit_rate_lower	u(18)
marker_bit	f(1)
bit_rate_upper	u(12)
low_delay	u(1)
marker_bit	f(1)
vbv_buffer_size	u(18)
abt_enable	u(1)
if_type	u(1)
reserved_bits	r(4)
next_start_code()	
}	

7.1.2.3 User data

user_data() {	**descriptor**
while (next_bits(24) ! = '0000 0000 0000 0000 0000 0001') {	
user_data_byte	b(8)
}	
}	

7.1.3 Frame

7.1.3.1 I Frame header

i_frame_header() {	**descriptor**
vbv_delay	u(16)
time_flag	u(1)
if(time_flag == '1') { /* Time code syntax elements */	
drop_frame_flag	u(1)

time_code_hours	u(5)
time_code_minutes	u(6)
time_code_seconds	u(6)
time_code_frames	u(6)
}	
marker_bit	f(1)
frame_distance	u(8)
if(low_delay == '1')	
vbv_check_times	ue(v)
fixed_frame_level_qp	u(1)
if(fixed_frame_level_qp)	
frame_qp	u(6)
reserved_bits	r(4)
loop_filter_disable	u(1)
if(!loop_filter_disable) {	
alpha_threshold	u(8)
beta_threshold	u(6)
}	
next_start_code()	
}	

7.1.3.2 PB Frame header

pb_frame_header() {	**descriptor**
vbv_delay	u(16)
frame_coding_type	u(2)
frame_sub_type	u(2)
frame_distance	u(8)
if(low_delay == '1')	
vbv_check_times	ue(v)
fixed_frame_level_qp	u(1)
if(fixed_frame_level_qp)	
frame_qp	u(6)
no_forward_reference_flag	u(1)
reserved_bits	r(3)
loop_filter_disable	u(1)
if(!loop_filter_disable) {	
alpha_threshold	u(8)
beta_threshold	u(6)
}	
next_start_code()	
}	

7.1.4 Slice

slice() {	**Descriptor**
if(vertical_size > 2800)	
slice_vertical_position_extension	u(3)
if(!fixed_frame_level_qp) {	
fixed_slice_level_qp	u(1)
slice_qp	u(6)
}	
do {	
if(!is_end_of_slice()) {	
macroblock()	
aec_mb_stuffing_bit	ae(v)
}	
} while (!is_end_of_slice())	
next_start_code()	
}	

7.1.5 Macroblock

7.1.5.1 General

macroblock() {	**descriptor**
if(FrameType != 0) { /* 0: I frame */	
mb_part_type	ae(v)
if((FrameType == 2) && (MbPartType != 'B_Skip')) { /* 2: B frame */	
if(MbPartType == 'B_16x16')	
mb_pred_type	ae(v)
else if((MbPartType == 'B_16x8') || (MbPartType == 'B_8x16')) {	
mb_pred_type	ae(v)
mb_pred_type	ae(v)
}	
else if(MbPartType == 'B_8x8') {	
mb_pred_type	ae(v)
mb_pred_type	ae(v)
mb_pred_type	ae(v)
mb_pred_type	ae(v)
}	
}	
else if(FrameType == 1) { /* 1: P frame */	
if(MbPartType == 'P_16x16')	
mb_pred_type	ae(v)
else if((MbPartType == 'P_16x8') || (MbPartType == 'P_8x16')) {	
mb_pred_type	ae(v)
mb_pred_type	ae(v)

}	
else if(MbPartType == ‘P_8x8’) {	
mb_pred_type	ae(v)
mb_pred_type	ae(v)
mb_pred_type	ae(v)
mb_pred_type	ae(v)
}	
} /* P frame */	
}	
if(MbPartType == ‘I_Block ’) /* intra macroblock */	
mb_trans_type	ae(v)
if((FrameType == 1) && (RefPicNumber > 1)) {	
if(MbPartType != ‘I_ Block’) {	
for (i=0; i<MvNum; i++)	
reference_frame_index	ae(v)
}	
}	
if(MbPartType == ‘I_Block’) {	
if(mb_trans_type == 0) /* 16x16 */	
intra_luma_pred_mode	ae(v)
else {	
for (i=0; i<4; i++) { /* 8x8 */	
submb_trans_type	ae(v)
if(submb_trans_type) {	
for(j = 0; j < 4; j++)	
intra_luma_pred_mode	ae(v)
}	
else	
intra_luma_pred_mode	ae(v)
}	
}	
intra_chroma_pred_mode	ae(v)
}	
else{	
for (i = 0; i < MvNum; i++) {	
mv_diff_x	ae(v)
mv_diff_y	ae(v)
}	
}	
coded_block_pattern()	
if((MbCBP > 0) && (! FixedQP))	

mb_qp_delta	ae(v)
block()	
}	

NOTE MvNum = MbPartMvNum *(16*16) / (MbPartWidth(mb_part_type) * MbPartHeight(mb_part_type))

7.1.5.2 Coded block pattern

coded_block_pattern() {	
if(mb_trans_type == 0) { /* 16x16 */	
cbp_luma_bit	ae(v)
if(cbp_luma_bit)	
MbCBP = 0xFFFF	
}	
else {	
for (i=0; i<4; i++) { /* 8x8 */	
if(submb_trans_type) {	
for(j = 0; j < 4; j++) {	
cbp_luma_bit	ae(v)
MbCBP += cbp_bit << (4*i+j)	
}	
}	
else {	
cbp_luma_bit	ae(v)
if(cbp_luma_bit)	
MbCBP += 0xF << (4 * i)	
}	
}	
cbp_chroma_bit	ae(v)
if(cbp_chroma_bit) {	
cbp_chroma_allnonzero_bit	ae(v)
if(cbp_chroma_allnonzero_bit)	
MbCBP += 0xFF0000	
else {	
cbp_chroma_nonzero_bit	ae(v)
cbp_chroma_nonzero_bit)	
MbCBP += 0xF00000	
else	
MbCBP += 0xF0000	
}	
}	
}	

7.1.6 Block

block() {	**Descriptor**
if(mb_trans_type == 16x16) {	
if(MbCBP & 1) {	
do {	
trans_coefficient	ae(v)
} while (trans_coefficient != ‘EOB’)	
}	
}	
else {	
if(submb_trans_type == 0) { /* 8x8 */	
for(i = 0; i < 4; i++) {	
if(MbCBP & (1<< i*4)) {	
do {	
trans_coefficient	ae(v)
} while (trans_coefficient != ‘EOB’)	
}	
}	
}	
else{ /* 4x4 */	
for (i = 0; i < 4; i++) {	
for(j = 0; j < 4; j++) {	
if(MbCBP &(1 << (i*4 + j))) {	
do {	
trans_coefficient	ae(v)
} while (trans_coefficient != ‘EOB’)	
}	
}	
}	
}	
}	
for(i = 0; i < 2; i++) { /* chroma */	
if(MbCBP & (1 << (16+4*i)) {	
do {	
trans_coefficient	ae(v)
} while (trans_coefficient != ‘EOB’)	
}	
}	
}	

7.2 Video bitstream semantics

7.2.1 Start code

start_code_prefix – the bit string ‘0x000001’. It indicates the prefix of a start code.

start_code_type – an 8-bit unsigned integer. It indicates the type of header.

7.2.2 Video sequence

7.2.2.1 Sequence header

profile_id – an 8-bit unsigned integer. It indicates the profile of a bitstream, as specified in subclause 10.2.

level_id – an 8-bit unsigned integer. It indicates the level of a bitstream, as specified in subclause 10.3.

horizontal_size – a 14-bit unsigned integer. It specifies the width of the displayable part of the luma component of the frames in samples. The width of the corresponding displayable part of the chroma component of the frames in samples is horizontal_size / 2. In order ensure that the chroma width is exactly half the luma width, horizontal_size shall be a multiple of 2. In order to avoid start code emulation and null video content, horizontal_size shall not be zero. The displayable part is left-aligned in the encoded frames.

The width of the encoded frames in macroblocks, PicWidthInMbs, is (horizontal_size + 15) / 16.

vertical_size – a 14-bit unsigned integer. It specifies the height of the displayable part of the luma component of the frames in samples. The height of the corresponding displayable part of the chroma component of the frames in samples is vertical_size / 2. In order ensure that the chroma width is exactly half the luma width, horizontal_size shall be a multiple of 2. In order to avoid start code emulation and null video content, vertical_size shall not be zero. The displayable part is top-aligned in the encoded frames.

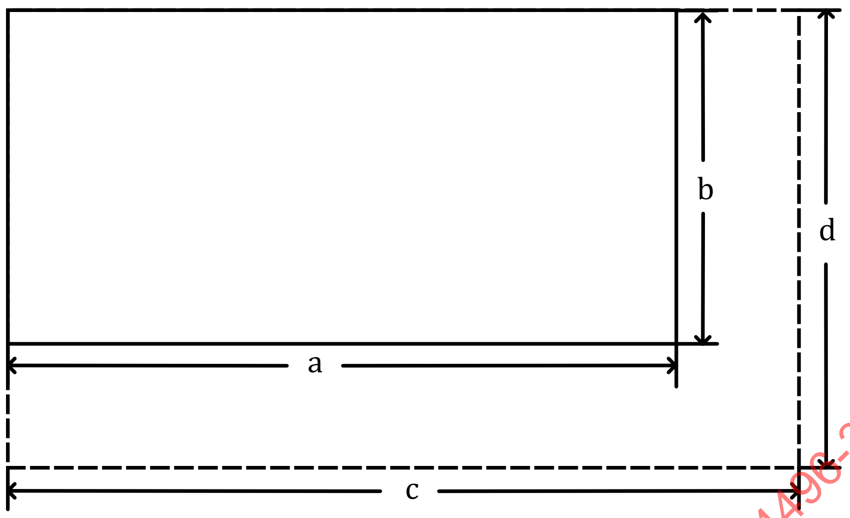
The height of the encoded frames in macroblocks, PicHeightInMbs, is (vertical_size + 15) / 16.

The width and height of the luma component of the coded frames are calculated by:

— PicWidth = PicWidthInMbs * 16;

— PicHeight = PicHeightInMbs * 16.

NOTE The relation of horizontal_size, vertical_size and frame boundaries is shown in Figure 9. Solid lines represent the boundaries of the displayable area, for which the luma width and the height are horizontal_size and vertical_size, respectively; dash lines represent the boundaries of the encoded frame, for which the width and the height are PicWidth and PicHeight, respectively. For example, if the horizontal_size is 1920, and vertical_size is 1080, the PicWidth is 1920, and the PicHeight is 1088.



Key

a	horizontal_size
b	vertical_size
c	PicWidth
d	PicHeight

Figure 9 — Luma component frame boundaries

chroma_format – a 2-bit unsigned integer. It specifies the chroma component format. Refer to Table 10 for its semantics. 01 indicates 4:2:0 format, and other values are reserved for future use.

Table 10 — Chroma format

chroma_format	Description
00	Reserved
01	4:2:0
10	Reserved
11	Reserved

sample_precision – a 3-bit unsigned integer. It specifies the precision of luma and chroma samples. Refer to Table 11 for its semantics. 001 indicates the precision of luma and chroma sample is 8-bit, and other values are reserved for future use.

Table 11 — Sample precision

sample_precision	Description
000	Forbidden
001	The precision of luma and chroma sample is 8-bit.
010 ... 111	Reserved

aspect_ratio – a 4-bit unsigned integer. It specifies the sample aspect ratio (SAR) or display aspect ratio (DAR) of reconstructed frames. Refer to Table 12 for its semantics.

Table 12 — Aspect ratio information

aspect_ratio	SAR	DAR
0000	Forbidden	Forbidden
0001	1.0	—
0010	—	4 ÷ 3
0011	—	16 ÷ 9
0100	—	2.21 ÷ 1
0101 – 1111	—	Reserved

The whole reconstructed frame is mapped to the whole active display area as follows:

SAR = (DAR * horizontal_size) ÷ vertical_size

NOTE horizontal_size and vertical_size are restricted by the SAR and selected DAR of a source frame.

frame_rate_code – a 4-bit unsigned integer. It specifies the frame rate. Refer to Table 13 for its semantics.

Table 13 — Frame rate codes

frame_rate_code	Frame rate
0000	Forbidden
0001	24000 ÷ 1001 (23.967...)
0010	24
0011	25
0100	30000 ÷ 1001 (29.97...)
0101	30
0110	50
0111	60000 ÷ 1001 (59.94...)
1000	60
1001 – 1111	Reserved

The time interval between two successive frames is reciprocal of frame rate.

bit_rate_lower – the low-order 18 bits of BitRate.

bit_rate_upper – the high-order 12 bits of BitRate.

BitRate = (bit_rate_upper << 18) + bit_rate_lower

BitRate is calculated in units of 400 bits/s and it expresses a ceiling on the video bit rate. BitRate shall not be 0.

low_delay – flag. '1' indicates that the sequence does not contain any B frames, that the frame re-ordering delay is not present.

vbv_buffer_size – a 18-bit unsigned integer. It specifies the requirement for bitstream buffer size of VBV for decoding. BBS is the minimum bitstream buffer size in bits for video decoding, and it is calculated by

BBS = 16 * 1024 * vbv_buffer_size

The VBV operation and associated conformance requirements are specified by Rec. ITU-T H.262 | ISO/IEC 13818-2:2013, Annex C.

abt_enable – flag. '1' indicates that either 16x16, 8x8 or 4x4 transform can be used in transform coding, "0" means only 8x8 transform is used in transform coding.

if_type – flag. ‘1’ indicates that either 4-tap, 6-tap or 10-tap filter can be used in luma component interpolation, ‘0’ means that only 6-tap filter is used in luma component interpolation. These interpolation filters are specified in subclause 8.3.3.3.

7.2.2.2 User_data

user_data_byte – an 8-bit integer. User data is defined by users for their specific applications. In the series of consecutive user_data bytes there shall not be a bit string of 23 or more consecutive zero bits.

7.2.3 Frame

7.2.3.1 I Frame header

vbv_delay – a 16-bit unsigned integer. In all cases other than when vbv_delay has the value hexadecimal FFFF, the value of vbv_delay is the number of periods of a 90 kHz clock derived from the 27 MHz system clock that the VBV waits after receiving the final byte of the frame start code before decoding the frame. vbv_delay shall be coded to represent the delay as specified above or it shall be coded with the value hexadecimal FFFF. If any vbv_delay field in a sequence is coded with hexadecimal FFFF, then all of them shall be coded with this value. The VBV operation and associated conformance requirements are specified by Rec. ITU-T H.262 | ISO/IEC 13818-2:2013, Annex C.

time_flag – flag. ‘1’ indicates that drop_frame_flag, time_code_hours, time_code_minutes, time_code_seconds, and time_code_frames are present in the bitstream, ‘0’ indicates that these syntax elements are not present in the bitstream.

drop_frame_flag, **time_code_hours**, **time_code_minutes**, **time_code_seconds**, and **time_code_frames** are unsigned integers that correspond to those defined in IEC 60461. These syntax elements are associated with the first frame in display order after a sequence header. The range of allowed values for these syntax elements is shown in Table 14.

Table 14 — Time code syntax elements

Syntax element	Value	Descriptor
drop_frame_flag	0, 1	u(1)
time_code_hours	0..23	u(5)
time_code_minutes	0..59	u(6)
time_code_seconds	0..59	u(6)
time_code_frames	0..59	u(6)

frame_distance – an 8-bit unsigned integer. It specifies the frame order of I or P frame, in modulo 256 operation.

vbv_check_times – If low_delay is equal to ‘0’, vbv_check_times is not present in the bitstream and VbvCheckTimes is set to 0. If vbv_check_times is present in the bitstream, VbvCheckTimes is obtained with parsing vbv_check_times. The value of vbv_check_times shall be less than 2^{16}–1. VbvCheckTimes plus 1 indicates the times VBV buffer has been checked. The VBV operation and associated conformance requirements are specified by Rec. ITU-T H.262 | ISO/IEC 13818-2:2013, Annex C.

fixed_frame_level_qp – flag. ‘1’ indicates the quantization parameter does not change in the frame, ‘0’ indicates the quantization parameter may change. The fixed quantization parameter flag FixedQP is set to fixed_frame_level_qp after fixed_frame_level_qp is parsed.

frame_qp – a 6-bit unsigned integer. It specifies the quantization parameter of the frame, ranging from 0 to 63 inclusive.

loop_filter_disable – flag. It specifies whether the operation of de-blocking filter is disabled. ‘1’ indicates the de-blocking filter operation is disabled, ‘0’ indicates the de-blocking filter is used.

alpha_threshold– a 8-bit unsigned integer. It specifies a threshold of level difference between the border samples across one block edge.

beta_threshold – a 6-bit unsigned integer. It specifies a threshold of level difference between the border samples on the same side of one block edge.

7.2.3.2 PB frame header

frame_coding_type – a 2-bit unsigned integer. It specifies the coding type of a frame. Its semantics are defined in Table 15.

Table 15 — Coding type of a frame

frame_coding_type	Coding type	FrameType
00	Forbidden	-
01	Forward inter prediction (P)	1
10	Bidirectional inter prediction (B)	2
11	Reserved	-

frame_sub_type – a 2-bit unsigned integer. It specifies the sub-type of P frames. Its semantics are defined in Table 6.

no_forward_reference_flag – flag. ‘1’ indicates that current frame does not use past reference frames for forward prediction, ‘0’ indicates that current frame can use past reference frames for forward prediction.

See subclause 7.2.3.1 for other syntax elements of PB frame header.

7.2.4 Slice

slice_vertical_position_extension – a 3-bit unsigned integer. If vertical_size of a coded frame is less than or equal to 2800, slice_vertical_position_extension shall not be present in the bitstream.

MbRow that indicates the number of macroblock rows in the current slice is derived by:

```
if( vertical_size > 2800 )
    MbRow = ( slice_vertical_position_extension << 7 ) + slice_vertical_position
else
    MbRow = slice_vertical_position
```

fixed_slice_level_qp – flag. ‘1’ indicates that the quantization parameter in the slice does not change, while ‘0’ indicates that the quantization parameter may change. The fixed quantization parameter flag FixedQP is equal to fixed_slice_level_qp after fixed_slice_level_qp is parsed.

slice_qp – a 6-bit unsigned integer. It specifies the quantization parameter of a slice, ranging from 0 to 63, inclusive.

aec_mb_stuffing_bit – flag. The aec_mb_stuffing_bit of the last macroblock of a slice shall be ‘1’.

7.2.5 Macroblock

7.2.5.1 General

mb_part_type – It indicates the partition type of a macroblock. The semantics depends on the frame coding type.

Tables and semantics are specified for the various partition types for macroblocks in P and B frames. Each table presents the value and name of mb_part_type (given by the MbPartType), the width of macroblock partitions(given by the MbPartWidth(mb_part_type)), and the height of macroblock partition (given by the MbPartHight(mb_part_type)).

— If current frame is a P frame,

  Refer to Table 16 for the semantics of mb_part_type.

— Otherwise, if current frame is a B frame,

  Refer to Table 17 for the semantics of mb_part_type.

Table 16 — MbPartTypes of macroblocks in P frames

mb_part_type	MbPartType	MbPartWidth(mb_part_type)	MbPartHight(mb_part_type)
0	P_16x16	16	16
1	P_8x16	8	16
2	P_16x8	16	8
3	I_Block	16	16
4	P_8x8	8	8

Table 17 — MbPartTypes of macroblocks in B frames

mb_part_type	MbPartType	MbPartWidth(mb_part_type)	MbPartHight(mb_part_type)
0	B_Skip	16	16
1	B_16x16	16	16
2	B_8x16	8	16
3	B_16x8	16	8
4	I_Block	16	16
5	B_8x8	8	8

mb_trans_type – flag. It indicates the transform type of an intra macroblock (given by the MbTransformType). If mb_trans_type is 0, the MbTransformType is set equal to 'Trans_16x16'. Otherwise, the MbTransformType is set equal to 'Trans_8x8'.

For the inter macroblock, the MbTransformType is determined by the value of mb_part_type. If both the MbPartWidth(mb_part_type) and the MbPartHight(mb_part_type) are 16, the MbTransformType is set equal to 'Trans_16x16'. Otherwise, the MbTransformType is set equal to 'Trans_8x8'.

reference_frame_index – It indicates the reference frame index of a macroblock partition.

RefPicNumber is a variable to indicate the number of available reference frames in reference frame buffer. It is initialized to 0 at the beginning of one video sequence. RefPicNumber is updated as specified in subclause 8.6.

mb_pred_type – It indicates the inter prediction type of each macroblock partition.

Tables 18 to 21 specify the semantics for the various inter prediction types for macroblock partitions in P and B frames. Each table lists the value and name of mb_pred_type (given by the MbPredType), the number of motion vectors of a macroblock partition in the bitstream (given by the MbPartMvNum), and the prediction mode of a macroblock partition (given by the MbPartPredMode).

— If current frame is a P frame,

  Refer to Table 18 and Table 21 for the semantics of mb_pred_type.

— Otherwise, if current frame is a B frame,

Refer to Table 20 and Table 21 for the semantics of mb_pred_type.

Table 18 — MbPredTypes of P_16x16 macroblock

mb_pred_type	MbPredType	MbPartMvNum	MbPartPredMode
0	Pred_Skip	0	Forward
2	Pred_Fwd	1	Forward
3	Pred_Mh	1	Forward

Table 19 — MbPredTypes of P_16x8, P_8x16, and P_8x8 macroblocks

mb_pred_type	MbPredType	MbPartMvNum	MbPartPredMode
0	Pred_Fwd	1	Forward
1	Pred_Mh	1	Forward

Table 20 — MbPredTypes of B_16x16, B_16x8, and B_8x16 macroblocks

mb_pred_type	MbPredType	MbPartMvNum	MbPartPredMode
0	Pred_Bck	1	Backward
2	Pred_Fwd	1	Forward
3	Pred_Sym	1	Bidirectional

Table 21 — MbPredTypes of B_8x8 macroblock

mb_pred_type	MbPredType	MbPartMvNum	MbPartPredMode
0	Pred_Skip	0	Bidirectional
1	Pred_Fwd	1	Forward
2	Pred_Bck	1	Backward
3	Pred_Sym	1	Bidirectional

submb_trans_type – flag. It indicates the transform type of an 8x8 block (given by the SubMbTransformType). If submb_trans_type is 0, the SubMbTransformType is set equal to ‘Trans_8x8’. Otherwise, the SubMbTransformType is set equal to ‘Trans_4x4’.

intra_luma_pred_mode – It specifies the type of intra prediction used for luma blocks (the block size can be either 16x16, 8x8 or 4x4).

intra_chroma_pred_mode – It specifies the type of intra prediction used for chroma blocks (the block size is 8x8).

mv_diff_x – the horizontal motion vector component difference. it is in one-quarter luma sample units, in range from −4096 to 4095 (the range is −1024 to 1023.75 in luma sample units).

mv_diff_y – the vertical motion vector component difference. it is in one-quarter luma sample units, in range from −4096 to 4095 (the range is −1024 to 1023.75 in luma sample units).

mb_qp_delta – It indicates the increment of current quantization parameters relative to predicted quantization parameters.

7.2.5.2 Coded block pattern

cbp_luma_bit – flag. It is used to indicate whether a luma block (the block size can be either 16x16, 8x8 or 4x4) contains nonzero quantized coefficients. If cbp_luma_bit is 1, the luma block contains nonzero quanization coefficients; otherwise, the luma block contains all-zero quantized coefficients.

cbp_chroma_bit – flag. It is used to indicate whether both the Cb and Cr chroma blocks (the block size is 8x8) in a macroblock contains all-zero quantized coefficients. If cbp_chroma_bit is 1, the Cb or Cr chroma blocks in a macroblock contain nonzero quantized coefficients; otherwise, both the Cb and Cr chroma blocks in a macroblock contain all-zero quantized coefficients.

cbp_chroma_allnonzero_bit – flag. It is used to indicate whether both the Cb and Cr chroma blocks (the block size is 8x8) in a macroblock contains nonzero quantized coefficients. If cbp_chroma_allnonzero_bit is 1, both the Cb and Cr chroma blocks in a macroblock contain nonzero quantized coefficients.

cbp_chroma_nonzero_bit – flag. It is used to indicate which chroma block (the block size is 8x8) in a macroblock contains nonzero quantized coefficients. If cbp_chroma_nonzero_bit is 1, only the Cr chroma block in a macroblock contains nonzero quantized coefficients; otherwise, only the Cb chroma block in a macroblock contains nonzero quantized coefficients.

7.2.6 Block

trans_coefficient – it is used to specify run length and nonzero quantized coefficient. The parsing process of trans_coefficient is specified in subclause 9.3.

The size of current block can be either 16x16, 8x8, or 4x4. When the block size is 8x8, the scan order of 8x8 blocks within a macroblock is given in Figure 5. When the block size is 4x4, the scan order of 4x4 blocks within an 8x8 block is given in Figure 6. The scan order within one block refers to subclause 8.4.2.

8 Decoding process

8.1 General

Outputs of this process are decoded samples of the current frame.

This clause describes the decoding process, given syntax elements and upper-case variables from Clause 5.

The decoding process is specified such that all decoders shall produce numerically identical results. Any decoding process that produces identical results to the process described here conforms to the decoding process requirements of this document.

The various parameters in the bitstream for macroblock() and all syntactic structures above macroblock() are interpreted as indicated in Clause 7. Many of these parameters affect the decoding process described in the following subclauses. Once all of the macroblocks in a given frame have been processed, the entire frame will have been reconstructed.

An overview of the decoding process is given as follows.

— The intra prediction process for I macroblocks is specified in subclause 8.2, has intra prediction samples as its output.

— The inter prediction process for P and B macroblocks is specified in subclause 8.3 with inter prediction samples being the output.

— The transform coefficient decoding process and frame reconstruction process prior to deblocking filter process are specified in subclause 8.4. That process derives samples for I, P and B macroblocks. The output are reconstructed samples prior to the deblocking filter process.

— The reconstructed samples prior to the deblocking filter process that are next to the edges of blocks and macroblocks are processed by a deblocking filter as specified in subclause 8.5 with the output being the decoded samples.

The sequence of reconstructed frames shall be re-ordered for output by the decoder as described in subclause 6.10. The reconstructed frames shall be output from the decoding process at regular intervals of the frame period, which is the inverse of the frame rate determined by the frame_rate_code syntax element.

8.2 Intra prediction

8.2.1 General

This process is invoked for intra macroblocks.

Inputs to this process are reconstructed samples prior to the deblocking filter process from neighbouring macroblocks.

Outputs of this process are the Intra prediction samples of components of the macroblock.

Depending on the MbTransformType of current macroblock, the process of intra prediction for the luma component is specified as follows.

— If the MbTransformType is equal to 'Trans_16x16', the macroblock prediction mode is equal to 'Intra_16x16', and the specification in subclause 8.2.4 applies.

— Otherwise, the current macroblock is divided into 4 8x8 blocks, and these 8x8 blocks are processed in the scan order specified in Figure 5 as follows.

— If the SubMbTransformType of current 8x8 block is equal to 'Trans_4x4', the prediction mode of the current 8x8 block is equal to 'Intra_4x4', and the specification in subclause 8.2.2 applies.

Otherwise, the prediction mode of the current 8x8 block is equal to 'Intra_8x8', and the specification in subclause 8.2.3 applies.The process of intra prediction for the chroma components is described in subclause 8.2.5.

8.2.2 Intra_4x4 prediction process for luma samples

8.2.2.1 General

This process is invoked when the prediction mode of current 8x8 block is equal to 'Intra_4x4'.

Inputs to this process are reconstructed luma samples prior to the deblocking filter process from neighbouring 8x8 blocks.

Outputs of this process are 4x4 luma sample arrays as part of the 8x8 luma array of prediction samples of the block pred8x8_L.

The luma component of an 8x8 block consists of 4 blocks of 4x4 luma samples. These blocks are inverse scanned using the 4x4 luma block inverse scanning process as specified in subclause 6.12.5.

For all 4x4 luma blocks of the luma component of an 8x8 block with luma4x4BlkIdx = 0..3, the variable Intra4x4PredMode[luma4x4BlkIdx] is derived as specified in subclause 8.2.2.2.

For each luma block of 4x4 samples indexed using luma4x4BlkIdx = 0..3,

1) The Intra_4x4 sample prediction process in subclause 8.2.2.3 is invoked with luma4x4BlkIdx and reconstructed samples prior (in decoding order) to the deblocking filter process from adjacent luma blocks as the input and the output are the Intra_4x4 luma prediction samples $\text{pred4x4}_L[\ x, y\ ]$ with x, y = 0..3.

2) The position of the upper-left sample of a 4x4 luma block with index luma4x4BlkIdx inside the current 8x8 block is derived by invoking the inverse 4x4 luma block scanning process in subclause 6.12.5 with luma4x4BlkIdx as the input and the output being assigned to (xO, yO) and x, y = 0..3.

$$\text{pred8x8}_L[\ xO + x, yO + y\ ] = \text{pred4x4}_L[\ x, y\ ]$$

3) The transform coefficient decoding process and frame reconstruction process prior to deblocking filter process in subclause 8.4 is invoked with pred8x8$_L$ and luma4x4BlkIdx as the input and the reconstructed samples for the current 4x4 luma block S$_L$ as the output.

8.2.2.2 Derivation process for the Intra4x4PredMode

Inputs to this process are the index of the 4x4 luma block luma4x4BlkIdx.

Output of this process is the variable IntraLumaPredMode [luma4x4BlkIdx].

The value of intra_luma_pred_mode of 4x4 block with luma4x4BlkIdx is derived from bitstream parsing, and assigned to the variable IntraLumaPredMode[luma4x4BlkIdx]. Table 22 specifies the values for intra_luma_pred_mode and the associated names.

Table 22 — Luma intra prediction modes

intra_luma_pred_mode	Name
0	Intra_Vertical
1	Intra_Horizontal
2	Intra_DC
3	Intra_Down_Left
4	Intra_Down_Right

The intra_luma_pred_mode labelled 0, 1, 3, and 4 represent directions of predictions as illustrated in Figure 10.

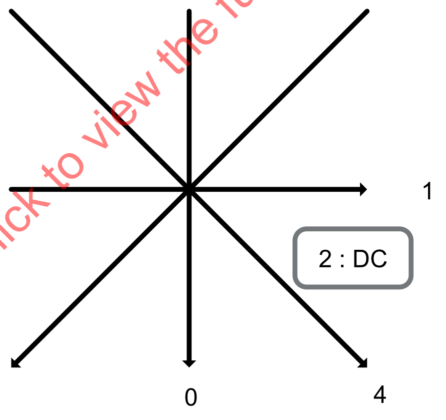


Figure 10 — Luma intra prediction mode directions

8.2.2.3 Intra_4x4 sample prediction

8.2.2.3.1 General

This process is invoked for each 4x4 luma block of a 8x8 block with prediction mode equal to 'Intra_4x4' followed by the transform decoding process and frame reconstruction process prior to deblocking for each 4x4 luma block.

Inputs to this process are the index of the 4x4 luma block with index luma4x4BlkIdx and reconstructed samples prior (in decoding order) to the deblocking filter process from adjacent luma blocks.

Output of this process are the prediction samples pred4x4$_L$[x, y], with x, y = 0..3 for the 4x4 luma block with index luma4x4BlkIdx.

The position of the upper-left sample of a 4x4 luma block with index luma4x4BlkIdx inside the current 8x8 block is derived by invoking the inverse 4x4 luma block scanning process in subclause 6.12.5 with luma4x4BlkIdx as the input and the output being assigned to (x0, y0).

8.2.2.3.2 Reference sample calculation

Let the decoded frame sample matrix of the current block be I. The reference samples for I is obtained by the following process: Let the coordinates of upper left corner sample of the current block be (x0, y0). The reference samples for current block are obtained by:

— If the samples with coordinates (x0+i−1, y0−1) (i=1..4) are "available", r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are "not available";

— If the samples with coordinates (x0+i−1, y0−1) (i=5..8) are "available", r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are equal to r[4], and availability of r[i] follows the availability of r[4];

— If the samples with coordinates (x0−1, y0+i−1) (i=1..4) are "available", c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are "not available";

— If the samples with coordinates (x0−1, y0+i−1) (i=5..8) are "available", c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are equal to c[4], and availability of c[i] follows the availability of c[4];

— If the sample with coordinate (x0−1, y0−1) is "available", r[0] is equal to I[x0−1, y0−1], and r[0] is "available"; otherwise:

  — If r[1] is "available" and c[1] is "not available", r[0] is equal to r[1], and r[0] is "available";

  — Otherwise, if c[1] is "available", and r[1] is "not available", r[0] is equal to c[1], and r[0] is "available";

  — Otherwise, r[0] is "not available";

— c[-1] is equal to r[0], and r[-1] is equal to r[0].

8.2.2.3.3 Specification of 4x4 Intra_Vertical prediction mode

This mode shall be used only when r[i] (i=1..4) is "available".

$\text{pred4x4}_L[x,y] = r[x + 1]$ (x,y=0..3)

8.2.2.3.4 Specification of 4x4 Intra_Horizontal prediction mode

This mode shall be used only when c[i] (i=1..4) is "available".

$\text{pred4x4}_L[x,y] = c[y + 1]$ (x,y=0..3)

8.2.2.3.5 Specification of 4x4 Intra_DC prediction mode

The intra prediction process of this mode is defined as follows.

— If both r[i] and c[i] (i=0..6) are "available",

$$\text{pred4x4}_L[x,y] = ((r[x - 1] + 4 * r[x] + 6 * r[x + 1] + 4 * r[x + 2] + r[x + 3] + 8) >> 4 + (c[y - 1] + 4 * c[y] + 6 * c[y + 1] + 4 * c[y + 2] + c[y + 3] + 8) >> 4) >> 1, (x,y=0..3)$$

— Otherwise, if only r[i] (i=0..6) is "available",

$$\text{pred4x4}_L[x,y] = (r[x - 1] + 4 * r[x] + 6 * r[x + 1] + 4 * r[x + 2] + r[x + 3] + 8) >> 4, (x,y=0..3)$$

— Otherwise, if only c[i] (i=0..6) is "available",

$pred4x4_L[x,y] = (c[y-1] + 4 * c[y] + 6 * c[y + 1] + 4 * c[y + 2] + c[y + 3] + 8) >> 4$, (x,y=0..3)

— Otherwise,

$pred4x4_L[x,y] = 128$ (x,y=0..3)

8.2.2.3.6 Specification of 4x4 Intra_Down_Left mode

This mode shall be used only when both r[i] and c[i] (i=2..8) are "available".

$pred4x4_L[x,y] = (r[x + y + 2] + c[x + y + 2]) >> 1$, (x,y=0..3)

8.2.2.3.7 Specification of 4x4 Intra_Down_Right mode

This mode shall be used only when both r[i] and c[i] (i=0..3) are "available".

— If x is equal to y,

$pred4x4_L[x,y] = r[0]$, (x,y=0..3)

— Otherwise, if x is greater than y,

$pred4x4_L[x,y] = r[x - y]$, (x,y=0..3)

— Otherwise,

$pred4x4_L[x,y] = c[y - x]$, (x,y=0..3)

8.2.3 Intra_8x8 prediction process for luma samples

8.2.3.1 General

This process is invoked when the prediction mode of the current block is equal to 'Intra_8x8'.

Inputs to this process are reconstructed luma samples prior to the deblocking filter process from neighbouring 8x8 blocks, and the index of the 8x8 luma block(given by the luma8x8BlkIdx).

Outputs of this process are 8x8 luma sample arrays as part of the 16x16 luma array of prediction samples of the macroblock $pred_L$.

Intra8x8PredMode[luma8x8BlkIdx] is derived as specified in subclause 8.2.3.2.

For current luma block of 8x8 samples indexed using luma8x8BlkIdx = 0..3,

1) The Intra_8x8 sample prediction process in subclause 8.2.3.3 is invoked with luma8x8BlkIdx and reconstructed samples prior (in decoding order) to the deblocking filter process from adjacent luma blocks as the input and the output are the Intra_8x8 luma prediction samples $pred8x8_L[x, y]$ with x, y = 0..7.

2) The transform coefficient decoding process and frame reconstruction process prior to deblocking filter process in subclause 8.4 is invoked with $pred_L$ and luma8x8BlkIdx as the input and the reconstructed samples for the current 8x8 luma block S_L as the output.

8.2.3.2 Derivation process for the Intra8x8PredMode

Inputs to this process are the index of the 8x8 luma block luma8x8BlkIdx.

Output of this process is the variable IntraLumaPredMode[luma8x8BlkIdx].

The value of intra_luma_pred_mode of 8x8 luma block with luma8x8BlkIdx is derived from bitstream parsing, and assigned to the variable IntraLumaPredMode [luma8x8BlkIdx].

Table 22 specifies the values for intra_luma_pred_mode and the associated names.

The intra_luma_pred_mode labelled 0, 1, 3, and 4 represent directions of predictions as illustrated in Figure 10.

8.2.3.3 Intra_8x8 sample prediction

8.2.3.3.1 General

This process is invoked for each 8x8 luma block of a macroblock with prediction mode equal to 'Intra_8x8' followed by the transform decoding process and frame reconstruction process prior to deblocking for each 8x8 luma block.

Inputs to this process are the index of the 8x8 luma block with index luma8x8BlkIdx and reconstructed samples prior (in decoding order) to the deblocking filter process from adjacent luma blocks.

Output of this process are the prediction samples $pred8x8_L[x, y]$, with x, y = 0..7 for the 8x8 luma block with index luma8x8BlkIdx.

The position of the upper-left sample of a 8x8 luma block with index luma8x8BlkIdx inside the current macroblock is derived by invoking the inverse 8x8 luma block scanning process in subclause 6.12.4 with luma8x8BlkIdx as the input and the output being assigned to (xO, yO).

8.2.3.3.2 Reference sample calculation

Let the decoded frame sample matrix of the current block be I;

The reference samples for I are obtained by the following process: Let the coordinates of upper left corner sample of the current block be (x0, y0). The reference samples for current block are obtained by:

— If the samples with coordinates (x0+i−1, y0−1) (i=1..8) are "available", r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are "not available";

— If the samples with coordinates (x0+i−1, y0−1) (i=9..16) are "available", r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are equal to r[8], and availability of r[i] follows the availability of r[8];

— If the samples with coordinates (x0−1, y0+i−1) (i=1..8) are "available", c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are "not available";

— If the samples with coordinates (x0−1, y0+i−1) (i=9..16) are "available", c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are equal to c[8], and availability of c[i] follows the availability of c[8];

— If the sample with coordinate (x0−1, y0−1) is "available", r[0] is equal to I[x0−1, y0−1], and r[0] is "available"; otherwise:

  — If r[1] is "available" and c[1] is "not available", r[0] is equal to r[1], and r[0] is "available";

  — Otherwise, if c[1] is "available", and r[1] is "not available", r[0] is equal to c[1], and r[0] is "available";

  — Otherwise, r[0] is "not available";

— c[-1] is equal to r[0], and r[-1] is equal to r[0].

8.2.3.3.3 Specification of 8x8 Intra_Vertical prediction mode

This mode shall be used only when r[i] (i=1..8) is "available".

$\text{pred8x8}_L[x,y] = r[x + 1]$ (x,y=0..7)

8.2.3.3.4 Specification of 8x8 Intra_Horizontal prediction mode

This mode shall be used only when c[i] (i=1..8) is "available".

$\text{pred8x8}_L[x,y] = c[y + 1]$ (x,y=0..7)

8.2.3.3.5 Specification of 8x8 Intra_DC prediction mode

The intra prediction process of this mode is defined as follows.

— If both r[i] and c[i] (i=0..10) are "available",

$\text{pred8x8}_L[x,y] = ((r[x - 1] + 4 * r[x] + 6 * r[x + 1] + 4 * r[x + 2] + r[x + 3] + 8) >> 4 +$

$(c[y - 1] + 4 * c[y] + 6 * c[y + 1] + 4 * c[y + 2] + c[y + 3] + 8) >> 4) >> 1$,(x,y=0..7)

— Otherwise, if only r[i] (i=0..10) is "available",

$\text{pred8x8}_L[x,y] = (r[x - 1] + 4 * r[x] + 6 * r[x + 1] + 4 * r[x + 2] + r[x + 3] + 8) >> 4$, (x,y=0..7)

— Otherwise, if only c[i] (i=0..10) is "available",

$\text{pred8x8}_L[x,y] = (c[y-1] + 4 * c[y] + 6 * c[y + 1] + 4 * c[y + 2] + c[y + 3] + 8) >> 4$, (x,y=0..7)

— Otherwise,

$\text{pred8x8}_L[x,y] = 128$ (x,y=0..7)

8.2.3.3.6 Specification of 8x8 Intra_Down_Left mode

This mode shall be used only when both r[i] and c[i] (i=2..16) are "available".

$\text{pred8x8}_L[x,y] = (r[x + y + 2] + c[x + y + 2]) >> 1$, (x,y=0..7)

8.2.3.3.7 Specification of 8x8 Intra_Down_Right mode

This mode shall be used only when both r[i] and c[i] (i=0..7) are "available".

— If x is equal to y, then

$\text{pred8x8}_L[x,y] = r[0]$, (x,y=0..7)

— Otherwise, if x is greater than y, then

$\text{pred8x8}_L[x,y] = r[x - y]$, (x,y=0..7)

— Otherwise,

$\text{pred8x8}_L[x,y] = c[y - x]$, (x,y=0..7)

8.2.4 Intra_16x16 prediction process for luma samples

8.2.4.1 General

This process is invoked when the macroblock prediction mode is equal to 'Intra_16x16'. It specifies how the Intra prediction luma samples for the current macroblock are derived.

Input to this process are reconstructed samples prior to the deblocking process from neighbouring luma blocks (if available).

Outputs of this process are Intra prediction luma samples for the current macroblock $pred_L[x, y]$.

The value of intra_luma_pred_mode of current macroblock is derived from bitstream parsing, and assigned to the variable Intra16x16PredMode.

Table 22 specifies the values for intra_luma_pred_mode and the associated names.

The intra_luma_pred_mode labelled 0, 1, 3, and 4 represent directions of predictions as illustrated in Figure 10.

Let $pred_L[x, y]$ with x, y = 0..15 denote the prediction samples for the 16x16 luma block samples.

8.2.4.2 Reference sample calculation

8.2.4.2.1 General

Let the decoded frame sample matrix of the current block be I;

The reference samples for I is obtained by the following process: Let the coordinates of upper left corner sample of the current block be (x0, y0). The reference samples for current block are obtained by:

— If the samples with coordinates (x0+i−1, y0−1) (i=1..16) are "available", then r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are "not available";

— If the samples with coordinates (x0+i−1, y0−1) (i=17..32) are "available", then r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are equal to r[16], and availability of r[i] follows the availability of r[16];

— If the samples with coordinates (x0−1, y0+i−1) (i=1..16) are "available", then c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are "not available";

— If the samples with coordinates (x0−1, y0+i−1) (i=17..32) are "available", then c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are equal to c[16], and availability of c[i] follows the availability of c[16];

— If the sample with coordinate (x0−1, y0−1) is "available", then r[0] is equal to I[x0−1, y0−1], and r[0] is "available"; otherwise:

  — If r[1] is "available" and c[1] is "not available", then r[0] is equal to r[1], and r[0] is "available";

  — Otherwise, if c[1] is "available", and r[1] is "not available", then r[0] is equal to c[1], and r[0] is "available";

  — Otherwise, r[0] is "not available";

— c[-1] is equal to r[0], and r[-1] is equal to r[0].

8.2.4.2.2 Specification of 16x16 Intra_Vertical prediction mode

This mode shall be used only when r[i] (i=1..16) is "available".

$pred_L[x,y] = r[x + 1]$ (x,y=0..15)

8.2.4.2.3 Specification of 16x16 Intra_Horizontal prediction mode

This mode shall be used only when c[i] (i=1..16) is "available".

$pred_L[x,y] = c[y + 1]$ (x,y=0..15)

8.2.4.2.4 Specification of 16x16 Intra_DC prediction mode

The intra prediction process of this mode is defined as follows.

— If both r[i] and c[i] (i=0..18) are “available”,

$\mathrm{pred}_L[x,y] = ((r[x-1] + 4 * r[x] + 6 * r[x+1] + 4 * r[x+2] + r[x+3] + 8) >> 4 + (c[y-1] + 4 * c[y] + 6 * c[y+1] + 4 * c[y+2] + c[y+3] + 8) >> 4) >> 1, (x,y=0..15)$

— Otherwise, if only r[i] (i=0..18) is “available”,

$\mathrm{pred}_L[x,y] = (r[x-1] + 4 * r[x] + 6 * r[x+1] + 4 * r[x+2] + r[x+3] + 8) >> 4, (x,y=0..15)$

— Otherwise, if only c[i] (i=0..18) is “available”,

$\mathrm{pred}_L[x,y] = (c[y-1] + 4 * c[y] + 6 * c[y+1] + 4 * c[y+2] + c[y+3] + 8) >> 4, (x,y=0..15)$

— Otherwise,

$\mathrm{pred}_L[x,y] = 128\ (x,y=0..15)$

8.2.4.2.5 Specification of 16x16 Intra_Down_Left mode

This mode shall be used only when both r[i] and c[i] (i=2..32) are “available”.

$\mathrm{pred}_L[x,y] = (r[x+y+2] + c[x+y+2]) >> 1, (x,y=0..15)$

8.2.4.2.6 Specification of 16x16 Intra_Down_Right mode

This mode shall be used only when both r[i] and c[i] (i=0..15) are “available”.

— If x is equal to y, then

$\mathrm{pred}_L[x,y] = r[0], (x,y=0..15)$

— Otherwise, if x is greater than y, then

$\mathrm{pred}_L[x,y] = r[x-y], (x,y=0..15)$

— Otherwise,

$\mathrm{pred}_L[x,y] = c[y-x], (x,y=0..15)$

8.2.5 Intra prediction for 8x8 chroma block

8.2.5.1 General

This process is invoked for intra macroblock. It specifies how the Intra prediction chroma samples for the current macroblock are derived.

Inputs to this process are reconstructed samples prior to the deblocking process from neighbouring chroma blocks (if available).

Outputs of this process are Intra prediction chroma samples for the current macroblock $\mathrm{pred}_{Cb}[x, y]$ and $\mathrm{pred}_{Cr}[x, y]$.

Both chroma blocks (Cb and Cr) of the macroblock share the same prediction mode. The prediction mode is applied to each of the chroma blocks separately. The process specified in this subclause is invoked for each chroma block. In the remainder of this subclause, chroma block refers to one of the two chroma blocks and the subscript C is used as a replacement of the subscript Cb or Cr.

Let $\mathrm{pred}_C[x, y]$ with x, y = 0..7 denote the prediction samples for the chroma block samples.

Intra chroma prediction mode of current 8x8 chroma block is parsed from the intra_chroma_pred_mode, which is specified in Table 23.

Table 23 — 8x8 Chroma intra prediction mode

intra_chroma_pred_mode	Name
0	Intra_Chroma_DC
1	Intra_Chroma_Horizontal
2	Intra_Chroma_Vertical
3	Intra_Chroma_Plane

8.2.5.2 Reference sample calculation

Let the decoded frame sample matrix of current chroma block be I. The reference samples for I is obtained by the following process: Let the coordinates of upper left corner sample of the current block be (x0, y0). The reference samples for current block are obtained by:

— If the samples with coordinates (x0+i−1, y0−1) (i=1..8) are "available", r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are "not available";

— If the samples with coordinates (x0+i−1, y0−1) (i=9..16) are "available", r[i] are equal to I[x0+i−1, y0−1], and r[i] are "available"; otherwise, r[i] are equal to r[8], and availability of r[i] follows the availability of r[8];

— If the samples with coordinates (x0−1, y0+i−1) (i=1..8) are "available", c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are "not available";

— If the samples with coordinates (x0−1, y0+i−1) (i=9..16) are "available", c[i] are equal to I[x0−1, y0+i−1], and c[i] are "available"; otherwise, c[i] are equal to c[8], and availability of c[i] follows the availability of c[8];

— If the sample with coordinate (x0−1, y0−1) is "available", both r[0] and c[0] are equal to I[x0−1, y0−1], and both r[0] and c[0] are "available"; otherwise:

  — If r[1] is "available" and c[1] is "not available", both r[0] and c[0] are equal to r[1], and both r[0] and c[0] are "available";

  — Otherwise, if c[1] is "available", and r[1] is "not available", both r[0] and c[0] are equal to c[1], and both r[0] and c[0] are "available";

  — Otherwise, both r[0] and c[0] are "not available".

8.2.5.3 Specification of Intra_Chroma_DC prediction mode

The values of the prediction samples $pred_C[x, y]$ with x = 0..7 and y = 0..7 are derived as follows.

— If both r[i] and c[i] (i=0..9) are "available",

$$pred_C[x,y] = ((r[x] + 2 * r[x + 1] + r[x + 2] + 2) >> 2 + (c[y] + 2 * c[y + 1] + c[y + 2] + 2) >> 2) >> 1, \quad (x,y=0..7)$$

— Otherwise, if r[i] (i=0..9) is "available", then

$$pred_C[x,y] = r[x + 1], (x,y=0..7)$$

— Otherwise, if c[i] (i=0..9) is "available", then

$$pred_C[x,y] = c[y + 1], (x,y=0..7)$$

— Otherwise,

$pred_C[x,y] = 128$, (x, y=0..7)

8.2.5.4 Specification of Intra_Chroma_Horizontal prediction mode

This mode shall be used only when c[i] (i=1..8) is "available".

$pred_C[x,y] = c[y + 1]$, (x, y=0..7)

8.2.5.5 Specficication of Intra_Chroma_Vertical prediction mode

This mode shall be used only when r[i] (i=1..8) is "available".

$pred_C[x,y] = r[x + 1]$, (x, y=0..7)

8.2.5.6 Specification of Intra_Chroma_Plane prediction mode

This mode shall be used only when both r[i] and c[i] (i=1..8) are "available".

Let,

$$\text{ih} = \sum_{i=0}^{3} (i+1) * (r[5+i] - r[3-i])$$

$$\text{iv} = \sum_{i=0}^{3} (i+1) * (c[5+i] - c[3-i])$$

ia = (r[8] + c[8]) << 4

ib = (17 * ih + 16) >> 5

ic = (17 * iv + 16) >> 5

Then,

$pred_C[x,y] = clip1((ia + (x - 3) * ib + (y - 3) * ic + 16) >> 5)$, (x, y = 0..7)

8.3 Inter prediction

8.3.1 General

This process is invoked when decoding inter macroblocks in P and B frames.

Outputs of this process are inter prediction samples for the current macroblock that are a 16x16 array $pred_L$ of luma samples and two 8x8 arrays $pred_{Cb}$ and $pred_{Cr}$ of chroma samples, one for each of the chroma components Cb and Cr.

The partitioning of a macroblock is specified by mb_part_type. Each macroblock partition is referred by mbPartIdx.

The following steps are specified for each macroblock partition.

The functions MbPartWidth(), MbPartHeight() describing the width and height of macroblock partitions are specified in Table 16, and Table 17.

The variables partWidth and partHeight are derived as follows.

— partWidth = MbPartWidth(mb_part_type);

— partHeight = MbPartHeight(mb_part_type);

with mbPartIdx proceeding over values 0..3.

The inter prediction process for a macroblock partition with mbPartIdx consists of the following ordered steps.

— Derivation process for motion vector components and reference indices as specified in subclause 8.3.2.

— Decoding process for inter prediction samples as specified in subclause 8.3.3.

For use in derivation processes of variables invoked later in the decoding process, the following assignments are made:

MvFst[mbPartIdx] = mvFst;

MvSnd[mbPartIdx] = mvSnd;

RefIdxFst[mbPartIdx] = refIdxFst;

RefIdxSnd[mbPartIdx] = refIdxSnd;

PredFlagFst[mbPartIdx] = predFlagFst;

PredFlagSnd[mbPartIdx] = predFlagSnd.

The location of the upper-left sample of the partition relative to the upper-left sample of the macroblock is derived by invoking the inverse macroblock partition scanning process as described in subclause 6.12.3 with mbPartIdx as the input and (xP, yP) as the output.

The macroblock prediction is formed by placing the partition prediction samples in their correct relative positions in the macroblock, as follows.

The variable predL[xP + x, yP + y] with x = 0 .. partWidth − 1, y = 0 .. partHeight − 1 is derived by

$\text{pred}_L[\ xP + x, yP + y\ ] = \text{predPart}_L[\ x, y\ ]$

The variable predC[xP / 2 + x, yP / 2 + y] with x = 0 .. partWidth / 2 − 1, y = 0 .. partHeight / 2 − 1, and C being replaced by Cb or Cr, is derived by

$\text{pred}_C[\ xP / 2 + x, yP / 2 + y\ ] = \text{predPart}_C[\ x, y\ ]$

8.3.2 Derivation process for motion vector components and reference indices

8.3.2.1 General

Input to this process is

— a macroblock partition mbPartIdx.

Outputs of this process are:

— 32-bit signed integer luma motion vectors mvFst and mvSnd as well as 32-bit signed integer chroma motion vectors mvCFst and mvCSnd;

— reference indices refIdxFst and refIdxSnd;

— prediction list utilization flags predFlagFst and predFlagSnd.

For the derivation of the variables mvFst and mvSnd as well as refIdxFst and refIdxSnd, the following applies.

— If MbPartType is equal to P_16x16 and MbPredType(mbPartIdx) is equal to ‘Pred_Skip’, the derivation process for luma motion vectors for skipped macroblocks in P frames in subclause 8.3.2.2 is invoked with the output being the luma motion vectors mvFst and reference indices refIdxFst, and predFlagFst is set equal to 1. mvSnd and refIdxSnd are marked as unavailable, and predFlagSnd is set equal to 0.

— Otherwise, if MbPartType is equal to ‘B_Skip’, or MbPartType is equal to B_8x8 and MbPredType(mbPartIdx) is equal to ‘Pred_Skip’, the derivation process for luma motion vectors for B_Skip in B frames in subclause 8.3.2.3 is invoked with mbPartIdx as the input and the output being the luma motion vectors mvFst, mvSnd, the reference indices refIdxFst, refIdxSnd, and the prediction utilization flags predFlagFst, predFlagSnd.

— Otherwise, if MbPredType(mbPartIdx) is equal to ‘Pred_Sym’, the derivation process for luma motion vectors for B_Sym in B frames in subclause 8.3.2.4 is invoked with mbPartIdx as the input and the output being the luma motion vectors mvFst, mvSnd, the reference indices refIdxFst, refIdxSnd, and the prediction utilization flags predFlagFst, predFlagSnd.

— Otherwise, if MbPredType(mbPartIdx) is equal to ‘Pred_Mh’, the derivation process for luma motion vectors for P_Mh in P frames in subclause 8.3.2.5 is invoked with mbPartIdx as the input and the output being the luma motion vectors mvFst, mvSnd, the reference indices refIdxFst, refIdxSnd, and the prediction utilization flags predFlagFst, predFlagSnd.

— Otherwise, for X being replaced by either ‘Fst’ or ‘Snd’ in the variables predFlagX, mvX, refIdxX, and in Pred_X and in the syntax elements ref_idx_X and mvd_X, and the following applies.

  The variables refIdxX and predFlagX are derived as follows.

  — If MbPredType (mbPartIdx) is equal to ‘Pred_Fwd’,

    refIdxFst = reference_frame_index

    predFlagFst = 1

  — Otherwise,

    refIdxFst = −1

    predFlagFst = 0

  — If MbPredType (mbPartIdx) is equal to ‘Pred_Bck’,

    refIdxSnd = 0

    predFlagSnd = 1

  — Otherwise,

    refIdxSnd = −1

    predFlagSnd = 0

When predFlagX is 1, the derivation process for luma motion vector prediction in subclause 8.3.2.6 is invoked with mbPartIdx, and list suffix X as the input and the output being mvpX. The luma motion vectors are derived by:

mvX[0] = mvpX[0] + mv_diff_x;

mvX[1] = mvpX[1] + mv_diff_y.

For the derivation of the variables for the chroma motion vectors, the following applies. When predFlagX is equal to 1, the derivation process for chroma motion vectors in subclause 8.3.2.8 is invoked with mvX as input and the output being mvCX.

8.3.2.2 Derivation process for luma motion vectors for skipped macroblock in P frame

This process is invoked when MbPartType is equal to P_16x16 and MbPredType is equal to 'Pred_Skip'.

Outputs of this process are the motion vector mvFst and the reference index refIdxFst.

The reference index refIdxFst for a skipped macroblock is derived as follows.

refIdxFst = 0

For the derivation of the motion vector mvFst, the following applies.

The process specified in subclause 8.3.2.7 is invoked with mbPartIdx set equal to 0, and list suffix Fst as input and the output is assigned to mbAddrA, mbAddrB, mvFstA, mvFstB, refIdxFstA, and refIdxFstB.

— If mbAddrA or mbAddrB is marked as "not available", mvFst is a zero vector.

— Otherwise, if mvFstA is a zero vector and refIdxFstA is 0, or if mvFst B is a zero vector and refIdxFstB is 0, then mvFst is a zero vector.

— Otherwise, the derivation process for luma motion vector prediction as specified in subclause 8.3.2.6 is invoked with mbPartIdx = 0 and list suffix Fst as input, and the output is assigned to mvFst.

8.3.2.3 Derivation process for luma motion vectors for B_Skip

This process is invoked when MbPartType is equal to 'B_skip', or MbPartType is equal to B_8x8 and MbPredType(mbPartIdx) is equal to 'Pred_Skip'.

Inputs to this process is mbPartIdx.

Outputs of this process are the reference indices refIdxFst, refIdxSnd, the motion vectors mvFst and mvSnd, and the prediction list utilization flags, predFlagFst and predFlagSnd.

Forward and backward reference frames of the current block are the default reference frames, i.e. reference frames with reference indices 0.

refIdxFst = 0

refIdxSnd = 0

Both forward and backward prediction lists are used.

predFlagFst = 1

predFlagSnd = 1

— If the mb_part_type of the collocated macroblock of current macroblock in backward reference frame is 'I_Block', the forward and backward motion vectors (given by mvFst and mvSnd) of the current block are the predicted forward and backward motion vectors of the macroblock containing current block.

  The predicted forward and backward motion vectors are obtained according to motion vector prediction method as specified in subclause 8.3.2.6. mvFst is derived with mbPartIdx and list suffix Fst as input, and mvSnd is derived with mbPartIdx and list suffix Snd as input.

— Otherwise,

  The frame_distance of the forward reference frame of current macroblock partition is assigned to DistanceIndexFw, and the frame_distance of backward reference frame of current macroblock partition is assigned to DistanceIndexBw as shown in Figure 11. The frame_distance of current frame is assigned to DistanceIndexCur. The reference index of the collocated macroblock partition in backward referene picture is assigned to refidxCol.

BlockDistanceFw = (DistanceIndexCur − DistanceIndexFw + 256) % 256

BlockDistanceBw = (DistanceIndexBw − DistanceIndexCur + 256) % 256

The motion vector of the collocated macroblock partition in backward reference frame is mvRef. Let i be a variable being set equal to 0 and 1, respectively.

If (BlockDistanceFw + BlockDistanceBw) is less than 5,

mvFst[i] = (mvRef[i] * MvWeightNum / MvWeightDen) / pfactor

mvSnd[i] = −(((mvRef[i] * (MvWeightDen − MvWeightNum)) / MvWeightDen) / pfactor)

where MvWeightNum and MvWeightDen are specified in Table 24, and

pfactor = (refidxCol==0) ? 1 : 2

Otherwise,

mvFst[i] = mvRef[i]

mvSnd[i] = −mvRef[i]

Table 24 — Look-up table for temporal MV prediction

BlockDistanceFw	BlockDistanceBw	MvWeightNum	MvWeightDen
1	1	1	2
1	2	1	3
2	1	2	3
1	3	1	4
2	2	1	2
3	1	3	4

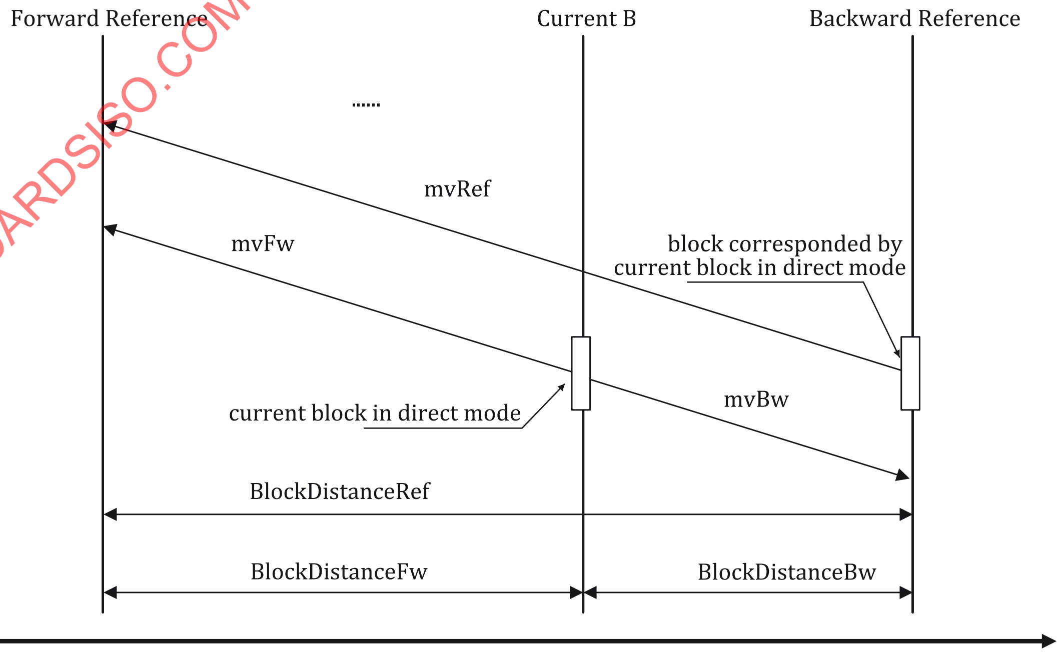


Figure 11 — Derivation process of motion vectors in B skip mode

8.3.2.4 Derivation process for luma motion vectors for B_Sym

This process is invoked when MbPredType is equal to 'Pred_Sym'.

Inputs to this process is mbPartIdx.

Outputs of this process are the reference indices refIdxFst and refIdxSnd, the motion vectors mvFst and mvSnd, and the prediction list utilization flags predFlagFst and predFlagSnd.

Reference frame indexes are derived as follows.

refIdxFst = 0

refIdxSnd = 0

Both forward and backward prediction lists are used.

predFlagFst = 1

predFlagSnd = 1

The forward motion vector of block in symmetrical mode mvFst is obtained as follows.

The derivation process for luma motion vector prediction in subclause 8.3.2.6 is invoked with mbPartIdx and list suffix Fst as input, and the output being mvpFst. The mvFst is derived by:

mvFst[0] = mvpFst[0] + mv_diff_x;

mvFst[1] = mvpFst[1] + mv_diff_y.

The backward motion vector mvSnd is derived based on mvFst as shown in Figure 12 by:

— If (BlockDistanceFw + BlockDistanceBw) is less than 5,

mvSnd[0] = −(mvFst[0] * (MvWeightDen − MvWeightNum) / MvWeightNum)

mvSnd[1] = −(mvFst[1]) * (MvWeightDen − MvWeightNum) / MvWeightNum)

where MvWeightNum and MvWeightDen are specified in Table 24, and BlockDistanceFw and BlockDistanceBw are defined in 8.3.2.3.

— Otherwise,

mvSnd[0] = −mvFst[0]

mvSnd[1] = −mvFst[1]

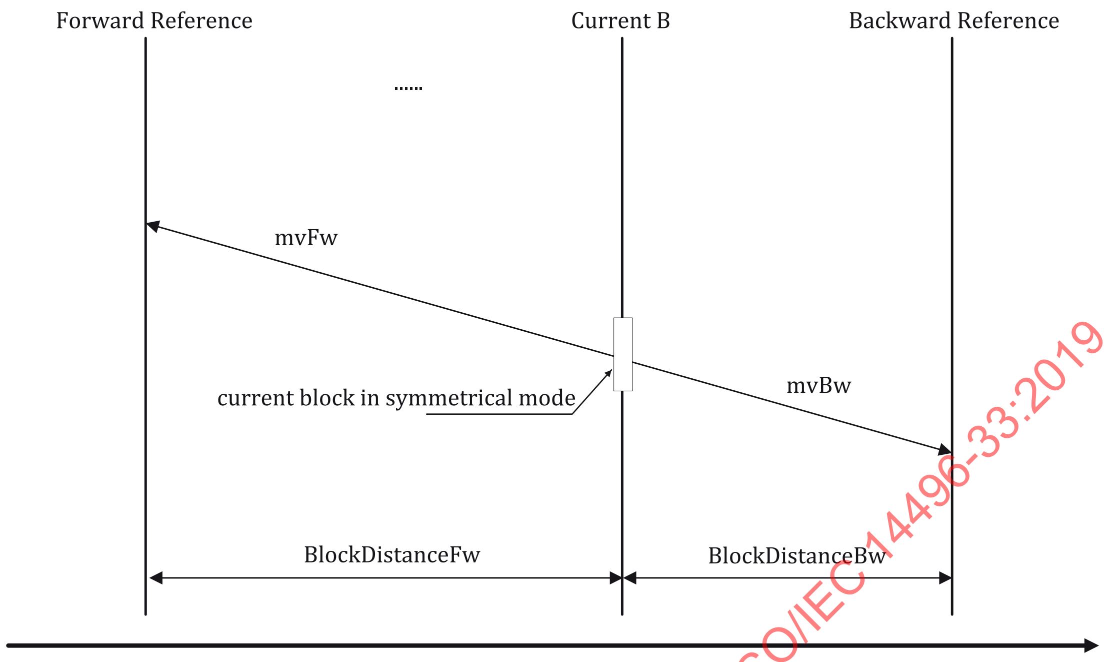


Figure 12 — Symmetrical mode

8.3.2.5 Derivation process for luma motion vectors for P_Mh

This process is invoked when MbPredType is equal to 'Pred_Mh'.

Inputs to this process is mbPartIdx.

Outputs of this process are the reference indices refIdxFst and refIdxSnd, the motion vectors mvFst and mvSnd, and the prediction list utilization flags predFlagFst and predFlagSnd.

refIdxFst is determined by the syntax element of reference_frame_index, and

refIdxSnd = refIdxFst

Both Fst and Snd lists are used.

predFlagFst = 1

predFlagSnd = 1

The derivation process for luma motion vector prediction in subclause 8.3.2.6 is invoked with mbPartIdx and suffix Fst, and the output being mvpFst. The mvFst is derived by:

mvFst[0] = mvpFst[0] + mv_diff_x;

mvFst[1] = mvpFst[1] + mv_diff_y.

The motion vector mvSnd is set equal to mvpFst:

mvSnd[0] = mvpFst[0];

mvSnd[1] = mvpFst[1].

8.3.2.6 Derivation process for luma motion vector prediction

Inputs to this process is the macroblock partition index mbPartIdx and list suffix X.

Output of this process is the prediction mvpX of the motion vector mvX.

The derivation process for the neighbouring blocks for motion data in subclause 8.3.2.7 is invoked with mbPartIdx and list suffix X as the input and with mbAddrN\mbPartIdxN, reference indices refIdxXN and the motion vectors mvXN with N being replaced by A, B, or C as the output.

The following rules are applied in sequential order to determine the motion vector predictor mvpX.

— If only one of refIdxXA, refIdxXB, and refIdxXC is available, and the motion vector of that block with available reference frame is mvXN,

  mvpX[0] = mvXN[0]

  mvpX[1] = mvXN[1]

— Otherwise,

  To derive mvpX[0]:

  — If mvXA[0] < 0 and mvXB[0] > 0 and mvXC[0] > 0, or if mvXA[0] > 0 and mvXB[0] < 0 and mvC[0] < 0,

    mvpX[0] = (mvXB[0] + mvXC[0]) / 2

  — Otherwise, if mvXB[0] < 0 and mvXA[0] > 0 and mvXC[0] > 0, or if mvXB[0] > 0 and mvXA[0] < 0 and mvXC[0] < 0,

    mvpX[0] = (mvXA[0] + mvXC[0]) / 2

  — Otherwise, if mvXC[0] < 0 and mvXA[0] > 0 and mvXB[0] > 0, or if mvXC[0] > 0 and mvXA[0] < 0 and mvXB[0] < 0,

    mvpX[0] = (mvXA[0] + mvXB[0]) / 2

  — Otherwise, calculate the distance between every two candidates, namely ABSVAB[0], ABSVBC[0] and ABSVCA[0], where,

    ABSVAB[0] = | mvXA[0] − mvXB[0] |

    ABSVBC[0] = | mvXB[0] − mvXC[0] |

    ABSVCA[0] = | mvXC[0] − mvXA[0] |

    — If ABSVAB[0] < ABSVBC[0] and ABSVAB[0] < ABSVCA[0], then,

      mvpX[0] = (mvXA[0] + mvXB[0]) / 2

    — Otherwise, if ABSVBC[0] < ABSVAB[0] and ABSVBC[0] < ABSVCA[0], then,

      mvpX[0] = (mvXB[0] + mvXC[0]) / 2

    — Otherwise,

      mvpX[0] = (mvXA[0] + mvXC[0]) / 2

To derive mvpX[1]:

— If mvXA[1] < 0 and mvXB[1] > 0 and mvXC[1] > 0, or if mvXA[1] > 0 and mvXB[1] < 0 and mvC[1] < 0,

  mvpX[1] = (mvXB[1] + mvXC[1]) / 2

— Otherwise, if mvXB[1] < 0 and mvXA[1] > 0 and mvXC[1] > 0, or if mvXB[1] > 0 and mvXA[1] < 0 and mvXC[1] < 0,

  mvpX[1] = (mvXA[1] + mvXC[1]) / 2

— Otherwise, if mvXC[1] < 0 and mvXA[1] > 0 and mvXB[1] > 0, or if mvXC[1] > 0 and mvXA[1] < 0 and mvXB[1] < 0,

  mvpX[1] = (mvXA[1] + mvXB[1]) / 2

— Otherwise, calculate the distance between every two candidates, namely ABSVAB[1], ABSVBC[1] and ABSVCA[1], where,

  ABSVAB[1] = | mvXA[1] − mvXB[1] |

  ABSVBC[1] = | mvXB[1] − mvXC[1] |

  ABSVCA[1] = | mvXC[1] − mvXA[1] |

  — If ABSVAB[1] < ABSVBC[1] and ABSVAB[1] < ABSVCA[1],

    mvpX[1] = (mvXA[1] + mvXB[1]) / 2

  — Otherwise, if ABSVBC[1] < ABSVAB[1] and ABSVBC[1] < ABSVCA[1],

    mvpX[1] = (mvXB[1] + mvXC[1]) / 2

  — Otherwise,

    mvpX[1] = (mvXA[1] + mvXC[1]) / 2

8.3.2.7 Derivation process for luma motion vectors

Inputs to this process are:

— the macroblock partition index mbPartIdx;

— the list suffix X.

Outputs of this process are (with N being replaced by A, B, or C)

— mbAddrN\mbPartIdxN specifying neighbouring partitions,

— the motion vectors mvXN of the neighbouring partitions, and

— the reference indices refIdxXN of the neighbouring partitions.

The partitions mbAddrN\mbPartIdxN with N being either A, B, or C are derived in the following ordered steps.

1) Let mbAddrD\mbPartIdxD be variables specifying an additional neighbouring partition.

2) The process in subclause 6.12.8.4 is invoked with mbPartIdx as the input and the output is mbAddrN\mbPartIdxN with N being replaced by A, B, C, or D.

3) When the partition mbAddrC\mbPartIdxC is not available, the following applies

   mbAddrC = mbAddrD

   mbPartIdxC = mbPartIdxD

The motion vectors mvXN and reference indices refIdxXN (with N being A, B, or C) are derived as follows.

— If the macroblock partition mbAddrN\mbPartIdxN is not available or mbAddrN is coded in intra prediction mode or predFlagX of mbAddrN\mbPartIdxN is equal to 0, both components of mvXN are set equal to 0 and refIdxXN is set equal to −1.

— Otherwise, the following applies.

The motion vector mvXN and reference index refIdxXN are set equal to MvX[mbPartIdxN] and RefIdxX[mbPartIdxN], respectively, which are the motion vector mvX and reference index refIdxX that have been assigned to the macroblock partition mbAddrN\mbPartIdxN.

8.3.2.8 Derivation process for chroma motion vectors

Inputs to this process are a luma motion vector mvX.

Outputs of this process are a chroma motion vector mvCX.

A chroma motion vector is derived from the corresponding luma motion vector. Since the accuracy of luma motion vectors is one-quarter sample and chroma has half resolution compared to luma, the accuracy of chroma motion vectors is one-eighth sample, i.e., a value of 1 for the chroma motion vector refers to a one-eighth sample displacement.

The horizontal and vertical components of the chroma motion vector mvCX are derived by dividing the corresponding components of luma motion vector mvX by 2,

mvCX[0] = mvX[0] / 2

mvCX[1] = mvX[1] / 2

8.3.3 Decoding process for inter prediction samples

8.3.3.1 General

Inputs to this process are:

— a macroblock partition mbPartIdx;

— variables specifying partition width and height, partWidth and partHeight;

— luma motion vectors mvFst and mvSnd and chroma motion vectors mvCFst and mvCSnd;

— reference indices refIdxFst and refIdxSnd;

— prediction list utilization flags, predFlagFst and predFlagSnd.

Outputs of this process are:

— the inter prediction samples predPart, which are a (partWidth)x(partHeight) array predPart_L of prediction luma samples, and two (partWidth/2)x(partHeight/2) arrays predPart_{Cb}, predPart_{Cr} of prediction chroma samples, one for each of the chroma components Cb and Cr.

Let predPartFst_L and predPartSnd_L be (partWidth)x(partHeight) arrays of predicted luma sample values and predPartFst_{Cb}, predPartSnd_{Cb}, predPartFst_{Cr}, and predPartSnd_{Cr} be (partWidth/2)x(partHeight/2) arrays of predicted chroma sample values.

For X being replaced by either ‘Fst’ or ‘Snd’ in the variables predFlagX, RefPicListX, refIdxX, refPicX, predPart_X, the following is specified.

When predFlagX is equal to 1, the following applies.

— The reference frame consisting of an ordered two-dimensional array refPicXL of luma samples and two ordered two-dimensional arrays refPicXCb and refPicXCr of chroma samples is derived by invoking the process specified in subclause 8.3.3.2 with refIdxX and RefPicListX given as input.

— The arrays predPartX_L, predPartX_{Cb}, and predPartX_{Cr} are derived by invoking the process specified in subclause 8.3.3.3 with the current partition specified by mbPartIdx, the motion vectors mvX, mvCX, and the reference arrays with refPicXL, refPicXCb, and refPicXCr given as input.

For C being replaced by L, Cb, or Cr, the array $predPart_C$ of the prediction samples of component C is derived by invoking the process specified in subclause 8.3.3.4 with the current partition specified by mbPartIdx and the array predPartFstC and predPartSndC as well as predFlagFst and predFlagSnd given as input.

8.3.3.2 Reference frame selection process

Input to this process is a reference index refIdxX.

Output of this process is a reference frame consisting of a two-dimensional array of luma samples refPicXL and two two-dimensional arrays of chroma samples refPicXCb and refPicXCr.

Reference frame list RefPicListX is a list of previously decoded reference frames.

The reference frame list RefPicListX is derived as specified in subclause 8.6.

The refIdx is mapped to another variable refIdx_2 by the following process:

```
If refIdx < 2
    refIdx_2 = refIdx
else
    refidx_2 = 4 * refIdx – 5
```

The output is the reference frame referred to by RefPicList [refIdx_2].

The output reference frame consists of a (PicWidth) × (PicHeight) array of luma samples refPicXL and two (PicWidth/2) × (PicHeight/2) arrays of chroma samples refPicXCb and refPicCr.

The reference frame sample arrays refPicXL, refPicXCb, refPicXCr correspond to decoded sample arrays S'_L,S'_{Cb}, S'_{Cr} derived in subclause 8.5 for previous decoded frames.

8.3.3.3 Fractional sample interpolation process

8.3.3.3.1 General

Inputs to this process are:

— the current partition given by its partition index mbPartIdx;

— the width and height partWidth, partHeight of this partition in luma-sample units;

— a luma motion vector mvX given in quarter-luma-sample units;

— a chroma motion vector mvCX given in eighth-chroma-sample units; and

— the selected reference frame sample arrays refPicXL, refPicXCb, and refPicXCr.

Outputs of this process are:

— a partWidth x partHeight array $predPartX_L$ of prediction luma sample values; and

— two (partWidth/2) × (partHeight/2) arrays $predPartX_{Cb}$, and $predPartX_{Cr}$ of prediction chroma sample values.

Let (xA_L, yA_L) be the location given in full-sample units of the upper-left luma sample of the current partition given by mbPartIdx relative to the upper-left luma sample location of the given two-dimensional array of luma samples.

Let ($xInt_L$, $yInt_L$) be a luma location given in full-sample units and ($xFrac_L$, $yFrac_L$) be an offset given in quartersample units. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays refPicXL, refPicXCb, and refPicXCr.

For each luma sample location ($0 <= x_L <$ partWidth, $0 <= y_L <$ partHeight) inside the prediction luma sample array $predPartX_L$, the corresponding predicted luma sample value $predPartX_L[x_L, y_L]$ is derived as follows.

$xIntL = xAL + (mvX[0] >> 2) + x_L$

$yIntL = yAL + (mvX[1] >> 2) + yL$

$xFracL = mvX[0] \& 3$

$yFracL = mvX[1] \& 3$

— The prediction sample value $predPartX_L[xL, yL]$ is derived by invoking the process specified in subclause 8.3.3.3.2 with (xIntL, yIntL), (xFracL, yFracL) and refPicXL given as input.

Let ($xInt_c$, $yInt_c$) be a chroma location given in full-sample units and ($xFrac_c$, $yFrac_c$) be an offset given in one-eighth sample units. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays refPicXCb, and refPicXCr.

For each chroma sample location ($0 <= x_c <$ partWidth/2, $0 <= y_c <$ partHeight/2) inside the prediction chroma sample arrays $predPartX_{Cb}$ and $predPartX_{Cr}$, the corresponding prediction chroma sample values $predPartX_{Cb}[x_c, y_c]$ and $predPartX_{Cr}[x_c, y_c]$ are derived as follows.

$xInt_c = (xAL >> 1) + (mvCX[0] >> 3) + x_c$

$yInt_c = (yAL >> 1) + (mvCX[1] >> 3) + y_c$

$xFrac_c = mvCX[0] \& 7$

$yFrac_c = mvCX[1] \& 7$

— The prediction sample value $predPartX_{Cb}[x_c, y_c]$ is derived by invoking the process specified in subclause 8.3.3.3.3 with ($xInt_c$, $yInt_c$), ($xFra_c$, $yFrac_c$) and refPicXCb given as input.

— The prediction sample value $predPartX_{Cr}[x_c, y_c]$ is derived by invoking the process specified in subclause 8.3.3.3.3 with ($xInt_c$, $yInt_c$), ($xFrac_c$, $yFrac_c$) and refPicXCr given as input.

8.3.3.3.2 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 (xFracL, yFracL);

— the luma reference sample array refPicXL.

Output of this process is a predicted luma sample value $predPartX_L[x_L, y_L]$

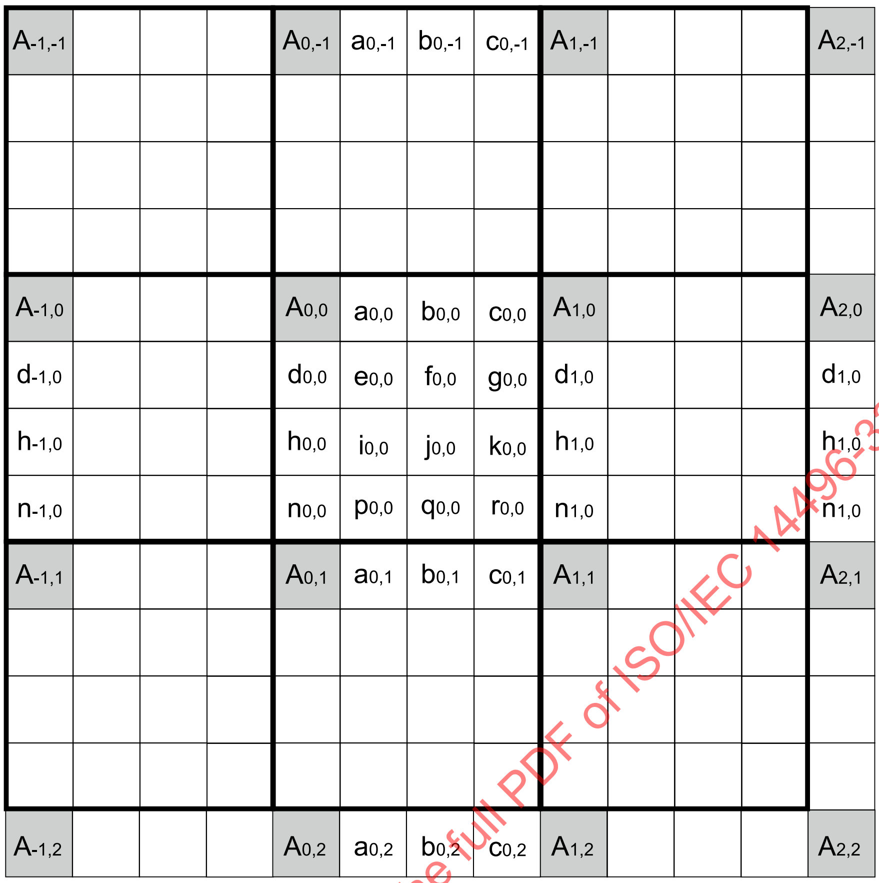


Figure 13 — Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for quarter sample luma interpolation

In Figure 13, the positions labelled with upper-case letters $A_{i,j}$ within shaded blocks represent luma samples at full-sample locations inside the given two-dimensional array refPicXL of luma samples. These samples may be used for generating the predicted luma sample value $predPartX_L[\, x_L, y_L \,]$. The locations ($xA_{i,j}$, $yA_{i,j}$) for each of the corresponding luma samples $A_{i,j}$ inside the given array refPicXL of luma samples are derived as follows:

$$xA_{i,j} = clip3(\, 0, PicWidth - 1, xInt_L + i \,)$$

$$yA_{i,j} = clip3(\, 0, PicHeight - 1, yInt_L + j \,)$$

The positions labelled with lower-case letters within un-shaded blocks represent luma samples at quarter-pel sample fractional locations. The luma location offset in fractional-sample units ($xFrac_L$, $yFrac_L$) specifies which of the generated luma samples at full-sample and fractional-sample locations is assigned to the predicted luma sample value $predPartX_L[\, x_L, y_L \,]$. This assignment is done according to Table 25. The value of $predPartX_L[\, x_L, y_L \,]$ is the output.

Table 25 — Assignment of the luma prediction sample $predPartX_L[\, x_L, y_L \,]$

$xFrac_L$	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
$yFrac_L$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
$predPartX_L[\, x_L, y_L \,]$	A	d	h	n	a	e	i	p	b	f	j	q	c	g	k	r

Given the luma samples $A_{i,j}$ at full-sample locations ($xA_{i,j}$, $yA_{i,j}$), the luma samples from '$a_{0,0}$' to '$r_{0,0}$' at fractional sample positions are derived by the following equations.

— If PicHeight is larger than or equal to 1600,

The samples labelled $a_{0,0}$, $b_{0,0}$, $c_{0,0}$, $d_{0,0}$, $h_{0,0}$, and $n_{0,0}$ are derived by applying the 4-tap filter to their nearest integer position samples, respectively, as follows.

$a'_{0,0} = -6 * A_{-1,0} + 56 * A_{0,0} + 15 * A_{1,0} - A_{2,0}$

$b'_{0,0} = -4 * A_{-1,0} + 36 * A_{0,0} + 36 * A_{1,0} - 4 * A_{2,0}$

$c'_{0,0} = -A_{-1,0} + 15 * A_{0,0} + 56 * A_{1,0} - 6 * A_{2,0}$

$d'_{0,0} = -6 * A_{0,-1} + 56 * A_{0,0} + 15 * A_{0,1} - A_{0,2}$

$h'_{0,0} = -4*A_{0,-1} + 36*A_{0,0} + 36*A_{0,1} - 4*A_{0,2}$

$n'_{0,0} = -A_{0,-1} + 15*A_{0,0} + 56*A_{0,1} - 6*A_{0,2}$

$a_{0,0} = a'_{0,0} >> 6$

$b_{0,0} = b'_{0,0} >> 6$

$c_{0,0} = c'_{0,0} >> 6$

$d_{0,0} = d'_{0,0} >> 6$

$h_{0,0} = h'_{0,0} >> 6$

$n_{0,0} = n'_{0,0} >> 6$

The samples labelled $e_{0,0}$, $i_{0,0}$, $p_{0,0}$, $f_{0,0}$, $j_{0,0}$, $q_{0,0}$, $g_{0,0}$, $k_{0,0}$ and $r_{0,0}$ are derived by applying the 4-tap filter to the samples $a'_{0,i}$, $b'_{0,i}$ and $c_{0,i}$ where i = −1..2 in the vertical direction, respectively, as follows.

$e_{0,0} = (-6 * a'_{0,-1} + 56 * a'_{0,0} + 15 * a'_{0,1} - a'_{0,2}) >> 12$

$i_{0,0} = (-4 * a'_{0,-1} + 36 * a'_{0,0} + 36 * a'_{0,1} - 4 * a'_{0,2}) >> 12$

$p_{0,0} = (-a'_{0,-1} + 15 * a'_{0,0} + 56 * a'_{0,1} - 6 * a'_{0,2}) >> 12$

$f_{0,0} = (-6 * b'_{0,-1} + 56 * b'_{0,0} + 15 * b'_{0,1} - b'_{0,2}) >> 12$

$j_{0,0} = (-4 * b'_{0,-1} + 36 * b'_{0,0} + 36 * b'_{0,1} - 4 * b'_{0,2}) >> 12$

$q_{0,0} = (-b'_{0,-1} + 15 * b'_{0,0} + 56 * b'_{0,1} - 6 * b'_{0,2}) >> 12$

$g_{0,0} = (-6 * c'_{0,-1} + 56 * c'_{0,0} + 15 * c'_{0,1} - c'_{0,2}) >> 12$

$k_{0,0} = (- 4 * c'_{0,-1} + 36 * c'_{0,0} + 36 * c'_{0,1} - 4 * c'_{0,2}) >> 12$

$r_{0,0} = (- c'_{0,-1} + 15 * c'_{0,0} + 56 * c'_{0,1} - c'_{0,2}) >> 12$

— Otherwise, if PicHeight is larger than or equal to 720,

The samples labelled $a_{0,0}$, $b_{0,0}$, $c_{0,0}$, $d_{0,0}$, $h_{0,0}$, and $n_{0,0}$ are derived by applying the 6-tap filter to their nearest integer position samples, respectively, as follows.

$a'_{0,0} = 2 * A_{-2,0} - 9 * A_{-1,0} + 57 * A_{0,0} + 17 * A_{1,0} - 4 * A_{2,0} + A_{3,0}$

$b'_{0,0} = 2 * A_{-2,0} - 9 * A_{-1,0} + 39 * A_{0,0} + 39 * A_{1,0} - 9 * A_{2,0} + 2 * A_{3,0}$

$c'_{0,0} = A_{-2,0} - 4 * A_{-1,0} + 17 * A_{0,0} + 57 * A_{1,0} - 9 * A_{2,0} + 2 * A_{3,0}$

$d'_{0,0} = 2 * A_{0,-2} - 9 * A_{0,-1} + 57 * A_{0,0} + 17 * A_{0,1} - 4 * A_{0,2} + A_{0,3}$

$h'_{0,0} = 2 * A_{0,-2} - 9 * A_{0,-1} + 39 * A_{0,0} + 39 * A_{0,1} - 9 * A_{0,2} + 2 * A_{0,3}$

$n'_{0,0} = A_{0,-2} - 4 * A_{0,-1} + 17 * A_{0,0} + 57 * A_{0,1} - 9 * A_{0,2} + 2 * A_{0,3}$

$a_{0,0} = a'_{0,0} >> 6$

$b_{0,0} = b'_{0,0} >> 6$

$c_{0,0} = c'_{0,0} >> 6$

$d_{0,0} = d'_{0,0} >> 6$

$h_{0,0} = h'_{0,0} >> 6$

$n_{0,0} = n'_{0,0} >> 6$

The samples labelled $e_{0,0}$, $i_{0,0}$, $p_{0,0}$, $f_{0,0}$, $j_{0,0}$, $q_{0,0}$, $g_{0,0}$, $k_{0,0}$ and $r_{0,0}$ are derived by applying the 6-tap filter to the samples $a'_{0,i}$, $b'_{0,i}$ and $c_{0,i}$ where i = −2..3 in vertical direction, respectively, as follows.

$e_{0,0} = (2 * a'_{0,-2} - 9 * a'_{0,-1} + 57 * a'_{0,0} + 17 * a'_{0,1} - 4 * a'_{0,2} + a'_{0,3}) >> 12$

$i_{0,0} = (2 * a'_{0,-2} - 9 * a'_{0,-1} + 39 * a'_{0,0} + 39 * a'_{0,1} - 9 * a'_{0,2} + 2 * a'_{0,3}) >> 12$

$p_{0,0} = (a'_{0,-2} - 4 * a'_{0,-1} + 17 * a'_{0,0} + 57 * a'_{0,1} - 9 * a'_{0,2} + 2 * a'_{0,3}) >> 12$

$f_{0,0} = (2 * b'_{0,-2} - 9 * b'_{0,-1} + 57 * b'_{0,0} + 17 * b'_{0,1} - 4 * b'_{0,2} + b'_{0,3}) >> 12$

$j_{0,0} = (2 * b'_{0,-2} - 9 * b'_{0,-1} + 39 * b'_{0,0} + 39 * b'_{0,1} - 9 * b'_{0,2} + 2 * b'_{0,3}) >> 12$

$q_{0,0} = (b'_{0,-2} - 4 * b'_{0,-1} + 17 * b'_{0,0} + 57 * b'_{0,1} - 9 * b'_{0,2} + 2 * b'_{0,3}) >> 12$

$g_{0,0}$ = (2 * $c'_{0,-2}$ − 9 * $c'_{0,-1}$ + 57 * $c'_{0,0}$ + 17 * $c'_{0,1}$ − 4 * $c'_{0,2}$ + $c'_{0,3}$) >> 12

$k_{0,0}$ = (2 * $c'_{0,-2}$ − 9 * $c'_{0,-1}$ + 39 * $c'_{0,0}$ + 39 * $c'_{0,1}$ − 9 * $c'_{0,2}$ + 2 * $c'_{0,3}$) >> 12

$r_{0,0}$ = ($c'_{0,-2}$ −4 * $c'_{0,-1}$ + 17 * $c'_{0,0}$ + 57 * $c'_{0,1}$ − 9 * $c'_{0,2}$ + 2 * $c'_{0,3}$) >> 12

— Otherwise,

The samples labelled $a_{0,0}$, $b_{0,0}$, $c_{0,0}$, $d_{0,0}$, $h_{0,0}$, and $n_{0,0}$ are derived by applying the following 10-tap filter to their nearest integer position samples, respectively, as follows.

$a'_{0,0}$ = $A_{-4,0}$ − 2 * $A_{-3,0}$ + 4 * $A_{-2,0}$ − 10 * $A_{-1,0}$ + 57 * $A_{0,0}$ + 19 * $A_{1,0}$ − 7 * $A_{2,0}$ + 3 * $A_{3,0}$ − $A_{4,0}$

$b'_{0,0}$ =$A_{-4,0}$ − 2 * $A_{-3,0}$ + 5 * $A_{-2,0}$ − 12 * $A_{-1,0}$ + 40 * $A_{0,0}$ + 40 * $A_{1,0}$ − 12 * $A_{2,0}$ + 5 * $A_{3,0}$ − 2 * $A_{4,0}$ + $A_{5,0}$

$c'_{0,0}$ = −$A_{-3,0}$ + 3 * $A_{-2,0}$ − 7 * $A_{-1,0}$ + 19 * $A_{0,0}$ + 57 * $A_{1,0}$ − 10 * $A_{2,0}$ + 4 * $A_{3,0}$ − 2 * $A_{4,0}$ + $A_{5,0}$

$d'_{0,0}$ = $A_{0,-4}$ − 2 * $A_{0,-3}$ + 4 * $A_{0,-2}$ − 10 * $A_{0,-1}$ + 57 * $A_{0,0}$ + 19 * $A_{0,1}$ − 7 * $A_{0,2}$ + 3 * $A_{0,3}$ − $A_{0,4}$

$h'_{0,0}$ = $A_{0,-4}$ − 2 * $A_{0,-3}$ + 5 * $A_{0,-2}$ − 12 * $A_{0,-1}$ + 40 * $A_{0,0}$ + 40 * $A_{0,1}$ − 12 * $A_{0,2}$ + 5 * $A_{0,3}$ −2 * $A_{0,4}$ + $A_{0,5}$

$n'_{0,0}$ = −$A_{0,-3}$ + 3 * $A_{0,-2}$ − 7 * $A_{0,-1}$ + 19 * $A_{0,0}$ + 57 * $A_{0,1}$ − 10 * $A_{0,2}$ + 4 * $A_{0,3}$ − 2 * $A_{0,4}$ + $A_{0,5}$

$a_{0,0}$ = $a'_{0,0}$ >> 6

$b_{0,0}$ = $b'_{0,0}$ >> 6

$c_{0,0}$ = $c'_{0,0}$ >> 6

$d_{0,0}$ = $d'_{0,0}$ >> 6

$h_{0,0}$ = $h'_{0,0}$ >> 6

$n_{0,0}$ = $n'_{0,0}$ >> 6

The samples labelled $e_{0,0}$, $i_{0,0}$, $p_{0,0}$, $f_{0,0}$, $j_{0,0}$, $q_{0,0}$, $g_{0,0}$, $k_{0,0}$ and $r_{0,0}$ are derived by applying the following 10-tap filter to the samples $a'_{0,i}$, $b'_{0,i}$ and $c_{0,i}$ where i = −4..5 in vertical direction, respectively, as follows.

$e_{0,0}$ = ($a'_{0,-4}$ − 2 * $a'_{0,-3}$ + 4 * $a'_{0,-2}$ − 10 * $a'_{0,-1}$ + 57 * $a'_{0,0}$ + 19 * $a'_{0,1}$ − 7 * $a'_{0,2}$ + 3 * $a'_{0,3}$ − $a'_{0,4}$) >> 12

$i_{0,0}$ = ($a'_{0,-4}$ − 2 * $a'_{0,-3}$ + 5 * $a'_{0,-2}$ − 12 * $a'_{0,-1}$ + 40 * $a'_{0,0}$ +40 * $a'_{0,1}$ − 12*$a'_{0,2}$ + 5*$a'_{0,3}$ − 2*$a'_{0,4}$ + $a'_{0,5}$) >> 12

$p_{0,0}$ = (−$a'_{0,-3}$ + 3*$a'_{0,-2}$ − 7*$a'_{0,-1}$ + 19*$a'_{0,0}$ +57*$a'_{0,1}$ − 10*$a'_{0,2}$ + 4*$a'_{0,3}$ − 2*$a'_{0,4}$ + $a'_{0,5}$) >> 12

$f_{0,0}$ = ($b'_{0,-4}$ − 2*$b'_{0,-3}$ + 4*$b'_{0,-2}$ − 10*$b'_{0,-1}$ + 57*$b'_{0,0}$ +19*$b'_{0,1}$ − 7*$a'_{0,2}$ + 3*$b'_{0,3}$ − $b'_{0,4}$) >> 12

$j_{0,0}$ = ($b'_{0,-4}$ − $2*b'_{0,-3}$ + $5*b'_{0,-2}$ − $12*b'_{0,-1}$ + $40*b'_{0,0}$ +$40*b'_{0,1}$ − $12*b'_{0,2}$ + $5*b'_{0,3}$ − $2*b'_{0,4}$ + $b'_{0,5}$) >> 12

$q_{0,0}$ = ($-b'_{0,-3}$ + $3*b'_{0,-2}$ − $7*b'_{0,-1}$ + $19*b'_{0,0}$ +$57*b'_{0,1}$ − $10*b'_{0,2}$ + $4*b'_{0,3}$ − $2*b'_{0,4}$ + $b'_{0,5}$) >> 12

$g_{0,0}$ = ($c'_{0,-4}$ − $2*c'_{0,-3}$ + $4*c'_{0,-2}$ − $10*c'_{0,-1}$ + $57*c'_{0,0}$ +$19*c'_{0,1}$ − $7*c'_{0,2}$ + $3*c'_{0,3}$ − $c'_{0,4}$) >> 12

$k_{0,0}$ = ($c'_{0,-4}$ − $2*c'_{0,-3}$ + $5*c'_{0,-2}$ − $12*c'_{0,-1}$ + $40*c'_{0,0}$ +$40*c'_{0,1}$ − $12*c'_{0,2}$ + $5*c'_{0,3}$ − $2*c'_{0,4}$ + $c'_{0,5}$) >> 12

$r_{0,0}$ = ($-c'_{0,-3}$ + $3*c'_{0,-2}$ − $7*c'_{0,-1}$ + $19*c'_{0,0}$ +$57*c'_{0,1}$ − $10*c'_{0,2}$ + $4*c'_{0,3}$ − $2*c'_{0,4}$ + $c'_{0,5}$) >> 12

8.3.3.3.3 Chroma sample interpolation process

Inputs to this process are:

— a chroma location in full-sample units ($xInt_C$, $yInt_C$);

— a chroma location offset in fractional-sample units ($xFrac_C$, $yFrac_C$);

— chroma component samples from the selected reference frame refPicXC.

Output of this process is a predicted chroma sample value $predPartX_C$[x_C, y_C].

In Figure 14, the positions labelled with upper-case letters $A_{i,j}$ within shaded blocks represent chroma samples at full-sample locations inside the given two-dimensional array refPicXC of chroma samples. These samples may be used for generating the predicted chroma sample value $predPartX_C$[x_C, y_C]. The locations ($xA_{i,j}$, $yA_{i,j}$) for each of the corresponding chroma samples $A_{i,j}$ inside the given array refPicXC of chroma samples are derived as follows:

$xA_{i,j}$ = clip3(0, PicWidth / 2 − 1, $xInt_C$ + i)

$yA_{i,j}$ = clip3(0, PicHeight / 2 − 1, $yInt_C$ + j)

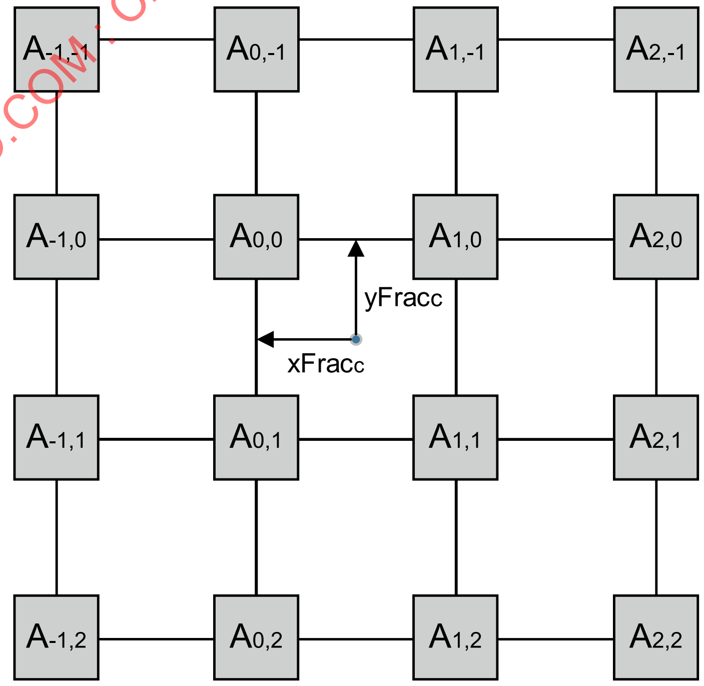


Figure 14 — Relation between variable positions and reference samples

A two-dimensional array C is defined as:

C[8, 4] = {

{ 0, 64, 0, 0 },

{ −4, 62, 6, 0 },

{ −6, 56, 15, −1 },

{ −5, 47, 25, −3 },

{ −4, 36, 36, −4 },

{ −3, 25, 47, −5 },

{ −1, 15, 56, −6 },

{ 0, 6, 62, −4 }

}

The elements of interpolated sample matrix $predPartX_C[\ x_c, y_c]$ are calculated as:

if($xFrac_c$ == 0)

$$predPartX_C[\ x_c, y_c] = (C[yFrac_c][0] * A_{0,-1} + C[yFrac_c][1] * A_{0,0} + C[yFrac_c][2] * A_{0,1} + C[yFrac_c][3] * A_{0,2} + 32) >> 6$$

else if($yFrac_c$ == 0)

$$predPartX_C[\ x_c, y_c] = (C[xFrac_c][0]) * A_{-1,0} + C[xFrac_c][1] * A_{0,0} + C[xFrac_c][2] * A_{1,0} + C[xFrac_c][3] * A_{2,0} + 32) >> 6$$

else

$$predPartX_C[\ x_c, y_c] = (C[yFrac_c][0] * a'_{0,-1}(xFrac_c,0) + C[yFrac_c][1] * a'_{0,0}(xFrac_c,0) + C[yFrac_c][2] * a'_{0,\ 1}(xFrac_c,0) + C[yFrac_c][3] * a'_{0,\ 2}(xFrac_c,0) + 2048) >> 12$$

where $a'_{0,-1}(xFrac_c, 0)$, $a'_{0,0}(xFrac_c, 0)$, $a'_{0,1}(xFrac_c, 0)$ and $a'_{0,2}(xFrac_c, 0)$, are calculated by

$$a'_{0,-1}(xFrac_c, 0) = C[xFrac_c][0] * A_{-1,-1} + C[xFrac_c][1] * A_{0,-1} + C[xFrac_c][2] * A_{1,-1} + C[xFrac_c][3] * A_{2,-1}$$

$$a'_{0,0}(xFrac_c, 0) = C[xFrac_c][0] * A_{-1,0} + C[xFrac_c][1] * A_{0,0} + C[xFrac_c][2] * A_{1,0} + C[xFrac_c][3] * A_{2,0}$$

$$a'_{0,1}(xFrac_c, 0) = C[xFrac_c][0] * A_{-1,1} + C[xFrac_c][1] * A_{0,1} + C[xFrac_c][2] * A_{1,1} + C[xFrac_c][3] * A_{2,1}$$

$$a'_{0,2}(xFrac_c, 0) = C[xFrac_c][0] * A_{-1,2} + C[xFrac_c][1] * A_{0,2} + C[xFrac_c][2] * A_{1,2} + C[xFrac_c][3] * A_{2,2}$$

8.3.3.4 Combining predictions

Inputs to this process are:

— mbPartIdx: the current partition given by the partition index;

— predFlagFst and predFlagSnd: prediction list utilization flags;

— $predPartX_L$: a partWidth x partHeight array of prediction luma samples (with X being replaced by ‘Fst’ or ‘Snd’ depending on predFlagFst and predFlagSnd);

— $predPartX_{Cb}$ and $predPartX_{Cr}$: two (partWidth/2) x (partHeight/2) arrays of prediction chroma samples, one for each of the chroma components Cb and Cr (with X being replaced by 'Fst' or 'Snd' depending on predFlagFst and predFlagSnd).

Outputs of this process are:

— $predPart_L$: a partWidth x partHeight array of prediction luma samples;

— $predPart_{Cb}$ and $predPart_{Cr}$: (partWidth/2)x(partHeight/2) arrays of prediction chroma samples, one for each of the chroma components Cb and Cr.

Depending on the component for which the prediction block is derived, the following applies.

— If the luma sample prediction values $predPart_L[x, y]$ are derived, the following applies with C set equal to L, x set equal to 0 .. partWidth − 1, and y set equal to 0 .. partHeight − 1.

— Otherwise, if the chroma Cb component sample prediction values $predPart_{Cb}[x, y]$ are derived, the following applies with C set equal to Cb, x set equal to 0 .. partWidth / 2 − 1, and y set equal to 0 .. partHeight / 2 − 1.

— Otherwise (the chroma Cr component sample prediction values $predPart_{Cr}[x, y]$ are derived), the following applies with C set equal to Cr, x set equal to 0 .. partWidth / 2 − 1, and y set equal to 0 .. partHeight / 2 − 1.

The prediction sample values are derived as follows.

— If predFlagFst is equal to 1 and predFlagSnd is equal to 0 for the current partition,

$predPart_C[x, y] = predPartFst_C[x, y]$

— Otherwise, if predFlagFst is equal to 0 and predFlagSnd is equal to 1 for the current partition,

$predPart_C[x, y]= predPartSnd_C[x, y]$

— Otherwise (both predFlagFst and predFlagSnd are equal to 1 for the current partition),

$predPart_C[x, y] = (predPartFst_C[x, y] + predPartSnd_C[x, y] + 1) >> 1$

8.4 Transform coefficient decoding process and frame reconstruction process

8.4.1 General

This subclause specifies transform coefficient decoding and frame reconstruction prior to the deblocking filter process.

Inputs to this process are quantized transform coefficients for luma and chroma components, and available Inter or Intra prediction sample arrays for the current macroblock for the applicable component $pred_L$, $pred_{Cb}$, or $pred_{Cr}$.

Outputs of this process are the reconstructed sample arrays prior to the deblocking filter process for the applicable component S_L, S_{Cb}, or S_{Cr}.

When the MbPredType of current macroblock is 'Pred_Skip' or the MbPartType of current macroblock is 'B_Skip', all values of quantized transform coefficients are set equal to 0 for the current macroblock.

8.4.2 Inverse scanning

This subclause specifies the inverse scanning process for block coefficients in Zigzag order.

— If MbTransformType is 'Trans_8x8' and SubMbTransformType is 'Trans_4x4',

  — Input of this process is an array Q (derived from subclause 9.3) with size of 16. The elements of the array are Q[n], with 0 <= n <= 15.

  — Output of this process is a two-dimensional array QuantCoeffMatrix with size of 4x4. The elements of the array are QuantCoeffMatrix[i, j], with 0 <= i <= 3, 0 <= j <= 3.

The conversion between the array Q and QuantCoeffMatrix is: QuantCoeffMatrix[i,j] = Q[n], and the relationship between i, j and n is defined as follows.

```
IVC_SCAN4[16] = {
    0, 1, 4, 8,
    5, 2, 3, 6,
    9, 12, 13, 10,
    7, 11, 14, 15
}
i = IVC_SCAN4[n] / 4,
j = IVC_SCAN4[n] % 4
```

— Otherwise, if MbTransformType is 'Trans_8x8' and SubMbTransformType is 'Trans_8x8',

  — Input of this process is an array Q (derived from subclause 9.3) with size of 64. The elements of the array are Q[n], with 0 <= n <= 63.

  — Output of this process is a two-dimensional array QuantCoeffMatrix with size of 8x8. The elements of the array are QuantCoeffMatrix[i,j], with 0 <= i <= 7, 0 <= j <= 7.

The conversion between the array Q and QuantCoeffMatrix is: QuantCoeffMatrix[i,j] = Q[n], and Table 26 shows the mapping from the index n of Q to the indices i and j of the array QuantCoeffMatrix.

Table 26 — Inverse scanning order of 8x8 block

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	1	0	0	1	2	3	2	1	0	0	1	2	3	4	5
j	0	0	1	2	1	0	0	1	2	3	4	3	2	1	0	0
n	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i	4	3	2	1	0	0	1	2	3	4	5	6	7	6	5	4
j	1	2	3	4	5	6	5	4	3	2	1	0	0	1	2	3
n	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
i	3	2	1	0	1	2	3	4	5	6	7	7	6	5	4	3
j	4	5	6	7	7	6	5	4	3	2	1	2	3	4	5	6
n	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
i	2	3	4	5	6	7	7	6	5	4	5	6	7	7	6	7
j	7	7	6	5	4	3	4	5	6	7	7	6	5	6	7	7

— Otherwise(the MbTransformType is 'Trans_16x16'),

  — Input of this process is an array Q (derived from subclause 9.3) with size of 256. The elements of the array are Q[n], with 0 <= n <= 255.

— Output of this process is a two-dimensional array QuantCoeffMatrix with size of 16x16. The elements of the array are QuantCoeffMatrix[i,j], with 0 <= i <= 16, 0 <= j <= 16.

The conversion between the array Q and QuantCoeffMatrix is: QuantCoeffMatrix[i,j] = Q[n], and the relationship between i, j and n is defined as follows.

```
IVC_SCAN16[256] = {
    0, 1, 16, 32, 17, 2, 3, 18, 33, 48, 64, 49, 34, 19, 4, 5,
    20, 35, 50, 65, 80, 96, 81, 66, 51, 36, 21, 6, 7, 22, 37, 52,
    67, 82, 97, 112, 128, 113, 98, 83, 68, 53, 38, 23, 8, 9, 24, 39,
    54, 69, 84, 99, 114, 129, 144, 160, 145, 130, 115, 100, 85, 70, 55, 40,
    25, 10, 11, 26, 41, 56, 71, 86, 101, 116, 131, 146, 161, 176, 192, 177,
    162, 147, 132, 117, 102, 87, 72, 57, 42, 27, 12, 13, 28, 43, 58, 73,
    88, 103, 118, 133, 148, 163, 178, 193, 208, 224, 209, 194, 179, 164, 149, 134,
    119, 104, 89, 74, 59, 44, 29, 14, 15, 30, 45, 60, 75, 90, 105, 120,
    135, 150, 165, 180, 195, 210, 225, 240, 241, 226, 211, 196, 181, 166, 151, 136,
    121, 106, 91, 76, 61, 46, 31, 47, 62, 77, 92, 107, 122, 137, 152, 167,
    182, 197, 212, 227, 242, 243, 228, 213, 198, 183, 168, 153, 138, 123, 108, 93,
    78, 63, 79, 94, 109, 124, 139, 154, 169, 184, 199, 214, 229, 244, 245, 230,
    215, 200, 185, 170, 155, 140, 125, 110, 95, 111, 126, 141, 156, 171, 186, 201,
    216, 231, 246, 247, 232, 217, 202, 187, 172, 157, 142, 127, 143, 158, 173, 188,
    203, 218, 233, 248, 249, 234, 219, 204, 189, 174, 159, 175, 190, 205, 220, 235,
    250, 251, 236, 221, 206, 191, 207, 222, 237, 252, 253, 238, 223, 239, 254, 255
}
i = IVC_SCAN16[n] / 16,
j = IVC_SCAN16[n] % 16
```

8.4.3 Inverse quantization

8.4.3.1 Quantization parameter

The range of quantization parameters for the luma component is 0..63, inclusive, and the range of quantization parameters for the chroma components is 0..51, inclusive.

The variables CurrentQP and PreviousDeltaQP for the current macroblock are first derived as follows.

PreviousDeltaQP is initialized to 0 if the current macroblock is the first macroblock in the current frame.

— If fixed_frame_level_qp is 1,

SliceQP = frame_qp

— Otherwise, if fixed_slice_level_qp is 1,

SliceQP = slice_qp

— Otherwise, if mb_qp_delta is not present in the bitstream for the current macroblock,

SliceQP = slice_qp

mb_qp_delta = 0

CurrentQP = clip3(0, 63, SliceQP + mb_qp_delta)

PreviousDeltaQP = mb_qp_delta

— Otherwise, mb_qp_delta is parsed from bitstream as specified in subclause 9.4,

SliceQP = slice_qp

CurrentQP = clip3(0, 63, SliceQP + mb_qp_delta)

PreviousDeltaQP = mb_qp_delta

If the current block is a luma block, quantization parameter QP of the block is set equal to CurrentQP of the macroblock which it belongs to. Otherwise, CurrentQP is used as an index to get the QP values of chroma blocks, respectively, from Table 27.

Table 27 — CurrentQPCb, CurrentQPCr and QP of chroma blocks

CurrentQP	Chroma QP
<43	CurrentQP
43	42
44	43
45	43
46	44
47	44
48	45
49	45
50	46
51	46
52	47
53	47
54	48
55	48
56	48
57	49
58	49
59	49
60	50
61	50
62	50
63	51

8.4.3.2 Inverse quantization process

This clause specifies the process to transform a two dimensional quantized transform coefficient array QuantCoeffMatrix (derived from subclause 8.4.2) to a two dimensional transform coefficient array D using quantization parameter QP.

Two dimensional transform coefficients array D is obtained by:

D[i,j] = (QuantCoeffMatrix[i,j] * DequantTable(QP) + (1 << (ShiftTable(QP)−1)) >> ShiftTable(QP), i,j=0..7

DequantTable and ShiftTable are defined in Table 28.

Table 28 — DequantTable and ShiftTable

QP	DequantTable(QP)	ShiftTable(QP)
0	32768	14
1	36061	14
2	38968	14
3	42495	14
4	46341	14
5	50535	14
6	55437	14
7	60424	14
8	32932	13
9	35734	13
10	38968	13
11	42495	13
12	46177	13
13	50535	13
14	55109	13
15	59933	13
16	65535	13
17	35734	12
18	38968	12
19	42577	12
20	46341	12
21	50617	12
22	55027	12
23	60097	12
24	32809	11
25	35734	11
26	38968	11
27	42454	11
28	46382	11
29	50576	11
30	55109	11
31	60056	11
32	65535	11
33	35734	10
34	38968	10

Table 28 *(continued)*

QP	DequantTable(QP)	ShiftTable(QP)
35	42495	10
36	46320	10
37	50515	10
38	55109	10
39	60076	10
40	65535	10
41	35744	9
42	38968	9
43	42495	9
44	46341	9
45	50535	9
46	55099	9
47	60087	9
48	65535	9
49	35734	8
50	38973	8
51	42500	8
52	46341	8
53	50535	8
54	55109	8
55	60097	8
56	32771	7
57	35734	7
58	38965	7
59	42497	7
60	46341	7
61	50535	7
62	55109	7
63	60099	7

8.4.4 Inverse transform process

8.4.4.1 Inverse transform for 4x4 block

This process of transform is applied to 4x4 block when MbTransformType is ‘Trans_8x8’ and SubMbTransformType is ‘Trans_4x4’.

Inputs of this process are:

— the variables of BitDepth;

— a two-dimensional array D (derived from subclause 8.4.3.2) with size of 4x4. The elements of the array are D_{ij}, with 0 <= i <= 3, 0 <= j <= 3.

Output of this process is

— a two-dimensional array R with size of 4x4. The elements of the array are R_{ij}, with 0 <= i <= 3, 0 <= j <= 3.

The 4x4 DCT transform core T_4 is defined as:

```
T4[4][4] = {
    {128, 128, 128, 128},
    {167, 69, −69, −167},
    {128, −128, −128, 128},
    {69, −167, 167, −69}
    }
```

The inverse transform process is specified as follows.

— Step1, horizontal inverse transform for the array D:

$H' = D * T_4^T$

Here, H' is the temporary result, T_4^T is the transpose of T_4

— Step2, vertical inverse transform on H' :

$H = T_4^T * H'$

— Step3, shift operation on H:

$R_{i,j} = \mathrm{sign}(\ \mathrm{abs}(H_{i,j}) + (1<<16)) >> 17$

8.4.4.2 Inverse transform for 8x8 block

This process of transform is applied to 8x8 block when MbTransformType is 'Trans_8x8' and SubMbTransformType is 'Trans_8x8'.

Inputs of this process are:

— the variables of BitDepth;

— a two-dimensional array D (derived from subclause 8.4.3.2) with size of 8x8, the elements of the array are d_{ij}, with 0 <= i <= 7, 0 <= j <= 7.

Output of this process is

— a two-dimensional array R with size of 8x8, the elements of the array are r_{ij}, with 0 <= i <= 7, 0 <= j <= 7

The inverse transform process is specified as follows.

— First, horizontal transform for the array D:

Step 1, with i = 0, 1, ... , 7

$e_{i0} = (d_{i0} + d_{i4}) * 181 >> 7$

$e_{i1} = (d_{i0} - d_{i4}) * 181 >> 7$

$e_{i2} = (d_{i2} * 196 >> 8) - (d_{i6} * 473 >> 8)$

$e_{i3} = (d_{i2} * 473 >> 8) + (d_{i6} * 196 >> 8)$

$t_{i4} = d_{i1} - d_{i7}$

$t_{i7} = d_{i1} + d_{i7}$

$t_{i5} = d_{i3} * 181 >> 7$

$t_{i6} = d_{i5} * 181 >> 7$

$e_{i4} = t_{i4} + t_{i6}$

$e_{i5} = t_{i7} - t_{i5}$

$e_{i6} = t_{i4} - t_{i6}$

$e_{i7} = t_{i7} + t_{i5}$

Data in the bitstream shall ensure that any element d_{ij}, t_{ij} and e_{ij} is in the range of integer values from $-2^{(BitDepth+7)}$ to $2^{(BitDepth+7)}-1$, inclusive.

Step 2, with i = 0, 1, … , 7

$f_{i0} = e_{i0} + e_{i3}$

$f_{i3} = e_{i0} - e_{i3}$

$f_{i1} = e_{i1} + e_{i2}$

$f_{i2} = e_{i1} - e_{i2}$

$f_{i4} = (e_{i4} * 301 >> 8) - (e_{i7} * 201 >> 8)$

$f_{i7} = (e_{i4} * 201 >> 8) + (e_{i7} * 301 >> 8)$

$f_{i5} = (e_{i5} * 710 >> 9) - (e_{i6} * 141 >> 9)$

$f_{i6} = (e_{i5} * 141 >> 9) + (e_{i6} * 710 >> 9)$

Data in the bitstream shall ensure that any element f_{ij} is in the range of integer values from $-2^{(BitDepth+7)}$ to $2^{(BitDepth+7)}-1$, inclusive.

Step 3, with i = 0, 1, … , 7

$g_{i0} = f_{i0} + f_{i7}$

$g_{i7} = f_{i0} - f_{i7}$

$g_{i1} = f_{i1} + f_{i6}$

$g_{i6} = f_{i1} - f_{i6}$

$g_{i2} = f_{i2} + f_{i5}$

$g_{i5} = f_{i2} - f_{i5}$

$g_{i3} = f_{i3} + f_{i4}$

$g_{i4} = f_{i3} - f_{i4}$

Data in the bitstream shall ensure that any element g_{ij} is in the range of integer values from $-2^{(BitDepth+7)}$ to $2^{(BitDepth+7)}-1$, inclusive.

— And then, vertical transform is invoked for the resulting matrix:

Step 1, with j = 0, 1, … , 7

$h_{0j} = (g_{0j} + g_{4j}) * 181 >> 7$

$h_{1j} = (g_{0j} - g_{4j}) * 181 >> 7$

$h_{2j} = (g_{2j} * 196 >> 8) - (g_{6j} * 473 >> 8)$

$h_{3j} = (g_{2j} * 473 >> 8) + (g_{6j} * 196 >> 8)$

$t_{4j} = g_{1j} - g_{7j}$

$t_{7j} = g_{1j} + g_{7j}$

$t_{5j} = g_{3j} * 181 >> 7$

$t_{6j} = g_{5j} * 181 >> 7$

$h_{4j} = t_{4j} + t_{6j}$

$h_{5j} = t_{7j} - t_{5j}$

$h_{6j} = t_{4j} - t_{6j}$

$h_{7j} = t_{7j} + t_{5j}$

Data in the bitstream shall ensure that any element h_{ij} is in the range of integer values from $-2^{(BitDepth+7)}$ to $2^{(BitDepth+7)}-1$, inclusive.

Step 2, with j = 0, 1, … , 7

$m_{0j} = h_{0j} + h_{3j}$

$m_{3j} = h_{0j} - h_{3j}$

$m_{1j} = h_{1j} + h_{2j}$

$m_{2j} = h_{1j} - h_{2j}$

$m_{4j} = (h_{4j} * 301 >> 8) - (h_{7j} * 201 >> 8)$

$m_{7j} = (h_{4j} * 201 >> 8) + (h_{7j} * 301 >> 8)$

$m_{5j} = (h_{5j} * 710 >> 9) - (h_{6j} * 141 >> 9)$

$m_{6j} = (h_{5j} * 141 >> 9) + (h_{6j} * 710 >> 9)$

Data in the bitstream shall ensure that any element m_{ij} is in the range of integer values from $-2^{(BitDepth+7)}$ to $2^{(BitDepth+7)}-1$, inclusive.

Step 3, with j = 0, 1, … , 7

$n_{0j} = m_{0j} + m_{7j}$

$n_{7j} = m_{0j} - m_{7j}$

$n_{1j} = m_{1j} + m_{6j}$

$n_{6j} = m_{1j} - m_{6j}$

$n_{2j} = m_{2j} + m_{5j}$

$n_{5j} = m_{2j} - m_{5j}$

$n_{3j} = m_{3j} + m_{4j}$

$n_{4j} = m_{3j} - m_{4j}$

Data in the bitstream shall ensure that any element n_{ij} is in the range of integer values from $-2^{(BitDepth+7)}$ to $2^{(BitDepth+7)}-1$, inclusive.

— At last, after horizontal and vertical transform, the final reconstructed values are derived as:

r_{ij} = sign ((abs(n_{ij}) + 16) >> 5, n_{ij}), with i=0,1...,7, j=0,1,...,7

8.4.4.3 Inverse transform for 16x16 block

This process of transform is applied to 16x16 block when MbTransformType is 'Trans_16x16'

Inputs of this process are:

— the variables of BitDepth;

— a two-dimensional array D (derived from subclause 8.4.3.2) with size of 16x16, the elements of the array are d_{ij}, with 0 <= i <= 15, 0 <= j <= 15.

Output of this process is

— a two-dimensional array R with size of 16x16, the elements of the array are r_{ij}, with 0 <= i <= 15, 0 <= j <= 15.

The inverse transform process is equivalent to the following.

The 16x16 DCT transform core T_{16} is defined as:

T_{16}[16][16] = {

{ 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32},

{ 45, 43, 40, 35, 29, 21, 13, 4, −4,−13,−21,−29,−35,−40,−43,−45},

{ 44, 38, 25, 9, −9,−25,−38,−44,−44,−38,−25, −9, 9, 25, 38, 44},

{ 43, 29, 4,−21,−40,−45,−35,−13, 13, 35, 45, 40, 21, −4,−29,−43},

{ 42, 17,−17,−42,−42,−17, 17, 42, 42, 17,−17,−42,−42,−17, 17, 42},

{ 40, 4,−35,−43,−13, 29, 45, 21,−21,−45,−29, 13, 43, 35, −4,−40},

{ 38, −9,−44,−25, 25, 44, 9,−38,−38, 9, 44, 25,−25,−44, −9, 38},

{ 35,−21,−43, 4, 45, 13,−40,−29, 29, 40,−13,−45, −4, 43, 21,−35},

{ 32,−32,−32, 32, 32,−32,−32, 32, 32,−32,−32, 32, 32,−32,−32, 32},

{ 29,−40,−13, 45, −4,−43, 21, 35,−35,−21, 43, 4,−45, 13, 40,−29},

{ 25,−44, 9, 38,−38, −9, 44,−25,−25, 44, −9,−38, 38, 9,−44, 25},

{ 21,−45, 29, 13,−43, 35, 4,−40, 40, −4,−35, 43,−13,−29, 45,−21},

{ 17,−42, 42,−17,−17, 42,−42, 17, 17,−42, 42,−17,−17, 42,−42, 17},

{ 13,−35, 45,−40, 21, 4,−29, 43,−43, 29, −4,−21, 40,−45, 35,−13},

{ 9,−25, 38,−44, 44,−38, 25, −9, −9, 25,−38, 44,−44, 38,−25, 9},

{ 4,−13, 21,−29, 35,−40, 43,−45, 45,−43, 40,−35, 29,−21, 13, −4}

}

The inverse transform process is specified as follows.

— Step1, horizontal inverse transform for the array D:

$H' = D * T_{16}^{T}$