Industria y Comercio
SUPERINTENDENCIA

Espacio reservado para el adhesivo de radicación

DIRECCIÓN DE NUEVAS CREACIONES
Procedimiento acelerado de exámenes de patente (PPH)

1	**NÚMERO DE SOLICITUD ANTE LA SIC NC2021/0007416**		
2	**GLOBAL - PPH** ☐ PAÍS	**3**	**PCT – PPH** ☐ PAÍS
4	**PROSUR** ☐ PAÍS	**5**	**Alianza del Pacifico (AdP)** ☐ PAÍS
6	**OFICINA DE PRIMERA PRESENTACIÓN (OPP)** ☐	**7**	**OFICINA DE EXAMEN ANTERIOR (OEA)** ☒
8	**NÚMERO DE SOLICITUD O PUBLICACIÓN DE LA SOLICITUD (OPP)** ☐ **NÚMERO DE SOLICITUD EN OFICINA DE EXAMEN ANTERIOR (OEA):** ☒ **NÚMERO DE SOLICITUD PCT:** ☐	1.	**16/642,748**
9	☐ **REPRESENTANTE LEGAL** ☒ **APODERADO**		

APELLIDOS	NOMBRES	IDENTIFICACIÓN	
DAZA MONTALVO	CAROLINA MERCEDES	C.C. 66783790	T.P. 141563 CSJ

DIRECCIÓN	Carrera 11 No. 86-53 Piso 3	No. TELÉFONO	6181088
CIUDAD	Bogotá	CORREO ELECTRÓNICO	notificaciones@clarkemodet.com.co
PAÍS	Colombia	No. RADICACIÓN O PROTOCOLO DE PODER GENERAL	

10 CONDICIONES PARA ACCEDER A ESTE PROCESO:

10.1. DATOS DE PRIORIDAD ☒ SI ☐ NO

(33) PAÍS DE ORIGEN	CÓDIGO PAÍS	(31) NÚMERO	(32) FECHA (AAAA/MM/DD)
1. ESTADOS UNIDOS DE AMÉRICA	US	62/757,536	2018/11/08

10.2. DATOS SOLICITUD INTERNACIONAL PCT ☒ SI ☐ NO

Solicitud Internacional No.	PCT/SE2019/051109	**Fecha**	2019/08/09
Publicación Internacional No.	WO/2020/096510	**Fecha**	2019/11/04

* El solicitante presenta una copia de la(s) reivindicación(es) patentable(s) u otorgable(s) de la solicitud de la Oficina de Primera Presentación que profirió el examen de patentabilidad, junto con una traducción al castellano. ☒ SI ☐ NO

* El solicitante presenta los resultados de búsqueda de la Oficina de Examen Anterior, junto con una traducción al castellano. ☒ SI ☐ NO

REQUISITOS PARA EL RECONOCIMIENTO DE LOS RESULTADOS DE EXÁMENES DE PATENTABILIDAD EFECTUADOS A SOLICITUDES EXTRANJERAS

Para que la Superintendencia de Industria y Comercio pueda reconocer los resultados de exámenes de patentabilidad producidos por otras Oficinas de Propiedad Industrial se debe cumplir con las siguientes condiciones:

(a) Al momento de solicitar el examen de patentabilidad, si el solicitante pide que se reconozcan exámenes de patentabilidad de otras oficinas extranjeras, deberá aportar junto con el pago de la tasa correspondiente, los siguientes anexos:

(i) Copia de las resoluciones, fallos o decisiones que han sido relevantes para la determinación de la patentabilidad de la(s) solicitud(es) en la Oficina que haya proferido el examen de patentabilidad que contienen reivindicaciones patentables u otorgables que son la base de la petición, y si es del caso, su traducción al castellano. ☒ SI ☐ NO

(ii) Copia de todas las reivindicaciones determinadas como patentables u otorgables en la Oficina que profirió el examen de patentabilidad y, si es del caso, su traducción al castellano. ☒ SI ☐ NO

(iii) Copia de los antecedentes, información o documentos citados por el examinador de la Oficina de Primera Presentación del examen de patentabilidad, en las resoluciones, fallos o decisiones correspondientes, incluyendo la literatura no patente. Los documentos de patente deberán presentarse siempre y cuando no estén disponibles para la Superintendencia de Industria y Comercio mediante el simple acceso a bases de datos gratuitas. ☒ SI ☐ NO

(iv) Una tabla de correspondencias de las reivindicaciones que han sido determinadas como patentables u otorgables en la solicitud de la Oficina que haya proferido el examen de patentabilidad. ☒ SI ☐ NO

11 FIRMA DEL SOLICITANTE, DEL APODERADO O DEL REPRESENTANTE LEGAL

Junto a cada firma, indicar el nombre del firmante y su calidad (si tal calidad no es obvia al leer el petitorio)



CAROLINA DAZA MONTALVO

10 (iv)	TABLA DE CORRESPONDENCIA DE REIVINDICACIONES	
Reivindicaciones en la solicitud en Colombia (NOTA. LAS REIVINDICACIONES ACÁ INDICADAS CORRESPONDEN A AQUELLAS MODIFICADAS EN MODIFICACIÓN VOLUNTARIA PRESENTADA EL 21 DE DICIEMBRE DEL 2021)	**Reivindicaciones otorgables en la Solicitud de Estados Unidos No. 16/642,748**	**Explicación sobre la suficiente correspondencia ó Comentarios que explican la correspondencia**
1. Un método de desbloqueo asimétrico (1300, 1400) para desbloquear un límite entre un bloque P de muestras y un bloque Q de muestras de modo tal que las muestras dentro del bloque P y las muestras dentro del bloque Q se modifiquen, el método comprende: determinar (1304, 1404) un valor refP según al menos p5, donde p5 es una muestra dentro del bloque P y hay otras cinco muestras (p0, p1, p2, p3 y p4) dentro del bloque P que separan p5 del límite; determinar (1306, 1406) un valor refQ según al menos qx, donde qx es una muestra dentro del bloque Q y i) hay otras tres muestras (q0, q1, y q2) dentro del bloque Q que separan qx del límite en el que qx=q3, en cuyo caso refQ es igual a (q2+qx +1)>>1,	1. Un método de desbloqueo asimétrico (1300, 1400) para desbloquear un límite entre un bloque P de muestras y un bloque Q de muestras de modo tal que las muestras dentro del bloque P y las muestras dentro del bloque Q se modifiquen, el método comprende: determinar (1304, 1404) un valor refP según al menos p5, donde p5 es una muestra dentro del bloque P y hay otras cinco muestras (p0, p1, p2, p3 y p4) dentro del bloque P que separan p5 del límite; determinar (1306, 1406) un valor refQ según al menos qx, donde qx es una muestra dentro del bloque Q y i) hay otras tres muestras (q0, q1, y q2) dentro del bloque Q que separan qx del límite en el que qx=q3, en cuyo caso refQ es igual a (q2+qx +1)>>1,	*Iguales*

o ii) hay otras siete muestras (q0, q1, q2, q3, q4, q5 y q6) dentro del bloque Q que separa qx del límite en el que qx=q7, en cuyo caso refQ es igual a: (q6+qx+1)>>1; determinar (1308, 1408) un valor refMiddle según al menos p0 y q0, donde p0 es directamente adyacente al límite y q0 es directamente adyacente al límite; donde i) en caso de que qx=q3, refMiddle es igual a (p3+p2+p1+p0+q0+q1+q2+q3+4) >> 3 o ii) en caso de que qx=7, refMiddle es igual a (p5+p4+p3+p2+2*(p1+p0+q0+q1)+q2+q3+q4+q5 + 8) >> 4); realizar (1310, 1410) una interpolación lineal entre refP y refMiddle; y realizar (1312, 1412) una interpolación lineal entre refQ y refMiddle.	o ii) hay otras siete muestras (q0, q1, q2, q3, q4, q5 y q6) dentro del bloque Q que separa qx del límite en el que qx=q7, en cuyo caso refQ es igual a: (q6+qx+1)>>1; determinar (1308, 1408) un valor refMiddle según al menos p0 y q0, donde p0 es directamente adyacente al límite y q0 es directamente adyacente al límite; donde i) en caso de que qx=q3, refMiddle es igual a (p3+p2+p1+p0+q0+q1+q2+q3+4) >> 3 o ii) en caso de que qx=7, refMiddle es igual a (p5+p4+p3+p2+2*(p1+p0+q0+q1)+q2+q3+q4+q5 + 8) >> 4); realizar (1310, 1410) una interpolación lineal entre refP y refMiddle; y realizar (1312, 1412) una interpolación lineal entre refQ y refMiddle.	
2. El método de la reivindicación 1, donde: el límite es un límite vertical, las muestras p0 a p5 están alineadas en una fila que es ortogonal al límite	2. El método de la reivindicación 1, donde: el límite es un límite vertical, las muestras p0 a p5 están alineadas en una fila que es ortogonal al límite	*Iguales*

vertical, y las muestras q0 a q3 o las muestras q0 a q6 están alineadas en la fila.	vertical, y las muestras q0 a q3 o las muestras q0 a q6 están alineadas en la fila.	
3. El método de la reivindicación 1, donde: el límite es un límite horizontal, las muestras p0 a p5 están alineadas en una columna que es ortogonal al límite horizontal, y las muestras q0 a q3 o las muestras q0 a q6 están alineadas en la columna.	3. El método de la reivindicación 1, donde: el límite es un límite horizontal, las muestras p0 a p5 están alineadas en una columna que es ortogonal al límite horizontal, y las muestras q0 a q3 o las muestras q0 a q6 están alineadas en la columna.	*Iguales*
4. El método de la reivindicación 1, donde realizar la interpolación lineal entre refP y refMiddle comprende calcular p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, donde x es de 0 a 4, p'(x) es una versión filtrada de px y f = {58,45,32,19,6}.	4. El método de la reivindicación 1, donde realizar la interpolación lineal entre refP y refMiddle comprende calcular p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, donde x es de 0 a 4, p'(x) es una versión filtrada de px y f = {58,45,32,19,6}.	*Iguales*
5. El método de la reivindicación 1, donde realizar la interpolación lineal entre refQ y refMiddle comprende calcular q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, donde x es de 0 a 6, q'(x) es una versión filtrada de qx, y g = {59,50,41,32,23,14,5}.	5. El método de la reivindicación 1, donde realizar la interpolación lineal entre refQ y refMiddle comprende calcular q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, donde x es de 0 a 6, q'(x) es una versión filtrada de qx, y g = {59,50,41,32,23,14,5}.	*Iguales*

6. El método de la reivindicación 1, donde realizar la interpolación lineal entre refP y refMiddle comprende calcular p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, donde x es de 0 a 4, p'(x) es una versión filtrada de px y f = {58,45,32,19,6}.	6. El método de la reivindicación 1, donde realizar la interpolación lineal entre refP y refMiddle comprende calcular p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, donde x es de 0 a 4, p'(x) es una versión filtrada de px y f = {58,45,32,19,6}.	*Iguales*
7. El método de la reivindicación 1, donde realizar la interpolación lineal entre refQ y refMiddle comprende calcular q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, donde x es de 0 a 2, q'(x) es una versión filtrada de q(x), y g = {53,32,11}.	7. El método de la reivindicación 1, donde realizar la interpolación lineal entre refQ y refMiddle comprende calcular q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, donde x es de 0 a 2, q'(x) es una versión filtrada de q(x), y g = {53,32,11}.	*Iguales*
8. El método de la reivindicación 1, donde refP es igual a: (p5+p4+1) >> 1.	8. El método de la reivindicación 1, donde refP es igual a: (p5+p4+1) >> 1.	*Iguales*
9. El método de la reivindicación 1, donde: el bloque P está arriba o a la izquierda del bloque Q, o el bloque P está debajo o a la derecha del bloque Q.	9. El método de la reivindicación 1, donde: el bloque P está arriba o a la izquierda del bloque Q, o el bloque P está debajo o a la derecha del bloque Q.	*Iguales*

10. El método de la reivindicación 1, donde: el bloque P tiene un tamaño en una dirección ortogonal al límite de 32 muestras; el bloque P usa subbloques de predicción.	10. El método de la reivindicación 1, donde: el bloque P tiene un tamaño en una dirección ortogonal al límite de 32 muestras; el bloque P usa subbloques de predicción.	*Iguales*
11. El método de la reivindicación 10, donde: el bloque Q tiene un tamaño en una dirección ortogonal al límite de menos de 32 muestras.	11. El método de la reivindicación 10, donde: el bloque Q tiene un tamaño en una dirección ortogonal al límite de menos de 32 muestras.	*Iguales*
12. El método de la reivindicación 10, donde: el bloque Q tiene un tamaño en una dirección ortogonal al límite igual o superior a 32 muestras; y el bloque Q no utiliza subbloques.	12. El método de la reivindicación 10, donde: el bloque Q tiene un tamaño en una dirección ortogonal al límite igual o superior a 32 muestras; y el bloque Q no utiliza subbloques.	*Iguales*
13. El método de la reivindicación 1, donde: el bloque P tiene un tamaño en una dirección ortogonal al límite de 32 muestras; y el límite es un límite de unidad de árbol de codificación horizontal (CTU).	13. El método de la reivindicación 1, donde: el bloque P tiene un tamaño en una dirección ortogonal al límite de 32 muestras; y el límite es un límite de unidad de árbol de codificación horizontal (CTU).	*Iguales*
14. El método de la reivindicación 4, que además comprende: para x de 0 a 4, determinando	14. El método de la reivindicación 4, que además comprende: para x de 0 a 4, determinando	*Iguales*

p''(x) = clip3(p'(x) + tcP$_x$, p'(x) – tcP$_x$, p'(x)), donde tcP$_x$ es un umbral de recorte y clip3(x, y, z) = $\begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{de lo contrario} \end{cases}$.	p''(x) = clip3(p'(x) + tcP$_x$, p'(x) – tcP$_x$, p'(x)), donde tcP$_x$ es un umbral de recorte y clip3(x, y, z) = $\begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{de lo contrario} \end{cases}$.	
15. El método de la reivindicación 5, que además comprende: para x de 0 a 2, determinando q''(x) = clip3(q'(x) + tcQ$_x$, q'(x) – tcQ$_x$, q'(x)), donde tcQ$_x$ es un umbral de recorte y clip3(x, y, z) = $\begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{de lo contrario} \end{cases}$.	15. El método de la reivindicación 5, que además comprende: para x de 0 a 2, determinando q''(x) = clip3(q'(x) + tcQ$_x$, q'(x) – tcQ$_x$, q'(x)), donde tcQ$_x$ es un umbral de recorte y clip3(x, y, z) = $\begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{de lo contrario} \end{cases}$.	*Iguales*
16. El método de la reivindicación 14, donde el umbral de recorte es un umbral de recorte dependiente del parámetro de cuantificación (QP).	16. El método de la reivindicación 14, donde el umbral de recorte es un umbral de recorte dependiente del parámetro de cuantificación (QP).	*Iguales*
17. Un medio legible por computadora no transitorio que tiene instrucciones (1544) almacenadas en el, las cuales cuando se ejecutan mediante circuitería de procesamiento (1502), hacen que la circuitería de procesamiento (1502) lleve a cabo el método de la reivindicación 1.	17. Un producto de programa informático que comprende un medio legible por ordenador no transitorio que almacena un programa informático que comprende instrucciones que, cuando se ejecutan mediante circuitos de procesamiento, hacen que los circuitos de procesamiento realicen el método de la	*En la modificación voluntaria presentada el 21 de diciembre de 2021, se modificó esta reivindicación por tratarse de un producto de programa informático, lo cual es materia no patentable.*

	reivindicación 1.	
18. Un aparato (1501) para desbloquear un límite entre un bloque P de muestras y un bloque Q de muestras de modo tal que las muestras dentro del bloque P y las muestras dentro del bloque Q se modifiquen, el aparato comprende: circuitería de procesamiento; y una memoria, conteniendo dicha memoria instrucciones ejecutables por dicha circuitería de procesamiento, por lo que dicho aparato es operativo para: determinar un valor refP basado en al menos p5, donde p5 es una muestra dentro del bloque P y hay otras cinco muestras (p0, p1, p2, p3 y p4) dentro del bloque P que separan p5 del límite; determinar un valor refQ basado en al menos qx, donde qx es una muestra dentro del bloque Q y i) hay otras tres muestras (q0, q1 y q2) dentro del bloque Q que separan qx del	18. Un aparato (1501) para desbloquear un límite entre un bloque P de muestras y un bloque Q de muestras de modo tal que las muestras dentro del bloque P y las muestras dentro del bloque Q se modifiquen, el aparato comprende: circuitería de procesamiento; y una memoria, conteniendo dicha memoria instrucciones ejecutables por dicha circuitería de procesamiento, por lo que dicho aparato es operativo para: determinar un valor refP basado en al menos p5, donde p5 es una muestra dentro del bloque P y hay otras cinco muestras (p0, p1, p2, p3 y p4) dentro del bloque P que separan p5 del límite; determinar un valor refQ basado en al menos qx, donde qx es una muestra dentro del bloque Q y i) hay otras tres muestras (q0, q1 y q2) dentro del bloque Q que separan qx del	*Iguales*

límite de tal manera que qx = q3, en cuyo caso refQ es igual a (q2 + qx +1) >> 1, o ii) hay otras siete muestras (q0, q1, q2, q3, q4, q5 y q6) dentro del bloque Q que separan qx desde el límite tal que qx = q7, en cuyo caso refQ es igual a: (q6 + qx + 1) >> 1; determinar un valor refMiddle basado en al menos p0 y q0, donde p0 es directamente adyacente al límite y q0 es directamente adyacente al límite, donde i) en el caso de que qx = q3, refMiddle es igual a (p3 + p2 + p1 + p0 + q0 + q1 + q2 + q3 + 4) >> 3 o ii) en el caso de que qx = 7, refMiddle es igual a (p5 + p4 + p3 + p2 + 2 * (p1 + p0 + q0 + q1) + q2 + q3 + q4 + q5 + 8) >> 4); realizar una interpolación lineal entre refP y refMiddle, modificando así las muestras dentro del bloque P; y realizar una interpolación lineal entre refQ y refMiddle, modificando así las muestras dentro del bloque Q.	límite de tal manera que qx = q3, en cuyo caso refQ es igual a (q2 + qx +1) >> 1, o ii) hay otras siete muestras (q0, q1, q2, q3, q4, q5 y q6) dentro del bloque Q que separan qx desde el límite tal que qx = q7, en cuyo caso refQ es igual a: (q6 + qx + 1) >> 1; determinar un valor refMiddle basado en al menos p0 y q0, donde p0 es directamente adyacente al límite y q0 es directamente adyacente al límite, donde i) en el caso de que qx = q3, refMiddle es igual a (p3 + p2 + p1 + p0 + q0 + q1 + q2 + q3 + 4) >> 3 o ii) en el caso de que qx = 7, refMiddle es igual a (p5 + p4 + p3 + p2 + 2 * (p1 + p0 + q0 + q1) + q2 + q3 + q4 + q5 + 8) >> 4); realizar una interpolación lineal entre refP y refMiddle, modificando así las muestras dentro del bloque P; y realizar una interpolación lineal entre refQ y refMiddle, modificando así las muestras dentro del bloque Q.	

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

NOTICE OF ALLOWANCE AND FEE(S) DUE

6449 7590 07/28/2021

ROTHWELL, FIGG, ERNST & MANBECK, P.C.
607 14TH STREET, N.W.
SUITE 800
WASHINGTON, DC 20005

EXAMINER
LUO, KATE H

ART UNIT	PAPER NUMBER
2488	

DATE MAILED: 07/28/2021

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
16/642,748	02/27/2020	Kenneth ANDERSSON	3602-1728US2	4151

TITLE OF INVENTION: ASYMMETRIC DEBLOCKING IN A VIDEO ENCODER AND/OR VIDEO DECODER

APPLN. TYPE	ENTITY STATUS	ISSUE FEE DUE	PUBLICATION FEE DUE	PREV. PAID ISSUE FEE	TOTAL FEE(S) DUE	DATE DUE
nonprovisional	UNDISCOUNTED	$1200	$0.00	$0.00	$1200	10/28/2021

THE APPLICATION IDENTIFIED ABOVE HAS BEEN EXAMINED AND IS ALLOWED FOR ISSUANCE AS A PATENT. PROSECUTION ON THE MERITS IS CLOSED. THIS NOTICE OF ALLOWANCE IS NOT A GRANT OF PATENT RIGHTS. THIS APPLICATION IS SUBJECT TO WITHDRAWAL FROM ISSUE AT THE INITIATIVE OF THE OFFICE OR UPON PETITION BY THE APPLICANT. SEE 37 CFR 1.313 AND MPEP 1308.

THE ISSUE FEE AND PUBLICATION FEE (IF REQUIRED) MUST BE PAID WITHIN THREE MONTHS FROM THE MAILING DATE OF THIS NOTICE OR THIS APPLICATION SHALL BE REGARDED AS ABANDONED. THIS STATUTORY PERIOD CANNOT BE EXTENDED. SEE 35 U.S.C. 151. THE ISSUE FEE DUE INDICATED ABOVE DOES NOT REFLECT A CREDIT FOR ANY PREVIOUSLY PAID ISSUE FEE IN THIS APPLICATION. IF AN ISSUE FEE HAS PREVIOUSLY BEEN PAID IN THIS APPLICATION (AS SHOWN ABOVE), THE RETURN OF PART B OF THIS FORM WILL BE CONSIDERED A REQUEST TO REAPPLY THE PREVIOUSLY PAID ISSUE FEE TOWARD THE ISSUE FEE NOW DUE.

HOW TO REPLY TO THIS NOTICE:

I. Review the ENTITY STATUS shown above. If the ENTITY STATUS is shown as SMALL or MICRO, verify whether entitlement to that entity status still applies.

If the ENTITY STATUS is the same as shown above, pay the TOTAL FEE(S) DUE shown above.

If the ENTITY STATUS is changed from that shown above, on PART B - FEE(S) TRANSMITTAL, complete section number 5 titled "Change in Entity Status (from status indicated above)".

For purposes of this notice, small entity fees are 1/2 the amount of undiscounted fees, and micro entity fees are 1/2 the amount of small entity fees.

II. PART B - FEE(S) TRANSMITTAL, or its equivalent, must be completed and returned to the United States Patent and Trademark Office (USPTO) with your ISSUE FEE and PUBLICATION FEE (if required). If you are charging the fee(s) to your deposit account, section "4b" of Part B - Fee(s) Transmittal should be completed and an extra copy of the form should be submitted. If an equivalent of Part B is filed, a request to reapply a previously paid issue fee must be clearly made, and delays in processing may occur due to the difficulty in recognizing the paper as an equivalent of Part B.

III. All communications regarding this application must give the application number. Please direct all communications prior to issuance to Mail Stop ISSUE FEE unless advised to the contrary.

IMPORTANT REMINDER: Maintenance fees are due in utility patents issuing on applications filed on or after Dec. 12, 1980. It is patentee's responsibility to ensure timely payment of maintenance fees when due. More information is available at www.uspto.gov/PatentMaintenanceFees.

PART B - FEE(S) TRANSMITTAL

Complete and send this form, together with applicable fee(s), by mail or fax, or via EFS-Web.

By mail, send to: Mail Stop ISSUE FEE
Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450

By fax, send to: (571)-273-2885

INSTRUCTIONS: This form should be used for transmitting the ISSUE FEE and PUBLICATION FEE (if required). Blocks 1 through 5 should be completed where appropriate. All further correspondence including the Patent, advance orders and notification of maintenance fees will be mailed to the current correspondence address as indicated unless corrected below or directed otherwise in Block 1, by (a) specifying a new correspondence address; and/or (b) indicating a separate "FEE ADDRESS" for maintenance fee notifications.

CURRENT CORRESPONDENCE ADDRESS (Note: Use Block 1 for any change of address)

6449 7590 07/28/2021

ROTHWELL, FIGG, ERNST & MANBECK, P.C.
607 14TH STREET, N.W.
SUITE 800
WASHINGTON, DC 20005

Note: A certificate of mailing can only be used for domestic mailings of the Fee(s) Transmittal. This certificate cannot be used for any other accompanying papers. Each additional paper, such as an assignment or formal drawing, must have its own certificate of mailing or transmission.

Certificate of Mailing or Transmission

I hereby certify that this Fee(s) Transmittal is being deposited with the United States Postal Service with sufficient postage for first class mail in an envelope addressed to the Mail Stop ISSUE FEE address above, or being transmitted to the USPTO via EFS-Web or by facsimile to (571) 273-2885, on the date below.

(Typed or printed name)
(Signature)
(Date)

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
16/642,748	02/27/2020	Kenneth ANDERSSON	3602-1728US2	4151

TITLE OF INVENTION: ASYMMETRIC DEBLOCKING IN A VIDEO ENCODER AND/OR VIDEO DECODER

APPLN. TYPE	ENTITY STATUS	ISSUE FEE DUE	PUBLICATION FEE DUE	PREV. PAID ISSUE FEE	TOTAL FEE(S) DUE	DATE DUE
nonprovisional	UNDISCOUNTED	$1200	$0.00	$0.00	$1200	10/28/2021

EXAMINER	ART UNIT	CLASS-SUBCLASS
LUO, KATE H	2488	375-240240

1. Change of correspondence address or indication of "Fee Address" (37 CFR 1.363).

❑ Change of correspondence address (or Change of Correspondence Address form PTO/SB/122) attached.

❑ "Fee Address" indication (or "Fee Address" Indication form PTO/SB/47; Rev 03-09 or more recent) attached. **Use of a Customer Number is required.**

2. For printing on the patent front page, list
(1) The names of up to 3 registered patent attorneys or agents OR, alternatively,
(2) The name of a single firm (having as a member a registered attorney or agent) and the names of up to 2 registered patent attorneys or agents. If no name is listed, no name will be printed.

1 _____
2 _____
3 _____

3. ASSIGNEE NAME AND RESIDENCE DATA TO BE PRINTED ON THE PATENT (print or type)

PLEASE NOTE: Unless an assignee is identified below, no assignee data will appear on the patent. If an assignee is identified below, the document must have been previously recorded, or filed for recordation, as set forth in 37 CFR 3.11 and 37 CFR 3.81(a). Completion of this form is NOT a substitute for filing an assignment.

(A) NAME OF ASSIGNEE (B) RESIDENCE: (CITY and STATE OR COUNTRY)

Please check the appropriate assignee category or categories (will not be printed on the patent) : ❑ Individual ❑ Corporation or other private group entity ❑ Government

4a. Fees submitted: ❑ Issue Fee ❑ Publication Fee (if required) ❑ Advance Order - # of Copies _____

4b. Method of Payment: *(Please first reapply any previously paid fee shown above)*

❑ Electronic Payment via EFS-Web ❑ Enclosed check ❑ Non-electronic payment by credit card (Attach form PTO-2038)

❑ The Director is hereby authorized to charge the required fee(s), any deficiency, or credit any overpayment to Deposit Account No. _____

5. **Change in Entity Status** (from status indicated above)

❑ Applicant certifying micro entity status. See 37 CFR 1.29

❑ Applicant asserting small entity status. See 37 CFR 1.27

❑ Applicant changing to regular undiscounted fee status.

NOTE: Absent a valid certification of Micro Entity Status (see forms PTO/SB/15A and 15B), issue fee payment in the micro entity amount will not be accepted at the risk of application abandonment.
NOTE: If the application was previously under micro entity status, checking this box will be taken to be a notification of loss of entitlement to micro entity status.
NOTE: Checking this box will be taken to be a notification of loss of entitlement to small or micro entity status, as applicable.

NOTE: This form must be signed in accordance with 37 CFR 1.31 and 1.33. See 37 CFR 1.4 for signature requirements and certifications.

Authorized Signature _____ Date _____

Typed or printed name _____ Registration No. _____

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
16/642,748	02/27/2020	Kenneth ANDERSSON	3602-1728US2	4151

6449 7590 07/28/2021

ROTHWELL, FIGG, ERNST & MANBECK, P.C.
607 14TH STREET, N.W.
SUITE 800
WASHINGTON, DC 20005

EXAMINER	
LUO, KATE H	

ART UNIT	PAPER NUMBER
2488	

DATE MAILED: 07/28/2021

Determination of Patent Term Adjustment under 35 U.S.C. 154 (b)
(Applications filed on or after May 29, 2000)

The Office has discontinued providing a Patent Term Adjustment (PTA) calculation with the Notice of Allowance.

Section 1(h)(2) of the AIA Technical Corrections Act amended 35 U.S.C. 154(b)(3)(B)(i) to eliminate the requirement that the Office provide a patent term adjustment determination with the notice of allowance. See Revisions to Patent Term Adjustment, 78 Fed. Reg. 19416, 19417 (Apr. 1, 2013). Therefore, the Office is no longer providing an initial patent term adjustment determination with the notice of allowance. The Office will continue to provide a patent term adjustment determination with the Issue Notification Letter that is mailed to applicant approximately three weeks prior to the issue date of the patent, and will include the patent term adjustment on the patent. Any request for reconsideration of the patent term adjustment determination (or reinstatement of patent term adjustment) should follow the process outlined in 37 CFR 1.705.

Any questions regarding the Patent Term Extension or Adjustment determination should be directed to the Office of Patent Legal Administration at (571)-272-7702. Questions relating to issue and publication fee payments should be directed to the Customer Service Center of the Office of Patent Publication at 1-(888)-786-0101 or (571)-272-4200.

OMB Clearance and PRA Burden Statement for PTOL-85 Part B

The Paperwork Reduction Act (PRA) of 1995 requires Federal agencies to obtain Office of Management and Budget approval before requesting most types of information from the public. When OMB approves an agency request to collect information from the public, OMB (i) provides a valid OMB Control Number and expiration date for the agency to display on the instrument that will be used to collect the information and (ii) requires the agency to inform the public about the OMB Control Number's legal significance in accordance with 5 CFR 1320.5(b).

The information collected by PTOL-85 Part B is required by 37 CFR 1.311. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 30 minutes to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, Virginia 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, Virginia 22313-1450. Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

Privacy Act Statement

The Privacy Act of 1974 (P.L. 93-579) requires that you be given certain information in connection with your submission of the attached form related to a patent application or patent. Accordingly, pursuant to the requirements of the Act, please be advised that: (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether disclosure of these records is required by the Freedom of Information Act.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, to another federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspection or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

Notice of Allowability	Application No. 16/642,748	Applicant(s) ANDERSSON et al.	
	Examiner KATE H LUO	Art Unit 2488	AIA (FITF) Status Yes

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address--

All claims being allowable, PROSECUTION ON THE MERITS IS (OR REMAINS) CLOSED in this application. If not included herewith (or previously mailed), a Notice of Allowance (PTOL-85) or other appropriate communication will be mailed in due course. **THIS NOTICE OF ALLOWABILITY IS NOT A GRANT OF PATENT RIGHTS.** This application is subject to withdrawal from issue at the initiative of the Office or upon petition by the applicant. See 37 CFR 1.313 and MPEP 1308.

1. ☑ This communication is responsive to 7/2/2021.
   ☐ A declaration(s)/affidavit(s) under **37 CFR 1.130(b)** was/were filed on _____.

2. ☐ An election was made by the applicant in response to a restriction requirement set forth during the interview on _____; the restriction requirement and election have been incorporated into this action.

3. ☑ The allowed claim(s) is/are 1-3,5-6,8-19 and 22 . As a result of the allowed claim(s), you may be eligible to benefit from the **Patent Prosecution Highway** program at a participating intellectual property office for the corresponding application. For more information , please see **http://www.uspto.gov/patents/init_events/pph/index.jsp** or send an inquiry to **PPHfeedback@uspto.gov.**

4. ☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

   Certified copies:

   a) ☐ All b) ☐ Some *c) ☐ None of the:

   1. ☐ Certified copies of the priority documents have been received.
   2. ☐ Certified copies of the priority documents have been received in Application No. _____ .
   3. ☐ Copies of the certified copies of the priority documents have been received in this national stage application from the International Bureau (PCT Rule 17.2(a)).

   * Certified copies not received: _____ .

Applicant has THREE MONTHS FROM THE "MAILING DATE" of this communication to file a reply complying with the requirements noted below. Failure to timely comply will result in ABANDONMENT of this application.
THIS THREE-MONTH PERIOD IS NOT EXTENDABLE.

5. ☐ CORRECTED DRAWINGS (as "replacement sheets") must be submitted.
   ☐ including changes required by the attached Examiner's Amendment / Comment or in the Office action of Paper No./Mail Date _____ .

   Identifying indicia such as the application number (see 37 CFR 1.84(c)) should be written on the drawings in the front (not the back) of each sheet. Replacement sheet(s) should be labeled as such in the header according to 37 CFR 1.121(d).

6. ☐ DEPOSIT OF and/or INFORMATION about the deposit of BIOLOGICAL MATERIAL must be submitted. Note the attached Examiner's comment regarding REQUIREMENT FOR THE DEPOSIT OF BIOLOGICAL MATERIAL.

Attachment(s)

1. ☐ Notice of References Cited (PTO-892)
2. ☐ Information Disclosure Statements (PTO/SB/08), Paper No./Mail Date _____.
3. ☐ Examiner's Comment Regarding Requirement for Deposit of Biological Material _____.
4. ☑ Interview Summary (PTO-413), Paper No./Mail Date. _____.
5. ☑ Examiner's Amendment/Comment
6. ☑ Examiner's Statement of Reasons for Allowance
7. ☐ Other _____.

/KATE H LUO/
Primary Examiner, Art Unit 2488

EXAMINER'S AMENDMENT

1. An examiner's amendment to the record appears below. Should the changes and/or additions be unacceptable to applicant, an amendment may be filed as provided by 37 CFR 1.312. To ensure consideration of such an amendment, it MUST be submitted no later than the payment of the issue fee.

Authorization for this examiner's amendment was given in a telephone interview with Mr. Brain Rosenbloom (Reg. No. 41,276) on July 20, 2021.

The application has been amended as follows:

2. Claim 8 depends on claim 2 and is amended as follows.

8. (Amended) The method of claim 1, wherein performing the linear interpolation between refP and refMiddle comprises calculating ~~calculating~~ p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, where x is 0 to 4, p'(x) is a filtered version of px, and f = {58,45,32,19,6}.

REASON FOR ALLOWANCE

1. The following is an examiner's statement of reasons for allowance: The instant invention is related to a method and an apparatus for deblocking a boundary between block samples.

2. Prior art was found and applied in the previous actions. In an amendment dated on 7/2/2021, the claims were amended to include subject matter that had been previously indicated as allowable. In consideration of the applicant's newly amended claims as whole, there is not

strong motivation or reasoning to combine references to arrive at the claimed invention. Claims 1-3, 5, 6, 8-19 and 22 are allowed.

3. Any comments considered necessary by applicant must be submitted no later than the payment of the issue fee and, to avoid processing delays, should preferably accompany the issue fee. Such submissions should be clearly labeled "Comments on Statement of Reasons for Allowance."

Conclusion

Any inquiry concerning this communication or earlier communications from the examiner should be directed to KATE H LUO whose telephone number is (571)270-5635. The examiner can normally be reached on 8:00-5:00PM.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Sath V. Perungavoor can be reached on (571)272-7455455. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free). If you would like assistance from a USPTO Customer Service Representative or access to the automated information system, call 800-786-9199 (IN USA OR CANADA) or 571-272-1000.

/KATE H LUO/
Primary Examiner, Art Unit 2488

Examiner-Initiated Interview Summary	**Application No.** 16/642,748	**Applicant(s)** ANDERSSON et al.		
	Examiner KATE H LUO	**Art Unit** 2488	**AIA (First Inventor to File) Status** Yes	**Page** **1 of 1**

All Participants (applicant, applicants representative, PTO personnel)	**Title**	**Type**
KATE H LUO	Primary Examiner	Telephonic
Mr. Brain Rosenbloom (Reg. No. 41,276)	Attorney of Record	

Date of Interview: 20 July 2021

Issues Discussed:

Proposed Amendment(s)

1. Examiner explained claim 8 is objected as stated in Non-final rejection of 4/2/2021. Applicant agreed to amend it to depend on claim 2.

/KATE H LUO/ Primary Examiner, Art Unit 2488	

Applicant is reminded that a complete written statement as to the substance of the interview must be made of record in the application file. It is the applicants responsibility to provide the written statement, unless the interview was initiated by the Examiner and the Examiner has indicated that a written summary will be provided. See MPEP 713.04
Please further see:
MPEP 713.04
Title 37 Code of Federal Regulations (CFR) § 1.133 Interviews, paragraph (b)
37 CFR § 1.2 Business to be transacted in writing

Applicant recordation instructions: It is not necessary for applicant to provide a separate record of the substance of interview.

Examiner recordation instructions: Examiners must summarize the substance of any interview of record. A complete and proper recordation of the substance of an interview should include the items listed in MPEP 713.04 for complete and proper recordation including the identification of the general thrust of each argument or issue discussed, a general indication of any other pertinent matters discussed regarding patentability and the general results or outcome of the interview, to include an indication as to whether or not agreement was reached on the issues raised.

Search Notes	Application/Control No.	Applicant(s)/Patent Under Reexamination
	16/642,748	ANDERSSON et al.
	Examiner	Art Unit
	KATE H LUO	2488

CPC - Searched*		
Symbol	Date	Examiner

CPC Combination Sets - Searched*		
Symbol	Date	Examiner

US Classification - Searched*			
Class	Subclass	Date	Examiner

* See search history printout included with this form or the SEARCH NOTES box below to determine the scope of the search.

Search Notes		
Search Notes	Date	Examiner
searched inventors	03/11/2021	KL
Searched USPAT, US-PGPUB, EPO, and JPO	03/11/2021	KL
update searches based on amendment	07/20/2021	KL

Interference Search			
US Class/CPC Symbol	US Subclass/CPC Group	Date	Examiner
Interference claim text search		07/20/2021	KL

/KATE H LUO/ Primary Examiner, Art Unit 2488	

Issue Classification	Application/Control No.	Applicant(s)/Patent Under Reexamination
	16/642,748	ANDERSSON et al.
	Examiner	**Art Unit**
	KATE H LUO	2488

CPC						
Symbol					**Type**	**Version**
H04N	/	19	/	865	F	2014-11-01
H04N	/	19	/	176	I	2014-11-01
H04N	/	19	/	86	I	2014-11-01
H04N	/	19	/	96	I	2014-11-01

CPC Combination Sets							
Symbol					**Type**	**Set**	**Ranking**	**Version**
	/		/					

NONE		Total Claims Allowed:	
(Assistant Examiner)	(Date)	19	
/KATE H LUO/ Primary Examiner, Art Unit 2488	20 July 2021	O.G. Print Claim(s)	O.G. Print Figure
(Primary Examiner)	(Date)	1	13

U.S. Patent and Trademark Office

Part of Paper No.: 20210720

Issue Classification	Application/Control No.	Applicant(s)/Patent Under Reexamination
	16/642,748	ANDERSSON et al.
	Examiner	**Art Unit**
	KATE H LUO	2488

INTERNATIONAL CLASSIFICATION				
CLAIMED				
H04N19/86	/	19	/	86
H04N19/176	/	19	/	176
H04N19/96	/	19	/	96

NON-CLAIMED				
	/		/	

US ORIGINAL CLASSIFICATION	
CLASS	**SUBCLASS**

CROSS REFERENCES(S)						
CLASS	**SUBCLASS (ONE SUBCLASS PER BLOCK)**					

NONE (Assistant Examiner) (Date)	**Total Claims Allowed:** 19	
/KATE H LUO/ Primary Examiner, Art Unit 2488 (Primary Examiner) 20 July 2021 (Date)	O.G. Print Claim(s) 1	O.G. Print Figure 13

Issue Classification	Application/Control No.	Applicant(s)/Patent Under Reexamination
	16/642,748	ANDERSSON et al.
	Examiner	**Art Unit**
	KATE H LUO	2488

☐ Claims renumbered in the same order as presented by applicant ☐ CPA ☐ T.D. ☐ R.1.47

CLAIMS

Final	Original	Final	Original	Final	Original	Final	Original	Final	Original	Final	Original	Final	Original	Final	Original
1	1	12	10	18	19										
2	2	13	11	-	20										
4	3	14	12	-	21										
-	4	15	13	19	22										
5	5	16	14												
8	6	17	15												
-	7	6	16												
10	8	9	17												
11	9	7	18												

NONE (Assistant Examiner)	(Date)	**Total Claims Allowed:** 19	
/KATE H LUO/ Primary Examiner, Art Unit 2488 (Primary Examiner)	20 July 2021 (Date)	O.G. Print Claim(s) 1	O.G. Print Figure 13

Index of Claims	Application/Control No.	Applicant(s)/Patent Under Reexamination
	16/642,748	ANDERSSON et al.
	Examiner	Art Unit
	KATE H LUO	2488

✓	Rejected	-	Cancelled	N	Non-Elected	A	Appeal
=	Allowed	÷	Restricted	I	Interference	O	Objected

CLAIMS

☐ Claims renumbered in the same order as presented by applicant ☐ CPA ☐ T.D. ☐ R.1.47

CLAIM		DATE								
Final	Original	03/11/2021	07/20/2021							
	1	✓	=							
	2	✓	=							
	3	✓	=							
	4	O	-							
	5	✓	=							
	6	✓	=							
	7	O	-							
	8	✓	=							
	9	✓	=							
	10	✓	=							
	11	✓	=							
	12	✓	=							
	13	✓	=							
	14	✓	=							
	15	✓	=							
	16	O	=							
	17	O	=							
	18	O	=							
	19	✓	=							
	20	✓	-							
	21	✓	-							
	22	✓	=							

EAST Search History

EAST Search History (Prior Art)

Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
L2	56	deblock$3 same (boundary with ("near" or close or adjacent or neighbor$3)) same shift$3	US-PGPUB; USPAT	OR	ON	2021/07/20 16:50
L5	0	deblock$3 same (boundary with ("near" or close or adjacent or neighbor$3)) same shift$3 same ((three or seven) with sample)	US-PGPUB; USPAT	OR	ON	2021/07/20 16:51
L6	19	deblock$3 same (boundary with ("near" or close or adjacent or neighbor$3)) same shift$3 same sample	US-PGPUB; USPAT	OR	ON	2021/07/20 16:51
L8	457	interpolation same ((boundary or edge) with ("near" or close or adjacent or neighbor$3)) same sample	US-PGPUB; USPAT	OR	ON	2021/07/20 16:52
L10	10	interpolation same ((boundary or edge) with ("near" or close or adjacent or neighbor$3)) same sample same middle	US-PGPUB; USPAT	OR	ON	2021/07/20 16:53
S1	0	"16642748"	US-PGPUB; USPAT	OR	ON	2021/03/09 12:01
S2	10,674	((("TELEFONAKTIEBOLAGET") near3 ("LM") near3 ("ERICSSON") near3 ("(PUBL)"))).AS,AANM.	USPAT	OR	ON	2021/03/09 12:03
S3	881	((("ANDERSSON") near3 ("Kenneth")) OR (("ENHORN") near3 ("Jack")) OR (("ZHANG") near3 ("Zhi"))).INV.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT	OR	ON	2021/03/09 12:05
S4	10,674	((("TELEFONAKTIEBOLAGET") near3 ("LM") near3 ("ERICSSON") near3 ("(PUBL)"))).AS,AANM.	USPAT	OR	ON	2021/03/10 09:30
S5	881	((("ANDERSSON") near3 ("Kenneth")) OR (("ENHORN") near3 ("Jack")) OR (("ZHANG") near3 ("Zhi"))).INV.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT	OR	ON	2021/03/10 09:30
S6	93	(S4 or S5) and deblock$3	US-PGPUB; USPAT	OR	ON	2021/03/10 09:30
S7	1	(S4 or S5) and ((deblock$3 with boundary) same interpolation)	US-PGPUB; USPAT	OR	ON	2021/03/10 09:31
S8	151	((deblock$3 with boundary) same interpolation)	US-PGPUB; USPAT	OR	ON	2021/03/10 09:42

S9	109	S8 and @ad<"20181108"	US-PGPUB; USPAT	OR	ON	2021/03/10 09:42
S10	107	S8 and @ad<"20181108" and H04N19$.cpc.	US-PGPUB; USPAT	OR	ON	2021/03/10 09:42
S11	24,895	(H04N19/176 or H04N19/865 or H04N19/96).cpc.	US-PGPUB; USPAT	OR	ON	2021/03/10 09:47
S12	100	(US-20150146795-$ or US-20180160127-$ or US-20180376143-$ or US-20160261864-$ or US-20150249842-$ or US-20170272749-$ or US-20180160126-$ or US-20190281294-$ or US-20180288381-$ or US-20160105657-$ or US-20170257644-$ or US-20190246123-$ or US-20070216763-$ or US-20190206360-$ or US-20180332286-$ or US-20170251209-$ or US-20180278934-$ or US-20180077414-$ or US-20060104349-$ or US-20170302920-$ or US-20160142739-$ or US-20080123750-$ or US-10432970-$ or US-20170302967-$ or US-20160212438-$ or US-20180054628-$ or US-20150208096-$ or US-20110194602-$ or US-20110194614-$ or US-20150172670-$ or US-20190089988-$ or US-20190208198-$ or US-20130287124-$ or US-20170238011-$ or US-20160219301-$ or US-20190037213-$ or US-20150229934-$ or US-20180167615-$ or US-20130003865-$ or US-20160234517-$ or US-20190191172-$ or US-20110194608-$ or US-20160234495-$ or US-20170111645-$ or US-20110026599-$ or US-20140079115-$ or US-20190068969-$ or US-20170366818-$ or US-20130051480-$ or US-20170332098-$ or US-20170295373-$ or US-20160241885-$ or US-20170208310-$ or US-20150071340-$ or US-20160360235-$ or US-20140050272-$ or US-20040005004-$ or US-10440416-$ or US-20150334407-$ or US-20130294525-$ or US-20110194609-$ or US-20160301930-$ or US-20140205015-$ or US-20140211841-$ or US-20110033127-$ or US-20130128979-$ or US-20130329814-$ or US-20070140554-$ or US-20110249728-$ or US-20150085927-$ or US-20150365671-$ or US-20150103902-$ or US-20120039389-$ or US-20120170858-$ or US-20020196849-$ or US-20140294068-	US-PGPUB; USPAT	OR	ON	2021/03/10 09:48

		$ or US-20110007801-$ or US-20200068223-$ or US-20140177718-$ or US-20140079135-$ or US-20180035131-$ or US-20150016505-$ or US-20160219306-$ or US-20150172677-$ or US-20100296745-$ or US-20090257492-$ or US-20110255603-$ or US-20160353115-$ or US-20170085917-$ or US-20170318298-$ or US-20190116361-$ or US-9918095-$ or US-20190238866-$ or US-20170359595-$ or US-20170085896-$ or US-20150281706-$ or US-20200162761-$ or US-20170272781-$ or US-20180213262-$ or US-20200244999-$).did.				
S13	1	S12 and @ad<"20181108" and S8	US-PGPUB; USPAT	OR	ON	2021/03/10 09:48
S14	61	(andrey near2 Norkin).in.	US-PGPUB; USPAT	OR	ON	2021/03/10 09:49
S15	34	(anand near2 meher near2 Kotra).in.	US-PGPUB; USPAT	OR	ON	2021/03/10 09:50
S16	50	(S14 or S15) and (deblock$3)	US-PGPUB; USPAT	OR	ON	2021/03/10 09:50
S17	0	(S14 or S15) and (deblock$3 same interpolation)	US-PGPUB; USPAT	OR	ON	2021/03/10 09:51
S18	36	(S14 or S15) and (deblock$3) and @ad<"20181108"	US-PGPUB; USPAT	OR	ON	2021/03/10 09:51
S19	32	(S14 or S15) and (deblock$3 same boundary) and @ad<"20181108"	US-PGPUB; USPAT	OR	ON	2021/03/10 09:52
S20	0	(S14 or S15) and (deblock$3 same boundary same interpolation) and @ad<"20181108"	US-PGPUB; USPAT	OR	ON	2021/03/10 09:57
S21	12	deblock$3 same interpolation same (boundary with ("near" or close or adjacent or neighbor$3))	US-PGPUB; USPAT	OR	ON	2021/03/10 10:03
S22	21,917	S11 and @ad<"20181108" and H04N19$.cpc.	US-PGPUB; USPAT	OR	ON	2021/03/10 10:03
S23	10	S21 and @ad<"20181108" and H04N19$.cpc.	US-PGPUB; USPAT	OR	ON	2021/03/10 10:03
S24	106	(US-20040101059-$ or US-20040062307-$ or US-20030053708-$ or US-20060126962-$ or US-20030053711-$ or US-20040076333-$ or US-20060268991-$ or US-20030152146-$ or US-6748113-$ or US-20030219073-$ or US-20030169931-$ or US-20030053541-$ or US-20030112864-$ or US-20080089417-	US-PGPUB; USPAT	OR	ON	2021/03/10 10:11

		$ or US-20060294171-$ or US-6539060-$ or US-6041145-$ or US-20120177107-$ or US-20040062310-$ or US-20030044080-$ or US-20040076237-$ or US-20080247467-$ or US-6240135-$ or US-20020136303-$ or US-20030081854-$ or US-20120294353-$ or US-6188799-$ or US-20060146941-$ or US-20130051454-$ or US-5852475-$ or US-6281942-$ or US-20050244063-$ or US-5796875-$ or US-20070025448-$ or US-6285801-$ or US-6529638-$ or US-5629769-$ or US-6167164-$ or US-20030206664-$ or US-20050013494-$ or US-20050276505-$ or US-20080075165-$ or US-20100027686-$ or US-20100002770-$ or US-20110274158-$ or US-20060078052-$ or US-5590064-$ or US-6728317-$ or US-7027654-$ or US-20090257664-$ or US-20080123750-$ or US-20080240252-$ or US-20120287988-$ "o...US-5576767-"$ or US-6327304-$ or US-5473384-$ or US-5831673-$ or US-5406326-$ or US-5479211-$ or US-6381277-$ or US-5929940-$ or US-6167157-$ or US-20060251330-$ or US-20030194009-$ or US-6950473-$ or US-6807317-$ or US-20050013497-$ or US-6909745-$ or US-20050259729-$ or US-20040013308-$ or US-20070160133-$ or US-20070091997-$ or US-20100128803-$ or US-20090257493-$ or US-20100046612-$ or US-20120033040-$ or US-20110026600-$ or US-20110026611-$ or US-20120082244-$ or US-20130051455-$ or US-20130177068-$ or US-5337086-$ or US-7031393-$ or US-4935816-$ or US-5787203-$ or US-5140414-$ or US-6037986-$ or US-5335013-$ or US-20020186890-$ or US-5565921-$ or US-20040017852-$ or US-20050276323-$ or US-20060098738-$ or US-20070223582-$ or US-20040179610-$ or US-20070009044-$ or US-20070074266-$ or US-20070140354-$ or US-20100260260-$ or US-20080175322-$ or US-20100284458-$ or US-20090086816-$ or US-20100008417-$ or US-20120027079-$ or US-20120092452-$ or US-5687011-$ or US-20130114677-$).did.				

S25	17	(US-20040101059-$ or US-20040062307-$ or US-20030053708-$ or US-20060126962-$ or US-20030053711-$ or US-20040076333-$ or US-20060268991-$ or US-20030152146-$ or US-6748113-$ or US-20030219073-$ or US-20030169931-$ or US-20030053541-$ or US-20030112864-$ or US-20080089417-$ or US-20060294171-$ or US-6539060-$ or US-6041145-$).did.	US-PGPUB; USPAT	OR	ON	2021/03/10 10:12
S26	12	deblock$3 same interpolation same ((boundary or edge) with ("near" or close or adjacent or neighbor$3))	US-PGPUB; USPAT	OR	ON	2021/03/10 10:20
S27	10	S26 and @ad<"20181108" and H04N19$.cpc.	US-PGPUB; USPAT	OR	ON	2021/03/10 10:20
S30	77	clip$4 with threshold with (quantization near5 parameter)	US-PGPUB; USPAT	OR	ON	2021/03/11 09:38
S31	28	(clip$4 with threshold with (quantization near5 parameter)) same (deblock$3 with filter)	US-PGPUB; USPAT	OR	ON	2021/03/11 09:39
S32	27	S31 and @ad<"20181108"	US-PGPUB; USPAT	OR	ON	2021/03/11 09:39

EAST Search History (Interference)

Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
L1	28	(clip$4 with threshold with (quantization near5 parameter)) same (deblock$3 with filter)	US-PGPUB; USPAT	OR	ON	2021/07/20 16:49
L3	56	deblock$3 same (boundary with ("near" or close or adjacent or neighbor$3)) same shift$3	US-PGPUB; USPAT	OR	ON	2021/07/20 16:50
L4	0	deblock$3 same (boundary with ("near" or close or adjacent or neighbor$3)) same shift$3 same ((three or seven) with sample)	US-PGPUB; USPAT	OR	ON	2021/07/20 16:51
L7	19	deblock$3 same (boundary with ("near" or close or adjacent or neighbor$3)) same shift$3 same sample	US-PGPUB; USPAT	OR	ON	2021/07/20 16:51
L9	457	interpolation same ((boundary or edge) with ("near" or close or adjacent or neighbor$3)) same sample	US-PGPUB; USPAT	OR	ON	2021/07/20 16:52
L11	10	interpolation same ((boundary or edge) with ("near" or close or adjacent or neighbor$3)) same sample same middle	US-PGPUB; USPAT	OR	ON	2021/07/20 16:53

7/20/2021 4:53:38 PM
C:\Users\kluo\Documents\EAST\Workspaces\Workspace2021\WS 16642748.wsp

Notificación de Concesión	**Solicitud No.** 16/642,748	**Solicitante(s)** ANDERSON et al.	
	Examinador KATE H LUO	**Unidad de Arte** 2488	**Estado AIA** Sí

-- La FECHA DE ENVIO de esta comunicación aparece en la hoja de cubierta con la dirección de correspondencia--

Todas las reivindicaciones son concedidas, TRÁMITE SOBRE EL FONDO ESTÁ (O PERMANECE) CERRADO en esta solicitud. Si no se incluye acá (o enviada previamente), una Notificación de Concesión (PTOL-85) u otra comunicación adecuada será enviada en su debido momento. **ESTA NOTIFICACION DE CONCESION NO ES UNA CONCESION DE DERECHOS DE PATENTE**. Esta solicitud está sujeta a retiro de emisión a la iniciativa de la Oficina o a petición del solicitante. Ver 37 CFR 1.313 y MPEP 1308.

1. ☑ Esta comunicación responde a 7/2/2021.

Una declaración bajo **37 CFR 1.130(b)** se presentó el_____.

2. ☐ Se hizo una elección por el solicitante en respuesta a un requerimiento de restricción definido durante la entrevista el ____; el requerimiento de restricción y la elección se han incorporado en esta acción.

3. ☑ Las reivindicaciones concedidas son *1-3, 5-6, 8-19 y 22*. Como resultado de las reivindicaciones aceptadas, usted puede ser elegible del beneficio del programa de **Procedimiento Acelerado de Patentes** en una oficina de propiedad intelectual participante para la correspondiente solicitud. Para más información, por favor ver http://www.uspto.gov/patents/init_events/pph/index.jsp o enviar una pregunta a PPHfeedback@uspto.gov.

4. ☐ Se reconoce una reivindicación de prioridad extranjera bajo 35 USC sección 119(a)-(d) o (f).

Copias certificadas:

a) ☐ Todas b) ☐ Algunas c) ☐ Ninguna de las:

1. ☐ Copias certificadas de los documentos de prioridad se han recibido.

2. ☐ Copias certificadas de los documentos de prioridad se han recibido en la Solicitud No. ____.

3. ☐ Copias de las copias certificadas de los documentos de prioridad se han recibido en esta solicitud de fase nacional de la Oficina Internacional (Regla PCT 17.2 (a)).

* Copias certificadas no recibidas: _____.

El Solicitante tiene TRES MESES A PARTIR DE LA “FECHA DE ENVIO” de esta comunicación para presentar una respuesta que cumpla con los requerimientos indicados anteriormente. Falla en cumplir a tiempo resultará en el ABANDONO de esta solicitud.

ESTE PERIODO DE TRES MESES NO ES PRORROGABLE

5. ☐ Se deben presentar FIGURAS CORREGIDAS (como “hojas de reemplazo”).

☐ Incluyendo cambios requeridos por la Enmienda/Comentario del Examinador adjunto o en la Acción Oficial de Papel No./Fecha de Envío ____.

Los indicios de identificación tal como el número de solicitud (ver 37 CFR 1.84(c)) se deben escribir en las figuras en el frente (no atrás) de cada hoja. Las hojas de reemplazo deben estar etiquetadas como tal en el encabezado de acuerdo con 37 CFR 1.121(d).

6. ☐ Se debe presentar DEPOSITO DE y/o INFORMACIÓN acerca del depósito de MATERIAL BIOLOGICO. Note el comentario del Examinador adjunto con relación al REQUERIMIENTO PARA EL DEPOSITO DE MATERIAL BIOLOGICO.

Adjunto(s)

1. ☐ Notificación de Referencias Citadas (PTO-892)
2. ☐ Declaraciones de Divulgación de Información (PTO/SB/08), Papel No./Fecha Envío____.
3. ☐ Comentario del Examinador con Relación a Requerimiento para Depósito de Material Biológico
4. ☑ Resumen de Entrevista (PTO-413), Papel No./Fecha Envío ____.
5. ☑ Enmienda/Comentario del Examinador
6. ☑ Declaración del Examinador de Razones para Concesión.
7. Otro ____.

/KATE H LUO/ Examinador Principal, Unidad de Arte 2488	

ENMIENDA DEL EXAMINADOR

1. A continuación se muestra una enmienda del examinador al expediente. Si los cambios y / o adiciones son inaceptables para el solicitante, se puede presentar una enmienda según lo dispuesto por 37 CFR 1.312. Para garantizar la consideración de dicha enmienda, DEBE presentarse a más tardar en el pago de la tarifa de emisión.

La autorización para la enmienda de este examinador se otorgó en una entrevista telefónica con el Sr. Brain Rosenbloom (Reg. No. 41.276) el 20 de julio de 2021.

La solicitud ha sido modificada como sigue:

2. La reivindicación 8 depende de la reivindicación 2 y se modifica como sigue.

8. (Modificado) El método de la reivindicación 1, en el que realizar la interpolación lineal entre refP y refMiddle comprende calcular ~~calcular~~ de p '(x) = (f (x) * refMiddle + (64-f (X)) * refP + 32) >> 6, donde X es de 0 a 4, p '(x) es una versión filtrada de px y f = {58,45,32,19,6}.

RAZONES PARA LA CONCESIÓN

1. La siguiente es una declaración del examinador de las razones para la concesión: La presente invención está relacionada con un método y un aparato para desbloquear un límite entre muestras de bloques.

2. La técnica anterior se encontró y aplicó en las acciones anteriores. En una enmienda con fecha del 2/7/2021, las reclamaciones se modificaron para incluir el tema que se había indicado anteriormente como admisible. Teniendo en cuenta el conjunto de las reivindicaciones recientemente modificadas del solicitante, no existe una motivación o un razonamiento sólidos para combinar referencias para llegar a la invención reivindicada. Las reivindicaciones 1-3, 5, 6, 8-19 y 22 son concedidas.

3. Cualquier comentario que el solicitante considere necesario debe enviarse antes del pago de la tasa de emisión y, para evitar retrasos en el procesamiento, preferiblemente debe acompañar a la tasa de emisión. Dichas presentaciones deben estar claramente etiquetadas como “Comentarios sobre la declaración de razones para la concesión”.

Conclusión

Cualquier duda relacionada con esta comunicación o comunicaciones anteriores del examinador se debe dirigir a KATE H LUO cuyo número de teléfono es (571)270-5635. El examinador puede ser encontrado normalmente de 8:00am - 5.00pm de lunes a viernes.

Si los intentos por encontrar al examinador por teléfono son fallidos, el supervisor del examinador, Sath V. Perungavoor se puede encontrar en (571)272-7455455. El número de fax para la organización donde esta solicitud o procedimiento es asignado es 571-273-8300.

La información relacionada con el estado de una solicitud se puede obtener del sistema de Recuperación de Información de Solicitud de Patente (PAIR). La información de estado para solicitudes publicadas se puede obtener del PAIR Privado o el PAIR Público. La información de estado para solicitudes sin publicar está disponible a través del PAIR Privado únicamente. Para mayor información acerca del sistema PAIR, ver http://pair-direct.uspto.gov. Si tiene preguntas sobre el acceso al sistema PAIR Privado, contacte el Centro de Negocio Electrónico (EBC)

en 866-217-9197 (totalmente gratis). Si desea asistencia de un Representante de Servicio al Cliente de la USPTO o acceso al sistema de información automatizado, llame a 800-786-9199 (EN USA O CANADA) o 571-272-1000.

/**KATE H LUO**/

Examinador Principal, Unidad de Arte 2488

Application No. : 16/642,748

Allowed Claims:

1. An asymmetric deblocking method for deblocking a boundary between a P block of samples and a Q block of samples such that samples within the P block and samples within the Q block are modified, the method comprising:

determining a value refP based on at least p5, wherein p5 is a sample within the P block and there are five other samples (p0, p1, p2, p3, and p4) within the P block that separate p5 from the boundary;

determining a value refQ based on at least qx, wherein qx is a sample within the Q block and either i) there are three other samples (q0, q1, and q2) within the Q block that separate qx from the boundary such that qx=q3, in which case refQ is equal to (q2+qx +1)>>1, or ii) there are seven other samples (q0, q1, q2, q3, q4, q5, and q6) within the Q block that separate qx from the boundary such that qx=q7, in which case refQ is equal to: (q6+qx+1)>>1;

determining a value refMiddle based on at least p0 and q0, wherein p0 is directly adjacent to the boundary and q0 is directly adjacent to the boundary, wherein i) in the case that qx=q3, refMiddle is equal to (p3+p2+p1+p0+q0+q1+q2+q3+4) >> 3 or ii) in the case that qx=7, refMiddle is equal to (p5+p4+p3+p2+2*(p1+p0+q0+q1)+q2+q3+q4+q5 + 8) >> 4);

performing a linear interpolation between refP and refMiddle; and

performing a linear interpolation between refQ and refMiddle.

2. The method of claim 1, wherein

the boundary is a vertical boundary,

samples p0 to p5 are aligned in a row that is orthogonal to the vertical boundary, and

samples q0 to q3 or samples q0 to q6 are aligned in the row.

3. The method of claim 1, wherein

the boundary is a horizontal boundary,

samples p0 to p5 are aligned in a column that is orthogonal to the horizontal boundary, and

samples q0 to q3 or samples q0 to q6 are aligned in the column.

4. The method of claim 1, wherein performing the linear interpolation between refP and refMiddle comprises calculating p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, where x is 0 to 4, p'(x) is a filtered version of px, and f = {58,45,32,19,6}.

5. The method of claim 1, wherein performing the linear interpolation between refQ and refMiddle comprises calculating q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, where x is 0 to 6, q'(x) is a filtered version of qx, and g = {59,50,41,32,23,14,5}.

6. The method of claim 1, wherein performing the linear interpolation between refP and refMiddle comprises calculating p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, where x is 0 to 4, p'(x) is a filtered version of px, and f = {58,45,32,19,6}.

7. The method of claim 1, wherein performing the linear interpolation between refQ and refMiddle comprises calculating q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, where x is 0 to 2, q'(x) is a filtered version of q(x), and g = {53,32,11}.

8. The method of claim 1, wherein refP is equal to: (p5+p4+1) >> 1.

9. The method of claim 1, wherein
the P block is above or to the left of the Q block, or
the P block is below or to the right of the Q block.

10. The method of claim 1, wherein
the P block has a size in a direction orthogonal to the boundary of 32 samples, and
the P block uses prediction sub-blocks.

11. The method of claim 10, wherein
the Q block has a size in a direction orthogonal to the boundary of less than 32 samples.

12. The method of claim 10, wherein

the Q block has a size in a direction orthogonal to the boundary of equal to or larger than 32 samples, and

the Q block does not uses sub-blocks.

13. The method of claim 1, wherein

the P block has a size in a direction orthogonal to the boundary of 32 samples, and

the boundary is a horizontal coding tree unit (CTU) boundary.

14. The method of claim 4, further comprising:

for x from 0 to 4, determining p''(x) = clip3(p'(x) + tcPx, p'(x) – tcPx, p'(x)), where tcPx is a clipping threshold, and

$$\text{clip3}(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases}.$$

15. The method of claim 5, further comprising:

for x from 0 to 2, determining q''(x) = clip3(q'(x) + tcQx, q'(x) – tcQx, q'(x)), where tcQx is a clipping threshold, and

$$\text{clip3}(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases}.$$

16. The method of claim 14, wherein the clipping threshold is a quantization parameter (QP) dependent clipping threshold.

17. A computer program product comprising a non-transitory computer readable medium storing a computer program comprising instructions which when executed by processing circuitry causes the processing circuitry to perform the method of claim 1.

18. An apparatus for deblocking a boundary between a P block of samples and a Q block of samples such that samples within the P block and samples within the Q block are modified , the apparatus comprising:

processing circuitry; and

a memory, said memory containing instructions executable by said processing circuitry, whereby said apparatus is operative to:

determine a value refP based on at least p5, wherein p5 is a sample within the P block and there are five other samples (p0, p1, p2, p3, and p4) within the P block that separate p5 from the boundary;

determine a value refQ based on at least qx, wherein qx is a sample within the Q block and either i) there are three other samples (q0, q1, and q2) within the Q block that separate qx from the boundary such that qx=q3, in which case refQ is equal to (q2+qx +1)>>1, or ii) there are seven other samples (q0, q1, q2, q3, q4, q5, and q6) within the Q block that separate qx from the boundary such that qx=q7, in which case refQ is equal to: (q6+qx+1)>>1;

determine a value refMiddle based on at least p0 and q0, wherein p0 is directly adjacent to the boundary and q0 is directly adjacent to the boundary, wherein i) in the case that qx=q3, refMiddle is equal to (p3+p2+p1+p0+q0+q1+q2+q3+4) >> 3 or ii) in the case that qx=7, refMiddle is equal to (p5+p4+p3+p2+2*(p1+p0+q0+q1)+q2+q3+q4+q5 + 8) >> 4);

perform a linear interpolation between refP and refMiddle, thereby modifying samples within the P block; and

perform a linear interpolation between refQ and refMiddle, thereby modifying samples within the Q block.

REIVINDICACIONES CONCEDIDAS US 16/842,748

1. Un método de desbloqueo asimétrico (1300, 1400) para desbloquear un límite entre un bloque P de muestras y un bloque Q de muestras de modo tal que las muestras dentro del bloque P y las muestras dentro del bloque Q se modifiquen, el método comprende:

determinar (1304, 1404) un valor refP según al menos p5, donde p5 es una muestra dentro del bloque P y hay otras cinco muestras (p0, p1, p2, p3 y p4) dentro del bloque P que separan p5 del límite;

determinar (1306, 1406) un valor refQ según al menos qx, donde qx es una muestra dentro del bloque Q y

i) hay otras tres muestras (q0, q1, y q2) dentro del bloque Q que separan qx del límite en el que qx=q3, en cuyo caso refQ es igual a (q2+qx +1)>>1,

o

ii) hay otras siete muestras (q0, q1, q2, q3, q4, q5 y q6) dentro del bloque Q que separa qx del límite en el que qx=q7, en cuyo caso refQ es igual a: (q6+qx+1)>>1;

determinar (1308, 1408) un valor refMiddle según al menos p0 y q0, donde p0 es directamente adyacente al límite y q0 es directamente adyacente al límite; donde i) en caso de que qx=q3, refMiddle es igual a (p3+p2+p1+p0+q0+q1+q2+q3+4) >> 3 o ii) en caso de que qx=7, refMiddle es igual a (p5+p4+p3+p2+2*(p1+p0+q0+q1)+q2+q3+q4+q5 + 8) >> 4);

realizar (1310, 1410) una interpolación lineal entre refP y refMiddle; y

realizar (1312, 1412) una interpolación lineal entre refQ y refMiddle.

2. El método de la reivindicación 1, donde: el límite es un límite vertical, las muestras p0 a p5 están alineadas en una fila que es ortogonal al límite vertical, y las muestras q0 a q3 o las muestras q0 a q6 están alineadas en la fila.

3. El método de la reivindicación 1, donde: el límite es un límite horizontal, las muestras p0 a p5 están alineadas en una columna que es ortogonal al límite horizontal, y las muestras q0 a q3 o las muestras q0 a q6 están alineadas en la columna.

4. El método de la reivindicación 1, donde realizar la interpolación lineal entre refP y refMiddle comprende calcular p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, donde x es de 0 a 4, p'(x) es una versión filtrada de px y f = {58,45,32,19,6}.

5. El método de la reivindicación 1, donde realizar la interpolación lineal entre refQ y refMiddle comprende calcular q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, donde x es de 0 a 6, q'(x) es una versión filtrada de qx, y g = {59,50,41,32,23,14,5}.

6. El método de la reivindicación 1, donde realizar la interpolación lineal entre refP y refMiddle comprende calcular p'(x) = (f(x)*refMiddle + (64-f(x))*refP + 32) >> 6, donde x es de 0 a 4, p'(x) es una versión filtrada de px y f = {58,45,32,19,6}.

7. El método de la reivindicación 1, donde realizar la interpolación lineal entre refQ y refMiddle comprende calcular q'(x) = (g(x)*refMiddle + (64-g(x))*refQ + 32) >> 6, donde x es de 0 a 2, q'(x) es una versión filtrada de q(x), y g = {53,32,11}.

8. El método de la reivindicación 1, donde refP es igual a: (p5+p4+1) >> 1.

9. El método de la reivindicación 1, donde: el bloque P está arriba o a la izquierda del bloque Q, o el bloque P está debajo o a la derecha del bloque Q.

10. El método de la reivindicación 1, donde: el bloque P tiene un tamaño en una dirección ortogonal al límite de 32 muestras; el bloque P usa subbloques de predicción.

11. El método de la reivindicación 10, donde: el bloque Q tiene un tamaño en una dirección ortogonal al límite de menos de 32 muestras.

12. El método de la reivindicación 10, donde: el bloque Q tiene un tamaño en una dirección ortogonal al límite igual o superior a 32 muestras; y el bloque Q no utiliza subbloques.

13. El método de la reivindicación 1, donde: el bloque P tiene un tamaño en una dirección ortogonal al límite de 32 muestras; y el límite es un límite de unidad de árbol de codificación horizontal (CTU).

14. El método de la reivindicación 4, que además comprende: para x de 0 a 4, determinando $p''(x) = clip3(p'(x) + tcP_x, p'(x) - tcP_x, p'(x))$, donde tcP_x es un umbral de recorte y

$$clip3(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{de lo contrario} \end{cases}.$$

15. El método de la reivindicación 5, que además comprende: para x de 0 a 2, determinando $q''(x) = clip3(q'(x) + tcQ_x, q'(x) - tcQ_x, q'(x))$, donde tcQ_x es un umbral de recorte y

$$clip3(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{de lo contrario} \end{cases}.$$

16. El método de la reivindicación 14, donde el umbral de recorte es un umbral de recorte dependiente del parámetro de cuantificación (QP).

17. Un producto de programa informático que comprende un medio legible por ordenador no transitorio que almacena un programa informático que comprende instrucciones que, cuando se ejecutan mediante circuitos de procesamiento, hacen que los circuitos de procesamiento realicen el método de la reivindicación 1.

18. Un aparato (1501) para desbloquear un límite entre un bloque P de muestras y un bloque Q de muestras de modo tal que las muestras dentro del bloque P y las muestras dentro del bloque Q se modifiquen, el aparato comprende:

circuitería de procesamiento; y

una memoria, conteniendo dicha memoria instrucciones ejecutables por dicha circuitería de procesamiento, por lo que dicho aparato es operativo para:

determinar un valor refP basado en al menos p5, donde p5 es una muestra dentro del bloque P y hay otras cinco muestras (p0, p1, p2, p3 y p4) dentro del bloque P que separan p5 del límite;

determinar un valor refQ basado en al menos qx, donde qx es una muestra dentro del bloque Q y i) hay otras tres muestras (q0, q1 y q2) dentro del bloque Q que separan qx del límite de tal manera que qx = q3, en cuyo caso refQ es igual a (q2 + qx +1) >> 1, o ii) hay otras siete muestras (q0, q1, q2, q3, q4, q5 y q6) dentro del bloque Q que separan qx desde el límite tal que qx = q7, en cuyo caso refQ es igual a: (q6 + qx + 1) >> 1;

determinar un valor refMiddle basado en al menos p0 y q0, donde

p0 es directamente adyacente al límite y q0 es directamente adyacente al límite, donde i) en el caso de que qx = q3, refMiddle es igual a (p3 + p2 + p1 + p0 + q0 + q1 + q2 + q3 + 4) >> 3 o ii) en el caso de que qx = 7, refMiddle es igual a (p5 + p4 + p3 + p2 + 2 * (p1 + p0 + q0 + q1) + q2 + q3 + q4 + q5 + 8) >> 4);

realizar una interpolación lineal entre refP y refMiddle, modificando así las muestras dentro del bloque P; y

realizar una interpolación lineal entre refQ y refMiddle, modificando así las muestras dentro del bloque Q.

Joint Video Experts Team (JVET)
of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11
12th Meeting: Macao, CN, 3–12 Oct. 2018

Document: JVET-L1001-v1

Title:	**Versatile Video Coding (Draft 3)**		
Status:	Output document approved by JVET		
Purpose:	Draft text of video coding specification		
Author(s) or Contact(s):	Benjamin Bross Jianle Chen Shan Liu	Email:	firstname.lastname@hhi.fraunhofer.de jianle.chen@huawei.com shanl@tencent.com
Source:	Editors		

Abstract

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

Draft 1 of Versatile Video Coding.

Ed. Notes:

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

Draft 2 of Versatile Video Coding.

Ed. Notes:

- Incorporated JVET-K0230: Separate trees for intra slices (without multi-DMs) with an implicit split to 64x64;
- Incorporated JVET-K0556: Prohibit ternary split of something bigger than 64 in width or height (and not send the bit to indicate ternary type at that level).
- Incorporated JVET-K0351 (test c): Keep only the TT restriction (preventing binary split with same orientation in center partition of the ternary split)
- Incorporated JVET-K0554: Implicit splitting at picture boundaries and ensure MinQTSize at boundary splits
- Fixed bug #65 typos and unused variables in section 6.4
- Fixed bug #67 implicit vertical BT split at picture boundary issue
- Incorporated JVET-K0072: Dependent quantization with fallback switch at the picture level and modified entropy coding supporting dependent quantization including:
    - adapted scaling to non-square transform blocks,
    - added binarization process for abs_remainder,
    - specified CoeffMin and CoeffMax with fixed values,
    - added 0-th order Exp-Golomb code parsing process.
- Incorporated JVET-K0310: Sign data hiding (can only be used when dependent quantization is disabled).
- Incorporated JVET-K0529: Intra prediction using 3MPM on 67 prediction modes (Planar, DC and 65 angular modes)
- Incorporated JVET-K0122: DC prediction without division.
- Incorporated JVET-K0500: Wide-angle intra prediction.

- Incorporated JVET-K0063: Position-dependent intra prediction combination
- Incorporated JVET-K0190: Cross-component linear model intra prediction
- Incorporated multiple transfrom selection (MTS) for both intra and inter, each controlled by an SPS flag.
- Incorporated transform skip.
- Fixed bug #68 various typos
- Fixed bug #71 various typos
- Fixed bug #72 on CCLM
- Incorporated JVET-K0357: adaptive motion vector resolution (AMVR)
- Incorpor,ated JVET-K0565: affine motion compensation (MC) including:
  - JVET-K0052: Affine merge bug fix
  - JVET-K0184: Affine MC (CE4.1.1a 4x4 fixed subblock size).
  - JVET-K0337: Affine MC coding and models (4.1.3a, affine MVP list construction, and 4.1.3b, MV difference coding, and 4.1.3c, 4/6 parameter model, no slice level switch).
  - JVET-K0367/JVET-K0052/JVET-K0103: Restriction of affine merge mode to CU sizes >= 8x8
- Incorporated 1/16 motion compensation (MC) including:
  - 1/16 MV storage
  - 1/16 merge and affine MVs
  - MVDs in AMVR accuracy (1/4,1,4) shifted to 1/16
  - Inter MVP candidates rounded to AMVR accuracy (1/4,1,4) and shifted to 1/16
  - 1/16 luma and 1/32 chroma interpolation filters
- Incorporated subblock-based temporal merging candidates with 8x8 motion vector storage (JVET-K0346).
- Incorporated JVET-K0371: 4x4 block classification based Adaptive Loop Filter (ALF).
- Fixed bug #75 regarding a bottom and right boundary partition issue.
- Fixed bug #90 typos in copying the control point vectors to temporal notion vectors.
- Fixed bug #86 in intra reference sample filtering.
- Incorporated JVET-K0325: High Level Syntax (HLS) starting point.
- Fixed bug #82 on zeroing-out high frequency transform coefficients for larger TUs (>32x32).
- Fixed bug #85 on MTS index coding.

Draft 3 of Versatile Video Coding.

Ed. Notes:
- Fixed bug #92 missing MV clipping in interplolation process.
- Incorporated JVET-L0664: disabling 5x5 ALF for luma component.
- Incorporated JVET-L0082/JVET-L0083: reduction of bits for ALF coefficients.
- Incorporated JVET-L0147: subsampling of ALF classifiers.
- Incorporated SAO as found in HEVC.
- Incorporated deblocking filter as found in HEVC with the following modifications:
  - adapt processing to dual tree CTU partitioning,
  - replace RQT based transfrom block processing with implicit TU split for large blocks (JVET-K0307, JVET-K0237, JVET-K0369, JVET-K0232, JVET-K0315),
  - replace prediction blocks with coding subblocks (JVET-L0074),
  - only apply deblocking on transfrom block / coding subblock edges if CU is aligned on 8x8 grid in the respecitive direction.
- Incorporated JVET-L0410: tc table fix
- Incorporated JVET-L0414: luma adaptive deblocking filter.

CONTENTS

Page

Abstract i

1 Scope 1

2 Normative references 1
- 2.1 Identical Recommendations | International Standards 1
- 2.2 Paired Recommendations | International Standards equivalent in technical content 1
- 2.3 Additional references 1

3 Definitions 1

4 Abbreviations 5

5 Conventions 8
- 5.1 General 8
- 5.2 Arithmetic operators 8
- 5.3 Logical operators 8
- 5.4 Relational operators 8
- 5.5 Bit-wise operators 8
- 5.6 Assignment operators 9
- 5.7 Range notation 9
- 5.8 Mathematical functions 9
- 5.9 Order of operation precedence 10
- 5.10 Variables, syntax elements and tables 11
- 5.11 Text description of logical operations 12
- 5.12 Processes 13

6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 14
- 6.1 Bitstream formats 14
- 6.2 Source, decoded and output picture formats 14
- 6.3 Partitioning of pictures, slices, CTUs 16
  - 6.3.1 Partitioning of pictures into slices 16
  - 6.3.2 Block, quadtree and multi-type tree structures 17
  - 6.3.3 Spatial or component-wise partitionings 17
- 6.4 Availability processes 17
  - 6.4.1 Allowed binary split process 17
  - 6.4.2 Allowed ternary split process 18
- 6.5 Scanning processes 19
  - 6.5.1 CTB raster and scanning process 19
  - 6.5.2 Up-right diagonal scan order array initialization process 19

7 Syntax and semantics 20
- 7.1 Method of specifying syntax in tabular form 20
- 7.2 Specification of syntax functions and descriptors 21
- 7.3 Syntax in tabular form 22
  - 7.3.1 NAL unit syntax 22
  - 7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 23
  - 7.3.3 Slice header syntax 25
  - 7.3.4 Slice data syntax 27
- 7.4 Semantics 38
  - 7.4.1 General 38
  - 7.4.2 NAL unit semantics 38
  - 7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 40
  - 7.4.4 Slice header semantics 44
  - 7.4.5 Slice data semantics 48

8 Decoding process 58
- 8.1 General decoding process 58
- 8.2 Decoding process for coding units coded in intra prediction mode 58
  - 8.2.1 General decoding process for coding units coded in intra prediction mode 58

8.2.2 Derivation process for luma intra prediction mode 59
8.2.3 Derivation process for chroma intra prediction mode 60
8.2.4 Decoding process for intra blocks 61
8.3 Decoding process for coding units coded in inter prediction mode 74
8.3.1 General decoding process for coding units coded in inter prediction mode 74
8.3.2 Derivation process for motion vector components and reference indices 76
8.3.3 Derivation process for affine motion vector components and reference indices 95
8.3.4 Decoding process for inter blocks 105
8.3.5 Decoding process for the residual signal of coding blocks coded in inter prediction mode 111
8.4 Scaling, transformation and array construction process 112
8.4.1 Derivation process for quantization parameters 112
8.4.2 Scaling and transformation process 112
8.4.3 Scaling process for transform coefficients 113
8.4.4 Transformation process for scaled transform coefficients 114
8.4.5 Picture reconstruction process 122
8.5 In-loop Filter Process 122
8.5.1 General 122
8.5.2 Deblocking filter process 122
8.5.3 Sample adaptive offset process 138
8.5.4 Adaptive loop filter process 140
9 Parsing process 144
9.1 General 144
9.2 Parsing process for k-th order Exp-Golomb codes 145
9.2.1 General 145
9.2.2 Mapping process for signed Exp-Golomb codes 146
9.3 Parsing process for truncated unary codes 147
9.4 Parsing process for truncated binary codes 147
9.5 CABAC parsing process for slice segment data 147
9.5.1 General 147
9.5.2 Initialization process 147
9.5.3 Binarization process 147
9.5.4 Decoding process flow 153

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Versatile video coding

1 Scope

This Recommendation | International Standard specifies versatile video coding.

2 Normative references

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

2.1 Identical Recommendations | International Standards

– None

2.2 Paired Recommendations | International Standards equivalent in technical content

– None

2.3 Additional references

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

3 Definitions

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

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

3.1 **access unit**: A set of *NAL units* that are associated with each other according to a specified classification rule, are consecutive in *decoding order,* and contain exactly one *coded picture*.

3.2 **AC transform coefficient**: Any *transform coefficient* for which the *frequency index* in at least one of the two dimensions is non-zero.

3.3 **bin**: One bit of a *bin string*.

3.4 **binarization**: A set of *bin strings* for all possible values of a *syntax element*.

3.5 **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.

3.6 **binary split**: A split of a rectangular MxN *block* of samples into two *blocks* where a vertical split results in a first (M / 2)xN *block* and a second (M / 2)xN *block*, and a horizontal split results in a first Mx(N / 2) *block* and a second Mx(N / 2) *block.*

3.7 **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.

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

3.9 **bitstream**: A sequence of bits, in the form of a *NAL unit stream* or a *byte stream*, that forms the representation of *coded pictures* and associated data forming one or more coded video sequences *(CVSs)*.

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

3.11 **byte**: A sequence of 8 bits, within which, when written or read as a sequence of bit values, the left-most and right-most bits represent the most and least significant bits, respectively.

3.12 **byte-aligned**: A position in a *bitstream* is byte-aligned when the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream*, and a bit or *byte* or *syntax element* is said to be byte-aligned when the position at which it appears in a *bitstream* is byte-aligned.

3.13 **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex TBD.

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

3.15 **chroma**: An adjective, represented by the symbols Cb and Cr, specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours.

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

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

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

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

3.19 **coding block**: An MxN *block* of samples for some values of M and N such that the division of a *CTB* into *coding blocks* is a *partitioning*.

3.20 **coding tree block (CTB)**: An NxN *block* of samples for some value of N such that the division of a *component* into *CTBs* is a *partitioning*.

3.21 **coding tree unit (CTU)**: A *CTB* of *luma* samples, two corresponding *CTBs* of *chroma* samples of a *picture* that has three sample arrays, or a *CTB* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.

3.22 **coding unit (CU)**: A *coding block* of *luma* samples, two corresponding *coding blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.

3.23 **component**: An array or single sample from one of the three arrays (*luma* and two *chroma*) that compose a *picture* in 4:2:0, 4:2:2, or 4:4:4 colour format or the array or a single sample of the array that compose a *picture* in monochrome format.

3.24 **context variable**: A variable specified for the *adaptive binary arithmetic decoding process* of a *bin* by an equation containing recently decoded *bins*.

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

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

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

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

3.29 **emulation prevention byte**: A *byte* equal to 0x03 that is present within a *NAL unit* when the *syntax elements* of the *bitstream* form certain patterns of *byte* values in a manner that ensures that no sequence of consecutive *byte-aligned bytes* in the *NAL unit* can contain a *start code prefix*.

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

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

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

3.33 **frequency index**: A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to the application of a *transform* in the *decoding process*.

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

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

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

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

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

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

3.39 **intra random access point (IRAP) access unit**: An *access unit* in which the *coded picture* is an *IRAP picture*.

3.40 **intra random access point (IRAP) picture**: A *coded picture* for which each *VCL NAL unit* has nal_unit_type equal to IRAP_NUT.

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

3.42 **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.

3.43 **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this Specification, or the value of a *transform coefficient* prior to *scaling*.

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

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

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

3.46 **luma**: An adjective, represented by the symbol or subscript Y or L, specifying that a sample array or single sample is representing the monochrome signal related to the primary colours.

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

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

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

3.48 **motion vector**: A two-dimensional vector used for *inter prediction* that provides an offset from the coordinates in the *decoded picture* to the coordinates in a *reference picture*.

3.49 **multi-type tree**: A *tree* in which a parent node can be split either into two child nodes using a *binary split* or into three child nodes using a *ternary split*, each of which may become parent node for another split into either two or three child nodes.

3.50 **must**: A term that is used in expressing an observation about a requirement or an implication of a requirement that is specified elsewhere in this Specification (used exclusively in an *informative* context).

3.51 **network abstraction layer (NAL) unit**: A *syntax structure* containing an indication of the type of data to follow and *bytes* containing that data in the form of an *RBSP* interspersed as necessary with *emulation prevention bytes*.

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

3.53 **non-IRAP picture**: A *coded picture* for which each *VCL NAL unit* has nal_unit_type equal to NON_IRAP_NUT.

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

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

3.56 **output order**: The order in which the *decoded pictures* are output from the *decoded picture buffer* (for the *decoded pictures* that are to be output from the *decoded picture buffer*).

3.57 **parameter**: A *syntax element* of a *sequence parameter set (SPS)* or *picture parameter set (PPS)*, or the second word of the defined term *quantization parameter*.

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

3.59 **picture**: An array of *luma* samples in monochrome format or an array of *luma* samples and two corresponding arrays of *chroma* samples in 4:2:0, 4:2:2, and 4:4:4 colour format.

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

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

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

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

3.63 **prediction process**: The use of a *predictor* to provide an estimate of the data element (e.g., sample value or motion vector) currently being decoded.

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

3.65 **predictor**: A combination of specified values or previously decoded data elements (e.g., sample value or motion vector) used in the *decoding process* of subsequent data elements.

3.66 **profile**: A specified subset of the syntax of this Specification.

3.67 **pulse code modulation (PCM)**: Coding of the samples of a *block* by directly representing the sample values without *prediction* or application of a transform.

3.68 **quadtree**: A *tree* in which a parent node can be split into four child nodes, each of which may become parent node for another split into four child nodes.

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

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

3.71 **raster scan**: A mapping of a rectangular two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the first top row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc., rows of the pattern (going down) each scanned from left to right.

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

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

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

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

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

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

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

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

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

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

3.80 **residual**: The decoded difference between a *prediction* of a sample or data element and its decoded value.

3.81 **scaling**: The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.

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

3.83 **shall**: A term used to express mandatory requirements for conformance to this Specification.

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

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

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

3.86 **slice**: An integer number of *CTUs* ordered consecutively in the *raster scan* and contained in a single *NAL unit*.

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

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

3.89 **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit*.

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

3.90 **string of data bits (SODB)**: A sequence of some number of bits representing *syntax elements* present within a *raw byte sequence payload* prior to the *raw byte sequence payload stop bit*, where the left-most bit is considered to be the first and most significant bit, and the right-most bit is considered to be the last and least significant bit.

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

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

3.93 **ternary split**: A split of a rectangular MxN *block* of samples into three *blocks* where a vertical split results in a first (M / 4)xN *block*, a second (M / 2)xN *block*, a third (M / 4)xN *block*, and a horizontal split results in a first Mx(N / 4) *block*, a second Mx(N / 2) *block*, a third Mx(N / 4) *block*.

3.94 **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.

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

3.96 **transform coefficient**: A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in a *transform* in the *decoding process*.

3.97 **transform coefficient level**: An integer quantity representing the value associated with a particular two-dimensional frequency index in the *decoding process* prior to *scaling* for computation of a *transform coefficient* value.

3.98 **transform unit (TU)**: A *transform block* of *luma* samples and two corresponding *transform blocks* of *chroma* samples of a *picture* and *syntax structures* used to transform the *transform block* samples.

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

3.100 **unspecified**: A term that may be used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.

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

4 Abbreviations

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

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

ALF	Adaptive Loop Filter
AMVR	Adaptive Motion Vector Resolution
B	Bi-predictive
CABAC	Context-based Adaptive Binary Arithmetic Coding
CB	Coding Block
CBR	Constant Bit Rate
CPB	Coded Picture Buffer
CRC	Cyclic Redundancy Check
CTB	Coding Tree Block
CTU	Coding Tree Unit
CU	Coding Unit
CVS	Coded Video Sequence
DPB	Decoded Picture Buffer
EG	Exponential-Golomb
EGk	k-th order Exponential-Golomb
FCC	Federal Communications Commission (of the United States)
FIFO	First-In, First-Out
FIR	Finite Impulse Response
FL	Fixed-Length
GBR	Green, Blue and Red
I	Intra
IRAP	Intra Random Access Point
LPS	Least Probable Symbol
LSB	Least Significant Bit
MPS	Most Probable Symbol
MSB	Most Significant Bit
MTS	Multiple Transform Selection
MVP	Motion Vector Prediction
NAL	Network Abstraction Layer
NTSC	National Television System Committee (of the United States)
P	Predictive
PCM	Pulse Code Modulation
POC	Picture Order Count
PPS	Picture Parameter Set
QP	Quantization Parameter
RBSP	Raw Byte Sequence Payload
RGB	Same as GBR
RPS	Reference Picture Set
SAO	Sample Adaptive Offset
SAR	Sample Aspect Ratio
SEI	Supplemental Enhancement Information
SMPTE	Society of Motion Picture and Television Engineers
SODB	String Of Data Bits
SPS	Sequence Parameter Set
TR	Truncated Rice
UCS	Universal Coded Character Set

UTF	UCS Transmission Format
VBR	Variable Bit Rate
VCL	Video Coding Layer

5 Conventions

5.1 General

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

5.2 Arithmetic operators

The following arithmetic operators are defined as follows:

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

5.3 Logical operators

The following logical operators are defined as follows:

x && y	Boolean logical "and" of x and y
x || y	Boolean logical "or" of x and y
!	Boolean logical "not"
x ? y : z	If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

5.4 Relational operators

The following relational operators are defined as follows:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
= =	Equal to
!=	Not equal to

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

5.5 Bit-wise operators

The following bit-wise operators are defined as follows:

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

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

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

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

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

5.6 Assignment operators

The following arithmetic operators are defined as follows:

= Assignment operator

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

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

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

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

5.7 Range notation

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

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

5.8 Mathematical functions

The following mathematical functions are defined:

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

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

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

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

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

$$\mathrm{Clip1}_Y(x) = \mathrm{Clip3}(0, (1 << \mathrm{BitDepth}_Y) - 1, x) \tag{5-6}$$

$$Clip1_C(x) = Clip3(0, (1 << BitDepth_C) - 1, x) \quad (5\text{-}7)$$

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

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

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

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

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

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

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

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

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

$$Round(x) = Sign(x) * Floor(Abs(x) + 0.5) \quad (5\text{-}17)$$

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

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

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

$$Swap(x, y) = (y, x) \quad (5\text{-}21)$$

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

5.9 Order of operation precedence

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

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

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

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

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

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

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

5.10 Variables, syntax elements and tables

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

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

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

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

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

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

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column

(horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[x][y] or as s_{yx}. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[x].

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

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

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

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

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

5.11 Text description of logical operations

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

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

may be described in the following manner:

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

- If condition 0, statement 0
- Otherwise, if condition 1, statement 1
- ...
- Otherwise (informative remark on remaining condition), statement n

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

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

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

may be described in the following manner:

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

- If all of the following conditions are true, statement 0:
  - condition 0a
  - condition 0b
- Otherwise, if one or more of the following conditions are true, statement 1:

  - condition 1a
  - condition 1b
- ...
- Otherwise, statement n

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

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

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

5.12 Processes

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

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

- If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.
- Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

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

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

6.1 Bitstream formats

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

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

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

6.2 Source, decoded and output picture formats

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

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

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

– Luma (Y) only (monochrome).

– Luma and two chroma (YCbCr or YCgCo).

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

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

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

The variables SubWidthC and SubHeightC are specified in Table 6-1, depending on the chroma format sampling structure, which is specified through chroma_format_idc and separate_colour_plane_flag. Other values of chroma_format_idc, SubWidthC and SubHeightC may be specified in the future by ITU-T | ISO/IEC.

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

chroma_format_idc	separate_colour_plane_flag	Chroma format	SubWidthC	SubHeightC
0	0	Monochrome	1	1
1	0	4:2:0	2	2
2	0	4:2:2	2	1
3	0	4:4:4	1	1
3	1	4:4:4	1	1

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

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

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

In 4:4:4 sampling, depending on the value of separate_colour_plane_flag, the following applies:

– If separate_colour_plane_flag is equal to 0, each of the two chroma arrays has the same height and width as the luma array.

– Otherwise (separate_colour_plane_flag is equal to 1), the three colour planes are separately processed as monochrome sampled pictures.

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

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

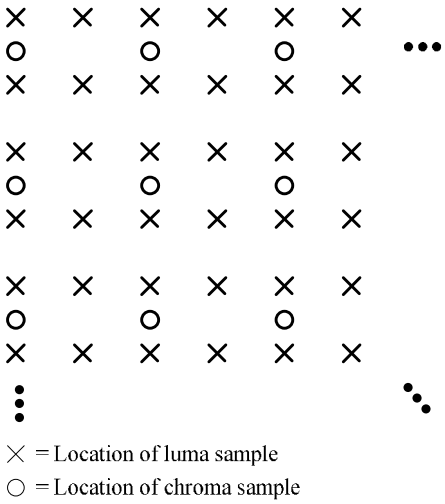


Figure 6-1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture

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

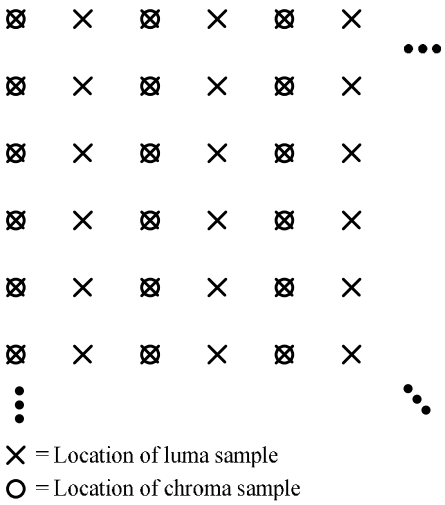


Figure 6-2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture

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

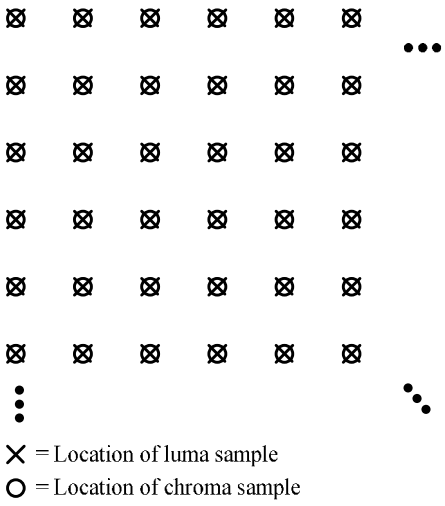


Figure 6-3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture

6.3 Partitioning of pictures, slices, CTUs

6.3.1 Partitioning of pictures into slices

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

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

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

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

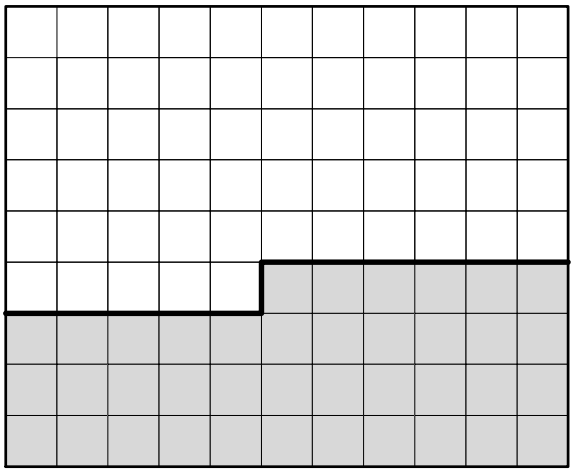


Figure 6-4 – A picture with 11 by 9 luma CTUs that is partitioned into two slices (informative)

6.3.2 Block, quadtree and multi-type tree structures

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

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

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

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

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

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

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

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

6.3.3 Spatial or component-wise partitionings

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

– The division of each picture into components

– The division of each component into CTBs

– The division of each picture into slices

– The division of each slice into CTUs

– The division of each CTU into CTBs

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

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

– The division of each coding unit into transform units

– The division of each coding unit into coding blocks

– The division of each coding block into transform blocks

– The division of each transform unit into transform blocks.

6.4 Availability processes

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

6.4.1 Allowed binary split process

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

Output of this process is the variable allowBtSplit.

Table 6-2 – Specification of parallelTtSplit and cbSize based on btSplit.

	btSplit = = SPLIT_BT_VER	**btSplit = = SPLIT_BT_HOR**
parallelTtSplit	SPLIT_TT_VER	SPLIT_TT_HOR
cbSize	cbWidth	cbHeight

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

The variable allowBtSplit is derived as follows:

– If one or more of the following conditions are true, allowBtSplit is set equal to FALSE:
  – cbSize is less than or equal to MinBtSizeY
  – cbWidth is greater than MaxBtSizeY
  – cbHeight is greater than MaxBtSizeY
  – mttDepth is greater than or equal to MaxMttDepth + depthOffset
– Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
  – btSplit is equal to SPLIT_BT_VER
  – y0 + cbHeight is greater than pic_height_in_luma_samples
– Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
  – btSplit is equal to SPLIT_BT_HOR
  – x0 + cbWidth is greater than pic_width_in_luma_samples
  – y0 + cbHeight is smaller than or equal to pic_height_in_luma_samples
– Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE:
  – mttDepth is greater than 0
  – partIdx is equal to 1
  – MttSplitMode[x0][y0][mttDepth − 1] is equal to parallelTtSplit
– Otherwise, allowBtSplit is set equal to TRUE.

6.4.2 Allowed ternary split process

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

Output of this process is the variable allowTtSplit.

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

	ttSplit = = SPLIT_TT_VER	**ttSplit = = SPLIT_TT_HOR**
cbSize	cbWidth	cbHeight

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

The variable allowTtSplit is derived as follows:

– If one or more of the following conditions are true, allowTtSplit is set equal to FALSE:
  – cbSize is less than or equal to 2 * MinTtSizeY
  – cbWidth is greater than MaxTtSizeY
  – cbHeight is greater than MaxTtSizeY

- mttDepth is greater than or equal to MaxMttDepth + depthOffset
- x0 + cbWidth is greater than pic_width_in_luma_samples
- y0 + cbHeight is greater than pic_height_in_luma_samples

– Otherwise, allowTtSplit is set equal to TRUE.

6.5 Scanning processes

6.5.1 CTB raster and scanning process

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

6.5.2 Up-right diagonal scan order array initialization process

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

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

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

7 Syntax and semantics

7.1 Method of specifying syntax in tabular form

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

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

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

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

7.2 Specification of syntax functions and descriptors

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

byte_aligned( ) is specified as follows:

- If the current position in the bitstream is on a byte boundary, i.e., the next bit in the bitstream is the first bit in a byte, the return value of byte_aligned( ) is equal to TRUE.
- Otherwise, the return value of byte_aligned( ) is equal to FALSE.

more_data_in_byte_stream( ), which is used only in the byte stream NAL unit syntax structure specified in Annex TBD, is specified as follows:

- If more data follow in the byte stream, the return value of more_data_in_byte_stream( ) is equal to TRUE.
- Otherwise, the return value of more_data_in_byte_stream( ) is equal to FALSE.

more_data_in_payload( ) is specified as follows:

- If byte_aligned( ) is equal to TRUE and the current position in the sei_payload( ) syntax structure is 8 * payloadSize bits from the beginning of the sei_payload( ) syntax structure, the return value of more_data_in_payload( ) is equal to FALSE.
- Otherwise, the return value of more_data_in_payload( ) is equal to TRUE.

more_rbsp_data( ) is specified as follows:

- If there is no more data in the raw byte sequence payload (RBSP), the return value of more_rbsp_data( ) is equal to FALSE.
- Otherwise, the RBSP data are searched for the last (least significant, right-most) bit equal to 1 that is present in the RBSP. Given the position of this bit, which is the first bit (rbsp_stop_one_bit) of the rbsp_trailing_bits( ) syntax structure, the following applies:
  - If there is more data in an RBSP before the rbsp_trailing_bits( ) syntax structure, the return value of more_rbsp_data( ) is equal to TRUE.
  - Otherwise, the return value of more_rbsp_data( ) is equal to FALSE.

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

more_rbsp_trailing_data( ) is specified as follows:

- If there is more data in an RBSP, the return value of more_rbsp_trailing_data( ) is equal to TRUE.
- Otherwise, the return value of more_rbsp_trailing_data( ) is equal to FALSE.

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

payload_extension_present( ) is specified as follows:

- If the current position in the sei_payload( ) syntax structure is not the position of the last (least significant, right-most) bit that is equal to 1 that is less than 8 * payloadSize bits from the beginning of the syntax structure (i.e., the position of the payload_bit_equal_to_one syntax element), the return value of payload_extension_present( ) is equal to TRUE.
- Otherwise, the return value of payload_extension_present( ) is equal to FALSE.

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

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

- ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in clause TBD.
- b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read_bits(8).

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

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

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

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

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

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

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

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

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

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

7.3 Syntax in tabular form

7.3.1 NAL unit syntax

7.3.1.1 General NAL unit syntax

nal_unit(NumBytesInNalUnit) {	**Descriptor**
nal_unit_header( )	
NumBytesInRbsp = 0	
for(i = 2; i < NumBytesInNalUnit; i++)	
if(i + 2 < NumBytesInNalUnit && next_bits(24) = = 0x000003) {	
rbsp_byte[NumBytesInRbsp++]	b(8)
rbsp_byte[NumBytesInRbsp++]	b(8)
i += 2	
emulation_prevention_three_byte /* equal to 0x03 */	f(8)
} else	
rbsp_byte[NumBytesInRbsp++]	b(8)
}	

7.3.1.2 NAL unit header syntax

nal_unit_header( ) {	Descriptor
forbidden_zero_bit	f(1)
nal_unit_type	u(5)
nuh_temporal_id_plus1	u(3)
nuh_reserved_zero_7bits	u(7)
}	

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

7.3.2.1 Sequence parameter set RBSP syntax

seq_parameter_set_rbsp( ) {	Descriptor
sps_seq_parameter_set_id	ue(v)
chroma_format_idc	ue(v)
if(chroma_format_idc = = 3)	
separate_colour_plane_flag	u(1)
pic_width_in_luma_samples	ue(v)
pic_height_in_luma_samples	ue(v)
bit_depth_luma_minus8	ue(v)
bit_depth_chroma_minus8	ue(v)
qtbtt_dual_tree_intra_flag	ue(v)
log2_ctu_size_minus2	ue(v)
log2_min_qt_size_intra_slices_minus2	ue(v)
log2_min_qt_size_inter_slices_minus2	ue(v)
max_mtt_hierarchy_depth_inter_slices	ue(v)
max_mtt_hierarchy_depth_intra_slices	ue(v)
sps_cclm_enabled_flag	u(1)
sps_sao_enabled_flag	u(1)
sps_alf_enabled_flag	u(1)
sps_temporal_mvp_enabled_flag	u(1)
if(sps_temporal_mvp_enabled_flag)	
sps_sbtmvp_enabled_flag	u(1)
if(sps_sbtmvp_enabled_flag)	
log2_sbtmvp_default_size_minus2	u(1)
sps_amvr_enabled_flag	u(1)
sps_affine_enabled_flag	u(1)
if(sps_affine_enabled_flag)	
sps_affine_type_flag	u(1)
sps_mts_intra_enabled_flag	u(1)
sps_mts_inter_enabled_flag	u(1)
sps_ladf_enabled_flag	u(1)
if (sps_ladf_enabled_flag) {	
sps_num_ladf_intervals_minus2	u(2)
sps_ladf_lowest_interval_qp_offset	se(v)
for(i = 0; i < sps_num_ladf_intervals_minus2 + 1; i++) {	
sps_ladf_qp_offset[i]	se(v)
sps_ladf_delta_threshold_minus1[i]	ue(v)

}	
}	
rbsp_trailing_bits( )	
}	

7.3.2.2 Picture parameter set RBSP syntax

pic_parameter_set_rbsp( ) {	**Descriptor**
pps_pic_parameter_set_id	ue(v)
pps_seq_parameter_set_id	ue(v)
transform_skip_enabled_flag	u(1)
deblocking_filter_control_present_flag	u(1)
if(deblocking_filter_control_present_flag) {	
deblocking_filter_override_enabled_flag	u(1)
pps_deblocking_filter_disabled_flag	u(1)
if(!pps_deblocking_filter_disabled_flag) {	
pps_beta_offset_div2	se(v)
pps_tc_offset_div2	se(v)
}	
}	
rbsp_trailing_bits( )	
}	

7.3.2.3 End of sequence RBSP syntax

end_of_seq_rbsp( ) {	**Descriptor**
}	

7.3.2.4 End of bitstream RBSP syntax

end_of_bitstream_rbsp( ) {	**Descriptor**
}	

7.3.2.5 Slice layer RBSP syntax

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

7.3.2.6 RBSP slice trailing bits syntax

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

7.3.2.7 RBSP trailing bits syntax

rbsp_trailing_bits( ) {	Descriptor
rbsp_stop_one_bit /* equal to 1 */	f(1)
while(!byte_aligned( ))	
rbsp_alignment_zero_bit /* equal to 0 */	f(1)
}	

7.3.2.8 Byte alignment syntax

byte_alignment( ) {	Descriptor
alignment_bit_equal_to_one /* equal to 1 */	f(1)
while(!byte_aligned( ))	
alignment_bit_equal_to_zero /* equal to 0 */	f(1)
}	

7.3.3 Slice header syntax

7.3.3.1 General slice header syntax

slice_header( ) {	Descriptor
slice_pic_parameter_set_id	ue(v)
slice_address	u(v)
slice_type	ue(v)
if (slice_type != I) {	
log2_diff_ctu_max_bt_size	ue(v)
if(sps_sbtmvp_enabled_flag) {	
sbtmvp_size_override_flag	u(1)
if(sbtmvp_size_override_flag)	
log2_sbtmvp_active_size_minus2	u(3)
}	
if(sps_temporal_mvp_enabled_flag)	
slice_temporal_mvp_enabled_flag	u(1)
if(slice_type == B)	
mvd_l1_zero_flag	u(1)
if(slice_temporal_mvp_enabled_flag) {	
if(slice_type == B)	
collocated_from_l0_flag	u(1)

}	
six_minus_max_num_merge_cand	ue(v)
}	
if(sps_sao_enabled_flag) {	
slice_sao_luma_flag	u(1)
if(ChromaArrayType != 0)	
slice_sao_chroma_flag	u(1)
}	
if(sps_alf_enabled_flag) {	
slice_alf_enabled_flag	u(1)
if(slice_alf_enabled_flag)	
alf_data( )	
}	
dep_quant_enabled_flag	u(1)
if(!dep_quant_enabled_flag)	
sign_data_hiding_enabled_flag	u(1)
if(deblocking_filter_override_enabled_flag)	
deblocking_filter_override_flag	u(1)
if(deblocking_filter_override_flag) {	
slice_deblocking_filter_disabled_flag	u(1)
if(!slice_deblocking_filter_disabled_flag) {	
slice_beta_offset_div2	se(v)
slice_tc_offset_div2	se(v)
}	
}	
byte_alignment( )	
}	

7.3.3.2 Adaptive loop filter data syntax

alf_data( ) {	**Descriptor**
alf_chroma_idc	tu(v)
alf_luma_num_filters_signalled_minus1	tb(v)
if(alf_luma_num_filters_signalled_minus1 > 0) {	
for(filtIdx = 0; filtIdx < NumAlfFilters; filtIdx++)	
alf_luma_coeff_delta_idx[filtIdx]	tb(v)
}	
alf_luma_coeff_delta_flag	u(1)
if (!alf_luma_coeff_delta_flag && alf_luma_num_filters_signalled_minus1 > 0)	
alf_luma_coeff_delta_prediction_flag	u(1)
alf_luma_min_eg_order_minus1	ue(v)
for(i = 0; i < 3; i++)	
alf_luma_eg_order_increase_flag[i]	u(1)

if (alf_luma_coeff_delta_flag) {	
for(sigFiltIdx = 0; sigFiltIdx <= alf_luma_num_filters_signalled_minus1; sigFiltIdx++)	
alf_luma_coeff_flag[sigFiltIdx]	u(1)
}	
for(sigFiltIdx = 0; sigFiltIdx <= alf_luma_num_filters_signalled_minus1; sigFiltIdx++) {	
if (alf_luma_coeff_flag[sigFiltIdx]) {	
for (j = 0; j < 12; j++) {	
alf_luma_coeff_delta_abs[sigFiltIdx][j]	uek(v)
if(alf_luma_coeff_delta_abs[sigFiltIdx][j])	
alf_luma_coeff_delta_sign[sigFiltIdx][j]	u(1)
}	
}	
}	
if (alf_chroma_idc > 0) {	
alf_chroma_min_eg_order_minus1	ue(v)
for(i = 0; i < 2; i++)	
alf_chroma_eg_order_increase_flag[i]	u(1)
alf_chroma_ctb_present_flag	u(1)
for(j = 0; j < 6; j++) {	
alf_chroma_coeff_abs[j]	uek(v)
if(alf_chroma_coeff_abs[j] > 0)	
alf_chroma_coeff_sign[j]	u(1)
}	
}	
}	

7.3.4 Slice data syntax

7.3.4.1 General slice data syntax

slice_data( ) {	**Descriptor**
do {	
coding_tree_unit( )	
end_of_slice_flag	ae(v)
CtbAddrInRs++	
} while(!end_of_slice_flag)	
}	

7.3.4.2 Coding tree unit syntax

coding_tree_unit( ) {	Descriptor
xCtb = (CtbAddrInRs % PicWidthInCtbsY) << CtbLog2SizeY	
yCtb = (CtbAddrInRs / PicWidthInCtbsY) << CtbLog2SizeY	
if(slice_sao_luma_flag || slice_sao_chroma_flag)	
sao(xCtb >> CtbLog2SizeY, yCtb >> CtbLog2SizeY)	
if(slice_alf_enable_flag){	
alf_ctb_flag[0][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)
if(alf_chroma_ctb_present_flag) {	
if(alf_chroma_idc = = 1 || alf_chroma_idc = = 3)	
alf_ctb_flag[1][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)
if(alf_chroma_idc = = 2 || alf_chroma_idc = = 3)	
alf_ctb_flag[2][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)
}	
}	
if(slice_type = = I && qtbtt_dual_tree_intra_flag)	
dual_tree_implicit_qt_split (xCtb, yCtb, CtbLog2SizeY, 0)	
else	
coding_quadtree(xCtb, yCtb, CtbLog2SizeY, 0, SINGLE_TREE)	
}	

dual_tree_implicit_qt_split(x0, y0, log2CbSize, cqtDepth) {	Descriptor
if(log2CbSize > 6) {	
x1 = x0 + (1 << (log2CbSize − 1))	
y1 = y0 + (1 << (log2CbSize − 1))	
dual_tree_implicit_qt_split(x0, y0, log2CbSize − 1, cqtDepth + 1)	
if(x1 < pic_width_in_luma_samples)	
dual_tree_implicit_qt_split(x1, y0, log2CbSize − 1, cqtDepth + 1)	
if(y1 < pic_height_in_luma_samples)	
dual_tree_implicit_qt_split(x0, y1, log2CbSize − 1, cqtDepth + 1)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
dual_tree_implicit_qt_split(x1, y1, log2CbSize − 1, cqtDepth + 1)	
} else {	
coding_quadtree(x0, y0, log2CbSize, cqtDepth, DUAL_TREE_LUMA)	
coding_quadtree(x0, y0, log2CbSize, cqtDepth, DUAL_TREE_CHROMA)	
}	
}	

7.3.4.3 Sample adaptive offset syntax

sao(rx, ry) {	Descriptor
if(rx > 0) {	
leftCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − 1]]	
if(leftCtbInTile)	
sao_merge_left_flag	ae(v)
}	
if(ry > 0 && !sao_merge_left_flag) {	
upCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − PicWidthInCtbsY]]	
if(upCtbInTile)	
sao_merge_up_flag	ae(v)
}	
if(!sao_merge_up_flag && !sao_merge_left_flag)	
for(cIdx = 0; cIdx < (ChromaArrayType != 0 ? 3 : 1); cIdx++)	
if((slice_sao_luma_flag && cIdx == 0) || (slice_sao_chroma_flag && cIdx > 0)) {	
if(cIdx == 0)	
sao_type_idx_luma	ae(v)
else if(cIdx == 1)	
sao_type_idx_chroma	ae(v)
if(SaoTypeIdx[cIdx][rx][ry] != 0) {	
for(i = 0; i < 4; i++)	
sao_offset_abs[cIdx][rx][ry][i]	ae(v)
if(SaoTypeIdx[cIdx][rx][ry] == 1) {	
for(i = 0; i < 4; i++)	
if(sao_offset_abs[cIdx][rx][ry][i] != 0)	
sao_offset_sign[cIdx][rx][ry][i]	ae(v)
sao_band_position[cIdx][rx][ry]	ae(v)
} else {	
if(cIdx == 0)	
sao_eo_class_luma	ae(v)
if(cIdx == 1)	
sao_eo_class_chroma	ae(v)
}	
}	
}	
}	

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

7.3.4.4 Coding quadtree syntax

coding_quadtree(x0, y0, log2CbSize, cqtDepth, treeType) {	**Descriptor**
if((((x0 + (1 << log2CbSize) <= pic_width_in_luma_samples) ? 1 : 0) + ((y0 + (1 << log2CbSize) <= pic_height_in_luma_samples) ? 1 : 0) + (((1 << log2CbSize) <= MaxBtSizeY) ? 1 : 0)) >= 2 && log2CbSize > MinQtLog2SizeY)	
qt_split_cu_flag[x0][y0]	ae(v)
}	
if(qt_split_cu_flag[x0][y0]) {	
x1 = x0 + (1 << (log2CbSize − 1))	
y1 = y0 + (1 << (log2CbSize − 1))	
coding_quadtree(x0, y0, log2CbSize − 1, cqtDepth + 1, treeType)	
if(x1 < pic_width_in_luma_samples)	
coding_quadtree(x1, y0, log2CbSize − 1, cqtDepth + 1, treeType)	
if(y1 < pic_height_in_luma_samples)	
coding_quadtree(x0, y1, log2CbSize − 1, cqtDepth + 1, treeType)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
coding_quadtree(x1, y1, log2CbSize − 1, cqtDepth + 1, treeType)	
} else	
multi_type_tree(x0, y0, 1 << log2CbSize, 1 << log2CbSize, 0, 0, 0, treeType)	
}	

7.3.4.5 Multi-type tree syntax

multi_type_tree(x0, y0, cbWidth, cbHeight, mttDepth, depthOffset, partIdx, treeType) {	**Descriptor**
if((allowSplitBtVer || allowSplitBtHor || allowSplitTtVer || allowSplitTtHor) && (x0 + cbWidth <= pic_width_in_luma_samples) && (y0 + cbHeight <= pic_height_in_luma_samples))	
mtt_split_cu_flag	ae(v)
if(mtt_split_cu_flag) {	
if((allowSplitBtHor || allowSplitTtHor) && (allowSplitBtVer || allowSplitTtVer))	
mtt_split_cu_vertical_flag	ae(v)
if((allowSplitBtVer && allowSplitTtVer && mtt_split_cu_vertical_flag) || (allowSplitBtHor && allowSplitTtHor && !mtt_split_cu_vertical_flag))	
mtt_split_cu_binary_flag	ae(v)
if(MttSplitMode[x0][y0][mttDepth] == SPLIT_BT_VER) {	
depthOffset += (x0 + cbWidth > pic_width_in_luma_samples) ? 1 : 0	
x1 = x0 + (cbWidth / 2)	
multi_type_tree(x0, y0, cbWidth / 2, cbHeight, mttDepth + 1, depthOffset, 0, treeType)	
if(x1 < pic_width_in_luma_samples)	
multi_type_tree(x1, y0, cbWidth / 2, cbHeightY, mttDepth + 1, depthOffset, 1, treeType)	
} else if(MttSplitMode[x0][y0][mttDepth] == SPLIT_BT_HOR) {	
depthOffset += (y0 + cbHeight > pic_height_in_luma_samples) ? 1 : 0	
y1 = y0 + (cbHeight / 2)	
multi_type_tree(x0, y0, cbWidth, cbHeight / 2, mttDepth + 1, depthOffset, 0, treeType)	
if(y1 < pic_height_in_luma_samples)	
multi_type_tree(x0, y1, cbWidth, cbHeight / 2, mttDepth + 1, depthOffset, 1, treeType)	
} else if(MttSplitMode[x0][y0][mttDepth] == SPLIT_TT_VER) {	
x1 = x0 + (cbWidth / 4)	
x2 = x0 + (3 * cbWidth / 4)	
multi_type_tree(x0, y0, cbWidth / 4, cbHeight, mttDepth + 1, depthOffset, 0, treeType)	
multi_type_tree(x1, y0, cbWidth / 2, cbHeight, mttDepth + 1, depthOffset, 1, treeType)	
multi_type_tree(x2, y0, cbWidth / 4, cbHeight, mttDepth + 1, depthOffset, 2, treeType)	
} else { /* SPLIT_TT_HOR */	
y1 = y0 + (cbHeight / 4)	
y2 = y0 + (3 * cbHeight / 4)	
multi_type_tree(x0, y0, cbWidth, cbHeight / 4, mttDepth + 1, depthOffset, 0, treeType)	
multi_type_tree(x0, y1, cbWidth, cbHeight / 2, mttDepth + 1, depthOffset, 1, treeType)	
multi_type_tree(x0, y2, cbWidth, cbHeight / 4, mttDepth + 1, depthOffset, 2 , treeType)	
}	
} else	
coding_unit(x0, y0, cbWidth, cbHeight, treeType)	
}	

7.3.4.6 Coding unit syntax

coding_unit(x0, y0, cbWidth, cbHeight, treeType) {	**Descriptor**
if(slice_type != I) {	
cu_skip_flag[x0][y0]	ae(v)

if(cu_skip_flag[x0][y0] = = 0)	
pred_mode_flag	ae(v)
}	
if(CuPredMode[x0][y0] = = MODE_INTRA) {	
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA) {	
intra_luma_mpm_flag[x0][y0]	ae(v)
if(intra_luma_mpm_flag[x0][y0])	
intra_luma_mpm_idx[x0][y0]	ae(v)
else	
intra_luma_mpm_remainder[x0][y0]	ae(v)
}	
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA)	
intra_chroma_pred_mode[x0][y0]	ae(v)
} else { /* MODE_INTER */	
if(cu_skip_flag[x0][y0]) {	
if(sps_affine_enabled_flag && cbWidth >= 8 && cbHeight >= 8 && (MotionModelIdc[x0 − 1][y0 + cbHeight − 1] != 0 || MotionModelIdc[x0 − 1][y0 + cbHeight] != 0 || MotionModelIdc[x0 − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth][y0 − 1]] != 0))	
merge_affine_flag[x0][y0]	ae(v)
if(merge_affine_flag[x0][y0] = = 0 && MaxNumMergeCand > 1)	
merge_idx[x0][y0]	ae(v)
} else {	
merge_flag[x0][y0]	ae(v)
if(merge_flag[x0][y0]) {	
if(sps_affine_enabled_flag && cbWidth >= 8 && cbHeight >= 8 && (MotionModelIdc[x0 − 1][y0 + cbHeight − 1] != 0 || MotionModelIdc[x0 − 1][y0 + cbHeight] != 0 || MotionModelIdc[x0 − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth][y0 − 1]] != 0))	
merge_affine_flag[x0][y0]	ae(v)
if(merge_affine_flag[x0][y0] = = 0 && MaxNumMergeCand > 1)	
merge_idx[x0][y0]	ae(v)
} else {	
if(slice_type = = B)	
inter_pred_idc[x0][y0]	ae(v)
if(sps_affine_enabled_flag && cbWidth >= 16 && cbHeight >= 16) {	
inter_affine_flag[x0][y0]	ae(v)
if(sps_affine_type_flag && inter_affine_flag[x0][y0])	
cu_affine_type_flag[x0][y0]	ae(v)
}	
if(inter_pred_idc[x0][y0] != PRED_L1) {	
if(num_ref_idx_l0_active_minus1 > 0)	
ref_idx_l0[x0][y0]	ae(v)
mvd_coding(x0, y0, 0, 0)	
if(MotionModelIdc[x0][y0] > 0)	
mvd_coding(x0, y0, 0, 1)	
if(MotionModelIdc[x0][y0] > 1)	

mvd_coding(x0, y0, 0, 2)	
mvp_l0_flag[x0][y0]	ae(v)
} else {	
MvdL0[x0][y0][0] = 0	
MvdL0[x0][y0][1] = 0	
}	
if(inter_pred_idc[x0][y0] != PRED_L0) {	
if(num_ref_idx_l1_active_minus1 > 0)	
ref_idx_l1[x0][y0]	ae(v)
if(mvd_l1_zero_flag && inter_pred_idc[x0][y0] = = PRED_BI) {	
MvdL1[x0][y0][0] = 0	
MvdL1[x0][y0][1] = 0	
MvdCpL1[x0][y0][0][0] = 0	
MvdCpL1[x0][y0][0][1] = 0	
MvdCpL1[x0][y0][1][0] = 0	
MvdCpL1[x0][y0][1][1] = 0	
MvdCpL1[x0][y0][2][0] = 0	
MvdCpL1[x0][y0][2][1] = 0	
} else {	
mvd_coding(x0, y0, 1, 0)	
if(MotionModelIdc[x0][y0] > 0)	
mvd_coding(x0, y0, 1, 1)	
if(MotionModelIdc[x0][y0] > 1)	
mvd_coding(x0, y0, 1, 2)	
mvp_l1_flag[x0][y0]	ae(v)
} else {	
MvdL1[x0][y0][0] = 0	
MvdL1[x0][y0][1] = 0	
}	
if(sps_amvr_enabled_flag && inter_affine_flag = = 0 && (MvdL0[x0][y0][0] != 0 || MvdL0[x0][y0][1] != 0 || MvdL1[x0][y0][0] != 0 || MvdL1[x0][y0][1] != 0))	
amvr_mode[x0][y0]	ae(v)
}	
}	
}	
if(CuPredMode[x0][y0] != MODE_INTRA && cu_skip_flag[x0][y0] = = 0)	
cu_cbf	ae(v)
if(cu_cbf)	
transform_tree(x0, y0, cbWidth, cbHeight, treeType)	
}	

7.3.4.7 Motion vector difference syntax

mvd_coding(x0, y0, refList ,cpIdx) {	Descriptor
abs_mvd_greater0_flag[0]	ae(v)
abs_mvd_greater0_flag[1]	ae(v)
if(abs_mvd_greater0_flag[0])	
abs_mvd_greater1_flag[0]	ae(v)
if(abs_mvd_greater0_flag[1])	
abs_mvd_greater1_flag[1]	ae(v)
if(abs_mvd_greater0_flag[0]) {	
if(abs_mvd_greater1_flag[0])	
abs_mvd_minus2[0]	ae(v)
mvd_sign_flag[0]	ae(v)
}	
if(abs_mvd_greater0_flag[1]) {	
if(abs_mvd_greater1_flag[1])	
abs_mvd_minus2[1]	ae(v)
mvd_sign_flag[1]	ae(v)
}	
}	

7.3.4.8 Transform tree syntax

transform_tree(x0, y0, tbWidth, tbHeight , treeType) {	Descriptor
if(tbWidth > MaxTbSizeY || tbHeight > MaxTbSizeY) {	
trafoWidth = (tbWidth > MaxTbSizeY) ? (tbWidth / 2) : tbWidth	
trafoHeight = (tbHeight > MaxTbSizeY) ? (tbHeight / 2) : tbHeight	
transform_tree(x0, y0, trafoWidth, trafoHeight)	
if(tbWidth > MaxTbSizeY)	
transform_tree(x0 + trafoWidth, y0, trafoWidth, trafoHeight, treeType)	
if(tbHeight > MaxTbSizeY)	
transform_tree(x0, y0 + trafoHeight, trafoWidth, trafoHeight, treeType)	
if(tbWidth > MaxTbSizeY && tbHeight > MaxTbSizeY)	
transform_tree(x0 + trafoWidth, y0 + trafoHeight, trafoWidth, trafoHeight, treeType)	
} else {	
transform_unit(x0, y0, tbWidth, tbHeight, treeType)	
}	
}	

7.3.4.9 Transform unit syntax

transform_unit(x0, y0, tbWidth, tbHeight, treeType) {	Descriptor
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA)	
tu_cbf_luma[x0][y0]	ae(v)
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA) {	

tu_cbf_cb[x0][y0]	ae(v)
tu_cbf_cr[x0][y0]	ae(v)
}	
if((((CuPredMode[x0][y0] = = MODE_INTRA) && sps_mts_intra_enabled_flag) || ((CuPredMode[x0][y0] = = MODE_INTER) && sps_mts_inter_enabled_flag)) && tu_cbf_luma[x0][y0] && treeType != DUAL_TREE_CHROMA && (tbWidth <= 32) && (tbHeight <= 32))	
tu_mts_flag[x0][y0]	ae(v)
if(tu_cbf_luma[x0][y0])	
residual_coding(x0, y0, log2(tbWidth), log2(tbHeight), 0)	
if(tu_cbf_cb[x0][y0])	
residual_coding(x0, y0, log2(tbWidth / 2), log2(tbHeight / 2), 1)	
if(tu_cbf_cr[x0][y0])	
residual_coding(x0, y0, log2(tbWidth / 2), log2(tbHeight / 2), 2)	
}	

7.3.4.10 Residual coding syntax

residual_coding(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	**Descriptor**
if(transform_skip_enabled_flag && (cIdx ! = 0 || tu_mts_flag[x0][y0] = = 0) && (log2TbWidth <= 2) && (log2TbHeight <= 2))	
transform_skip_flag[x0][y0][cIdx]	ae(v)
last_sig_coeff_x_prefix	ae(v)
last_sig_coeff_y_prefix	ae(v)
if(last_sig_coeff_x_prefix > 3)	
last_sig_coeff_x_suffix	ae(v)
if(last_sig_coeff_y_prefix > 3)	
last_sig_coeff_y_suffix	ae(v)
log2SbSize = (Min(log2TbWidth, log2TbHeight) < 2 ? 1 : 2)	
numSbCoeff = 1 << (log2SbSize << 1)	
lastScanPos = numSbCoeff	
lastSubBlock = (1 << (log2TbWidth + log2TbHeight − 2 * log2SbSize)) − 1	
do {	
if(lastScanPos = = 0) {	
lastScanPos = numSbCoeff	
lastSubBlock− −	
}	
lastScanPos− −	
xS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize][lastSubBlock][0]	
yS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize][lastSubBlock][1]	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][lastScanPos][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][lastScanPos][1]	
} while((xC != LastSignificantCoeffX) || (yC != LastSignificantCoeffY))	
numSigCoeff = 0	
QState = 0	
for(i = lastSubBlock; i >= 0; i− −) {	

startQStateSb = QState	
xS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize] [lastSubBlock][0]	
yS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize] [lastSubBlock][1]	
inferSbDcSigCoeffFlag = 0	
if((i < lastSubBlock) && (i > 0)) {	
coded_sub_block_flag[xS][yS]	ae(v)
inferSbDcSigCoeffFlag = 1	
}	
firstSigScanPosSb = numSbCoeff	
lastSigScanPosSb = −1	
for(n = (i == lastSubBlock) ? lastScanPos − 1 : numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(coded_sub_block_flag[xS][yS] && (n > 0 || !inferSbDcSigCoeffFlag)) {	
sig_coeff_flag[xC][yC]	ae(v)
if(sig_coeff_flag[xC][yC])	
inferSbDcSigCoeffFlag = 0	
}	
if(sig_coeff_flag[xC][yC]) {	
numSigCoeff++	
par_level_flag[n]	ae(v)
rem_abs_gt1_flag[n]	ae(v)
if(lastSigScanPosSb == −1)	
lastSigScanPosSb = n	
firstSigScanPosSb = n	
}	
AbsLevelPass1[xC][yC] = sig_coeff_flag[xC][yC] + par_level_flag[n] + 2 * rem_abs_gt1_flag[n]	
if(dep_quant_enabled_flag)	
QState = QStateTransTable[QState][par_level_flag[n]]	
}	
for(n = numSbCoeff − 1; n >= 0; n− −)	
if(rem_abs_gt1_flag[n])	
rem_abs_gt2_flag[n]	ae(v)
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(rem_abs_gt2_flag[n])	
abs_remainder[n]	
AbsLevel[xC][yC] = AbsLevelPass1[xC][yC] + 2 * (rem_abs_gt2_flag[n] + abs_remainder[n])	
}	
if(dep_quant_enabled_flag || !sign_data_hiding_enabled_flag)	
signHidden = 0	
else	
signHidden = (lastSigScanPosSb − firstSigScanPosSb > 3 ? 1 : 0)	

for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(sig_coeff_flag[xC][yC] && (!signHidden || (n != firstSigScanPosSb)))	
coeff_sign_flag[n]	ae(v)
}	
if(dep_quant_enabled_flag) {	
QState = startQStateSb	
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(sig_coeff_flag[xC][yC])	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = (2 * AbsLevel[xC][yC] − (QState > 1 ? 1 : 0)) * (1 − 2 * coeff_sign_flag[n])	
QState = QStateTransTable[QState][par_level_flag[n]]	
} else {	
sumAbsLevel = 0	
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(sig_coeff_flag[xC][yC]) {	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = AbsLevel[xC][yC] * (1 − 2 * coeff_sign_flag[n])	
if(signHidden) {	
sumAbsLevel += AbsLevel[xC][yC]	
if((n == firstSigScanPosSb) && (sumAbsLevel % 2) == 1))	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = −TransCoeffLevel[x0][y0][cIdx][xC][yC]	
}	
}	
}	
}	
}	
if(tu[x0][y0] && (cIdx == 0) && ((CuPredMode[x0][y0] == MODE_INTRA && numSigCoeff > 2) || (CuPredMode[x0][y0] == MODE_INTER)))	
mts_idx[x0][y0][cIdx	ae(v)
}	

7.4 Semantics

7.4.1 General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this clause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this Specification.

7.4.2 NAL unit semantics

7.4.2.1 General NAL unit semantics

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

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

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

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

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

– Otherwise, the RBSP contains the SODB as follows:

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

2) rbsp_trailing_bits( ) are present after the SODB as follows:

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

   ii) The next bit consists of a single rbsp_stop_one_bit equal to 1.

   iii) When the rbsp_stop_one_bit is not the last bit of a byte-aligned byte, one or more rbsp_alignment_zero_bit is present to result in byte alignment.

3) One or more cabac_zero_word 16-bit syntax elements equal to 0x0000 may be present in some RBSPs after the rbsp_trailing_bits( ) at the end of the RBSP.

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

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

emulation_prevention_three_byte is a byte equal to 0x03. When an emulation_prevention_three_byte is present in the NAL unit, it shall be discarded by the decoding process.

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

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

– 0x000000

– 0x000001

– 0x000002

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

– 0x00000300

– 0x00000301

– 0x00000302

– 0x00000303

7.4.2.2 NAL unit header semantics

forbidden_zero_bit shall be equal to 0.

nal_unit_type specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7-1.

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

nal_unit_type	Name of nal_unit_type	Content of NAL unit and RBSP syntax structure	NAL unit type class
0	NON_IRAP_NUT	Coded slice segment of a non-IRAP picture slice_layer_rbsp( )	VCL
1	IRAP_NUT	Coded slice of an IRAP picture slice_layer_rbsp( )	VCL
2-15	RSV_VCL_NUT	Reserved VCL NAL Units	VCL
16	SPS_NUT	Sequence parameter set seq_parameter_set_rbsp( )	non-VCL
17	PPS_NUT	Picture parameter set pic_parameter_set_rbsp( )	non-VCL
18	EOS_NUT	End of sequence end_of_seq_rbsp( )	non-VCL
19	EOB_NUT	End of bitstream end_of_bitstream_rbsp( )	non-VCL
20, 21	PREFIX_SEI_NUT SUFFIX_SEI_NUT	Supplemental enhancement information sei_rbsp( )	non-VCL
22-26	RSV_NVCL	Reserved	non-VCL
27-31	UNSPEC	Unspecified	non-VCL

nuh_temporal_id_plus1 minus 1 specifies a temporal identifier for the NAL unit. The value of nuh_temporal_id_plus1 shall not be equal to 0.

The variable TemporalId is derived as follows:

$$\text{TemporalId} = \text{nuh\_temporal\_id\_plus1} - 1 \tag{7-1}$$

When nal_unit_type is equal to IRAP_NUT, the coded slice belongs to an IRAP picture, TemporalId shall be equal to 0.

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

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

– If nal_unit_type is equal to SPS_NUT, TemporalId shall be equal to 0 and the TemporalId of the access unit containing the NAL unit shall be equal to 0.

– Otherwise, if nal_unit_type is equal to EOS_NUT or EOB_NUT, TemporalId shall be equal to 0.

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

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

nuh_reserved_zero_7bits shall be equal to '0000000'. Other values of nuh_reserved_zero_7bits may be specified in the future by ITU-T | ISO/IEC. Decoders shall ignore (i.e. remove from the bitstream and discard) NAL units with values of nuh_reserved_zero_7bits not equal to '0000000'.

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

7.4.3.1 Sequence parameter set RBSP semantics

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

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

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

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

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

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

– Otherwise (separate_colour_plane_flag is equal to 1), ChromaArrayType is set equal to 0.

pic_width_in_luma_samples specifies the width of each decoded picture in units of luma samples. pic_width_in_luma_samples shall not be equal to 0 and shall be an integer multiple of MinCbSizeY.

pic_height_in_luma_samples specifies the height of each decoded picture in units of luma samples. pic_height_in_luma_samples shall not be equal to 0 and shall be an integer multiple of MinCbSizeY.

bit_depth_luma_minus8 specifies the bit depth of the samples of the luma array $BitDepth_Y$ and the value of the luma quantization parameter range offset $QpBdOffset_Y$ as follows:

$$BitDepth_Y = 8 + bit\_depth\_luma\_minus8 \quad (7\text{-}2)$$

$$QpBdOffset_Y = 6 * bit\_depth\_luma\_minus8 \quad (7\text{-}3)$$

bit_depth_luma_minus8 shall be in the range of 0 to 8, inclusive.

bit_depth_chroma_minus8 specifies the bit depth of the samples of the chroma arrays $BitDepth_C$ and the value of the chroma quantization parameter range offset $QpBdOffset_C$ as follows:

$$BitDepth_C = 8 + bit\_depth\_chroma\_minus8 \quad (7\text{-}4)$$

$$QpBdOffset_C = 6 * bit\_depth\_chroma\_minus8 \quad (7\text{-}5)$$

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

qtbtt_dual_tree_intra_flag equal to 1 specifies that for I slices, each CTU is split into coding units with 64x64 luma samples using an implicit quadtree split and that these coding units are the root of two separate coding_quadtree syntax structure for luma and chroma.

log2_ctu_size_minus2 plus 2 specifies the luma coding tree block size of each CTU.

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

$$CtbLog2SizeY = log2\_ctu\_size\_minus2 + 2 \quad (7\text{-}6)$$

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

$$MinCbLog2SizeY = 2 \quad (7\text{-}8)$$

$$MinCbSizeY = 1 \ll MinCbLog2SizeY \quad (7\text{-}9)$$

$$MinTbSizeY = 4 \quad (7\text{-}10)$$

$$MaxTbSizeY = 64 \quad (7\text{-}11)$$

PicWidthInCtbsY = Ceil(pic_width_in_luma_samples ÷ CtbSizeY) (7-12)

PicHeightInCtbsY = Ceil(pic_height_in_luma_samples ÷ CtbSizeY) (7-13)

PicSizeInCtbsY = PicWidthInCtbsY * PicHeightInCtbsY (7-14)

PicWidthInMinCbsY = pic_width_in_luma_samples / MinCbSizeY (7-15)

PicHeightInMinCbsY = pic_height_in_luma_samples / MinCbSizeY (7-16)

PicSizeInMinCbsY = PicWidthInMinCbsY * PicHeightInMinCbsY (7-17)

PicSizeInSamplesY = pic_width_in_luma_samples * pic_height_in_luma_samples (7-18)

PicWidthInSamplesC = pic_width_in_luma_samples / SubWidthC (7-19)

PicHeightInSamplesC = pic_height_in_luma_samples / SubHeightC (7-20)

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

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

– If chroma_format_idc is equal to 0 (monochrome) or separate_colour_plane_flag is equal to 1, CtbWidthC and CtbHeightC are both equal to 0.

– Otherwise, CtbWidthC and CtbHeightC are derived as follows:

CtbWidthC = CtbSizeY / SubWidthC (7-21)

CtbHeightC = CtbSizeY / SubHeightC (7-22)

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

log2_min_qt_size_intra_slices_minus2 plus 2 specifies the minimum luma size of a leaf block resulting from quadtree splitting of a CTU in slices with slice_type equal to 2 (I). The value of log2_min_qt_size_intra_slices_minus2 shall be in the range of 0 to CtbLog2SizeY − 2, inclusive.

MinQtLog2SizeIntraY = log2_min_qt_size_intra_slices_minus2 + 2 (7-23)

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

log2_min_qt_size_inter_slices_minus2 plus 2 specifies the minimum luma size of a leaf block resulting from quadtree splitting of a CTU in slices with slice_type equal to 0 (B) or 1 (P). The value of log2_min_qt_size_inter_slices_minus2 shall be in the range of 0 to CtbLog2SizeY − 2, inclusive.

MinQtLog2SizeInterY = log2_min_qt_size_inter_slices_minus2 + 2 (7-24)

max_mtt_hierarchy_depth_inter_slices specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice_type equal to 0 (B) or 1 (P). The value of max_mtt_hierarchy_depth_inter_slices shall be in the range of 0 to CtbLog2SizeY − MinTbLog2SizeY, inclusive.

max_mtt_hierarchy_depth_intra_slices specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice_type equal to 2 (I). The value of max_mtt_hierarchy_depth_intra_slices shall be in the range of 0 to CtbLog2SizeY − MinTbLog2SizeY, inclusive.

sps_cclm_enabled_flag equal to 0 specifies that the cross-component linear model intra prediction from luma component to chroma component is disabled. sps_cclm_enabled_flag equal to 1 specifies that the cross-component linear model intra prediction from luma component to chroma componenent is enabled.

sps_sao_enabled_flag equal to 1 specifies that the sample adaptive offset process is applied to the reconstructed picture after the deblocking filter process. sps_sao_enabled_flag equal to 0 specifies that the sample adaptive offset process is not applied to the reconstructed picture after the deblocking filter process.

sps_alf_enabled_flag equal to 0 specifies that the adaptive loop filter is disabled. sps_alf_enabled_flag equal to 1 specifies that the adaptive loop filter is enabled.

sps_temporal_mvp_enabled_flag equal to 1 specifies that slice_temporal_mvp_enabled_flag is present in the slice headers of slices with slice_type not equal to I in the CVS. sps_temporal_mvp_enabled_flag equal to 0 specifies that slice_temporal_mvp_enabled_flag is not present in slice headers and that temporal motion vector predictors are not used in the CVS.

sps_sbtmvp_enabled_flag equal to 1 specifies that subblock-based temporal motion vector predictors may be used in decoding of pictures with all slices having slice_type not equal to I in the CVS. sps_sbtmvp_enabled_flag equal to 0 specifies that subblock-based temporal motion vector predictors are not used in the CVS. When sps_sbtmvp_enabled_flag is not present, it is inferred to be equal to 0

log2_sbtmvp_default_size_minus2 specifies the inferred value of the syntax element log2_sbtmvp_active_size_minus2 in the slice headers of slices with slice_type not equal to I in the CVS when slice_sbtmvp_size_override_flag is equal to 0.

sps_amvr_enabled_flag equal to 1 specifies that adaptive motion vector difference resolution is used in motion vector coding. amvr_enabled_flag equal to 0 specifies that adaptive motion vector difference resolution is not used in motion vector coding.

sps_affine_enabled_flag specifies whether affine model based motion compensation can be used for inter prediction. If sps_affine_enabled_flag is equal to 0, the syntax shall be constrained such that no affine model based motion compensation is used in the CVS, and merge_affine_flag, inter_affine_flag and cu_affine_type_flag are not present in coding unit syntax of the CVS. Otherwise (sps_affine_enabled_flag is equal to 1), affine model based motion compensation can be used in the CVS.

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

sps_mts_intra_enabled_flag equal to 1 specifies that tu_mts_flag may be present in the residual coding syntax for intra coding units. sps_mts_intra_enabled_flag equal to 0 specifies that tu_mts_flag is not present in the residual coding syntax for intra coding units.

sps_mts_inter_enabled_flag specifies that tu_mts_flag may be present in the residual coding syntax for inter coding units. sps_mts_inter_enabled_flag equal to 0 specifies that tu_mts_flag is not present in the residual coding syntax for inter coding units.

sps_ladf_enabled_flag equal to 1, specifies that sps_num_ladf_intervals_minus2, sps_ladf_lowest_interval_qp_offset, sps_ladf_qp_offset[i], and sps_ladf_delta_threshold_minus1[i] are present in the SPS.

sps_num_ladf_intervals_minus2 plus 1 specifies the number of sps_ladf_delta_threshold_minus1[i] and sps_ladf_qp_offset[i] syntax elements that are present in the SPS. The value of sps_num_ladf_intervals_minus2 shall be in the range of 0 to 3, inclusive.

sps_ladf_lowest_interval_qp_offset specifies the offset used to derive the variable qP as specified in clause 8.5.2.6.3. The value of sps_ladf_lowest_interval_qp_offset shall be in the range of [TBD: min QP value] to [TBD: max QP value], inclusive.

sps_ladf_qp_offset[i] specifies the offset array used to derive the variable qP as specified in clause 8.5.2.6.3. The value of sps_ladf_qp_offset[i] shall be in the range of [TBD: min QP value] to [TBD: max QP value], inclusive.

sps_ladf_delta_threshold_minus1[i] is used to compute the values of SpsLadfIntervalLowerBound[i], which specifies the lower bound of the i-th luma intensity level interval. The value of sps_ladf_delta_threshold_minus1[i] shall be in the range of 0 to $2^{\mathrm{BitDepthY}} - 3$, inclusive.

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

For each value of i in the range of 0 to sps_num_ladf_intervals_minus2, inclusive, the variable SpsLadfIntervalLowerBound[i + 1] is derived as follows:

$$\begin{aligned} \mathrm{SpsLadfIntervalLowerBound}[\,i+1\,] = {} & \mathrm{SpsLadfIntervalLowerBound}[\,i\,] \\ & + \mathrm{sps\_ladf\_delta\_threshold\_minus1}[\,i\,] + 1 \end{aligned} \tag{7-25}$$

[Ed. (BB): Update highlighted parts when QP signalling proposals are integrated.]

7.4.3.2 Picture parameter set RBSP semantics

pps_pic_parameter_set_id identifies the PPS for reference by other syntax elements. The value of pps_pic_parameter_set_id shall be in the range of 0 to 63, inclusive.

pps_seq_parameter_set_id specifies the value of sps_seq_parameter_set_id for the active SPS. The value of pps_seq_parameter_set_id shall be in the range of 0 to 15, inclusive.

transform_skip_enabled_flag equal to 1 specifies that transform_skip_flag may be present in the residual coding syntax. transform_skip_enabled_flag equal to 0 specifies that transform_skip_flag is not present in the residual coding syntax.

deblocking_filter_control_present_flag equal to 1 specifies the presence of deblocking filter control syntax elements in the PPS. deblocking_filter_control_present_flag equal to 0 specifies the absence of deblocking filter control syntax elements in the PPS.

deblocking_filter_override_enabled_flag equal to 1 specifies the presence of deblocking_filter_override_flag in the slice headers for pictures referring to the PPS. deblocking_filter_override_enabled_flag equal to 0 specifies the absence of deblocking_filter_override_flag in the slice headers for pictures referring to the PPS. When not present, the value of deblocking_filter_override_enabled_flag is inferred to be equal to 0.

pps_deblocking_filter_disabled_flag equal to 1 specifies that the operation of deblocking filter is not applied for slices referring to the PPS in which slice_deblocking_filter_disabled_flag is not present. pps_deblocking_filter_disabled_flag equal to 0 specifies that the operation of the deblocking filter is applied for slices referring to the PPS in which slice_deblocking_filter_disabled_flag is not present. When not present, the value of pps_deblocking_filter_disabled_flag is inferred to be equal to 0.

pps_beta_offset_div2 and **pps_tc_offset_div2** specify the default deblocking parameter offsets for β and tC (divided by 2) that are applied for slices referring to the PPS, unless the default deblocking parameter offsets are overridden by the deblocking parameter offsets present in the slice headers of the slices referring to the PPS. The values of pps_beta_offset_div2 and pps_tc_offset_div2 shall both be in the range of −6 to 6, inclusive. When not present, the value of pps_beta_offset_div2 and pps_tc_offset_div2 are inferred to be equal to 0.

7.4.3.3 End of sequence RBSP semantics

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

7.4.3.4 End of bitstream RBSP semantics

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

7.4.3.5 Slice layer RBSP semantics

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

7.4.3.6 RBSP slice trailing bits semantics

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

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

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

Let the variable RawMinCuBits be derived as follows:

$$RawMinCuBits = MinCbSizeY * MinCbSizeY * (BitDepth_Y + 2 * BitDepth_C / (SubWidthC * SubHeightC)) \quad (7\text{-}26)$$

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

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

7.4.3.7 RBSP trailing bits semantics

rbsp_stop_one_bit shall be equal to 1.

rbsp_alignment_zero_bit shall be equal to 0.

7.4.3.8 Byte alignment semantics

alignment_bit_equal_to_one shall be equal to 1.

alignment_bit_equal_to_zero shall be equal to 0.

7.4.4 Slice header semantics

7.4.4.1 General slice header semantics

When present, the value of the slice header syntax element slice_pic_parameter_set_id shall be the same in all slice headers of a coded picture.

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

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

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

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

Table 7-2 – Name association to slice_type

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

When nal_unit_type is equal to IRAP_NUT, i.e., the picture is an IRAP picture, slice_type shall be equal to 2.

log2_diff_ctu_max_bt_size specifies the difference between the luma CTB size and the maximum luma size (width or height) of a coding block that can be split using a binary split. The value of log2_diff_ctu_max_bt_size shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

When log2_diff_ctu_max_bt_size is not present, the value of log2_diff_ctu_max_bt_size is inferred to be equal to 2.

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

$$MinQtLog2SizeY = (slice\_type == I) ? MinQtLog2SizeIntraY : MinQtLog2SizeInterY \quad (7\text{-}27)$$

$$MaxBtLog2SizeY = CtbLog2SizeY - log2\_diff\_ctu\_max\_bt\_size \quad (7\text{-}28)$$

$$MinBtLog2SizeY = MinCbLog2SizeY \quad (7\text{-}29)$$

$$MaxTtLog2SizeY = (slice\_type == I) ? 5 : 6 \quad (7\text{-}30)$$

$$MinTtLog2SizeY = MinCbLog2SizeY \quad (7\text{-}31)$$

$$MinQtSizeY = 1 << MinQtLog2SizeY \quad (7\text{-}32)$$

$$MaxBtSizeY = 1 << MaxBtLog2SizeY \quad (7\text{-}33)$$

$$MinBtSizeY = 1 << MinBtLog2SizeY \quad (7\text{-}34)$$

$$MaxTtSizeY = 1 << MaxTtLog2SizeY \quad (7\text{-}35)$$

$$MinTtSizeY = 1 << MinTtLog2SizeY \quad (7\text{-}36)$$

MaxMttDepth = (slice_type == I) ? max_mtt_hierarchy_depth_intra_slices : max_mtt_hierarchy_depth_inter_slices (7-37)

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

sbtmvp_size_override_flag equal to 1 specifies that the syntax element log2_sbtmvp_active_size_minus2 is present for the current slice. sbtmvp_size_override_flag equal to 0 specifies that the syntax element log2_atmvp_active_size_minus2 is not present and log2_sbtmvp_size_active_minus2 is inferred to be equal to log2_sbtmvp_default_size_minus2.

log2_sbtmvp_active_size_minus2 plus 2 specifies the value of the subblock size that is used for deriving the motion parameters for the subblock-based TMVP of the current slice. When log2_sbtmvp_size_active_minus2 is is not present, it is inferred to be equal to log2_sbtmvp_default_size_minus2. The variable is derived as follows:

Log2SbtmvpSize = log2_sbtmvp_size_active_minus2 + 2 (7-38)

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

mvd_l1_zero_flag equal to 1 indicates that the mvd_coding(x0, y0, 1) syntax structure is not parsed and MvdL1[x0][y0][compIdx] is set equal to 0 for compIdx = 0..1. mvd_l1_zero_flag equal to 0 indicates that the mvd_coding(x0, y0, 1) syntax structure is parsed.

collocated_from_l0_flag equal to 1 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 0. collocated_from_l0_flag equal to 0 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 1. When collocated_from_l0_flag is not present, it is inferred to be equal to 1.

six_minus_max_num_merge_cand specifies the maximum number of merging motion vector prediction (MVP) candidates supported in the slice subtracted from 6. The maximum number of merging MVP candidates, MaxNumMergeCand is derived as follows:

MaxNumMergeCand = 6 − six_minus_max_num_merge_cand (7-39)

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

slice_sao_luma_flag equal to 1 specifies that SAO is enabled for the luma component in the current slice; slice_sao_luma_flag equal to 0 specifies that SAO is disabled for the luma component in the current slice. When slice_sao_luma_flag is not present, it is inferred to be equal to 0.

slice_sao_chroma_flag equal to 1 specifies that SAO is enabled for the chroma component in the current slice; slice_sao_chroma_flag equal to 0 specifies that SAO is disabled for the chroma component in the current slice. When slice_sao_chroma_flag is not present, it is inferred to be equal to 0.

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

dep_quant_enabled_flag equal to 0 specifies that dependent quantization is disabled. dep_quant_enabled_flag equal to 1 specifies that dependent quantization is enabled.

sign_data_hiding_enabled_flag equal to 0 specifies that sign bit hiding is disabled. sign_data_hiding_enabled_flag equal to 1 specifies that sign bit hiding is enabled. When sign_data_hiding_enabled_flag is not present, it is inferred to be equal to 0.

deblocking_filter_override_flag equal to 1 specifies that deblocking parameters are present in the slice header. deblocking_filter_override_flag equal to 0 specifies that deblocking parameters are not present in the slice header. When not present, the value of deblocking_filter_override_flag is inferred to be equal to 0.

slice_deblocking_filter_disabled_flag equal to 1 specifies that the operation of the deblocking filter is not applied for the current slice. slice_deblocking_filter_disabled_flag equal to 0 specifies that the operation of the deblocking filter is applied for the current slice. When slice_deblocking_filter_disabled_flag is not present, it is inferred to be equal to pps_deblocking_filter_disabled_flag.

slice_beta_offset_div2 and **slice_tc_offset_div2** specify the deblocking parameter offsets for β and tC (divided by 2) for the current slice. The values of slice_beta_offset_div2 and slice_tc_offset_div2 shall both be in the range of −6 to 6, inclusive. When not present, the values of slice_beta_offset_div2 and slice_tc_offset_div2 are inferred to be equal to pps_beta_offset_div2 and pps_tc_offset_div2, respectively.

7.4.4.2 Adaptive loop filter data semantics

alf_chroma_idc equal to 0 specifies that the adaptive loop filter is not applied to Cb and Cr colour components. alf_chroma_idc equal to 1 indicates that the adaptive loop filter is applied to the Cb colour component. alf_chroma_idc equal to 2 indicates that the adaptive loop filter is applied to the Cr colour component. alf_chroma_idc equal to 3 indicates that the adaptive loop filter is applied to Cb and Cr colour components.

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

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

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

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

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

The maximum value maxVal of the truncated binary binarization tb(v) is set equal to alf_luma_num_filters_signalled_minus1.

alf_luma_coeff_delta_flag equal to 1 indicates that alf_luma_coeff_delta_prediction_flag is not signalled. alf_luma_coeff_delta_flag equal to 0 indicates that alf_luma_coeff_delta_prediction_flag may be signalled.

alf_luma_coeff_delta_prediction_flag equal to 1 specifies that the signalled luma filter coefficient deltas are predicted from the deltas of the previous luma coefficients. alf_luma_coeff_delta_prediction_flag equal to 0 specifies that the signalled luma filter coefficient deltas are not predicted from the deltas of the previous luma coefficients. When not present, alf_luma_coeff_delta_prediction_flag is inferred to be equal to 0.

alf_luma_min_eg_order_minus1 plus 1 specifies the minimum order of the exp-Golomb code for luma filter coefficient signalling. The value of alf_luma_min_eg_order_minus1 shall be in the range of 0 to 6, inclusive.

alf_luma_eg_order_increase_flag[i] equal to 1 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is incremented by 1. alf_luma_eg_order_increase_flag[i] equal to 0 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is not incremented by 1.

The order expGoOrderY[i] of the exp-Golomb code used to decode the values of alf_luma_coeff_delta_abs[sigFiltIdx][j] is derived as follows:

expGoOrderY[i] = alf_luma_min_eg_order_minus1 + 1+ alf_luma_eg_order_increase_flag[i] (7-40)

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

alf_luma_coeff_delta_abs[sigFiltIdx][j] specifies the absolute value of the j-th coefficient delta of the signalled luma filter indicated by sigFiltIdx. When alf_luma_coeff_delta_abs[sigFiltIdx][j] is not present, it is inferred to be equal 0.

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

golombOrderIdxY[ ] = { 0, 0, 1, 0, 0, 1, 2, 1, 0, 0, 1, 2 } (7-41)

k = expGoOrderY[golombOrderIdxY[j]] (7-42)

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

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

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

When alf_luma_coeff_delta_sign[sigFiltIdx][j] is not present, it is inferred to be equal to 0.

The variable filterCoefficients[sigFiltIdx][j] with sigFiltIdx = 0..alf_luma_num_filters_signalled_minus1, j = 0..11 is initialized as follows:

filterCoefficients[sigFiltIdx][j] = alf_luma_coeff_delta_abs[sigFiltIdx][j] * (1 − 2 * alf_luma_coeff_delta_sign[sigFiltIdx][j]) (7-43)

When alf_luma_coeff_delta_prediction_flag is equal 1, filterCoefficients[sigFiltIdx][j] with sigFiltIdx = 1..alf_luma_num_filters_signalled_minus1 and j = 0..11 are modified as follows:

filterCoefficients[sigFiltIdx][j] += filterCoefficients[sigFiltIdx − 1][j] (7-44)

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

$AlfCoeff_L$[filtIdx][j] = filterCoefficients[alf_luma_coeff_delta_idx[filtIdx]][j] (7-45)

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

$AlfCoeff_L$[filtIdx][12] = 128 − Σ_k ($AlfCoeff_L$[filtIdx][k] << 1), with k = 0..11 (7-46)

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

alf_chroma_min_eg_order_minus1 plus 1 specifies the minimum order of the exp-Golomb code for chroma filter coefficient signalling. The value of alf_chroma_min_eg_order_minus1 shall be in the range of 0 to 6, inclusive.

alf_chroma_eg_order_increase_flag[i] equal to 1 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is incremented by 1. alf_chroma_eg_order_increase_flag[i] equal to 0 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is not incremented by 1

The order expGoOrderC[i] of the exp-Golomb code used to decode the values of alf_chroma_coeff_abs[j] is derived as follows:

expGoOrderC[i] = alf_chroma_min_eg_order_minus1 + 1+ alf_chroma_eg_order_increase_flag[i] (7-47)

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

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

alf_chroma_coeff_abs[j] specifies the absolute value of the j-th chroma filter coefficient. When alf_chroma_coeff_abs[j] is not present, it is inferred to be equal 0. It is a requirement of bitstream conformance that the values of alf_chroma_coeff_abs[j] shall be in the range of 0 to $2^7 − 1$, inclusive.

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

golombOrderIdxC[ ] = { 0, 0, 1, 0, 0, 1 } (7-48)

k = expGoOrderC[golombOrderIdxC[j]] (7-49)

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

– If alf_chroma_coeff_sign[j] is equal to 0, the corresponding chroma filter coefficient has a positive value.

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

When alf_chroma_coeff_sign[j] is not present, it is inferred to be equal to 0.

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

$AlfCoeff_C$[j] = alf_chroma_coeff_abs[j] * (1 − 2 * alf_chroma_coeff_sign[j]) (7-50)

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

$$AlfCoeff_C[\ 6\ ] = 128 - \Sigma_k\ (\ AlfCoeff_C[\ k\ ]\ <<\ 1\ ),\ \text{with}\ k = 0..5 \tag{7-51}$$

It is a requirement of bitstream conformance that the values of AlfCoeff$_C$[j] with j = 0..5 shall be in the range of -2^7 to $2^7 - 1$, inclusive and that the values of AlfCoeff$_C$[6] shall be in the range of 0 to $2^8 - 1$, inclusive.

7.4.5 Slice data semantics

7.4.5.1 General slice data semantics

end_of_slice_flag equal to 0 specifies that another CTU is following in the slice. end_of_slice_flag equal to 1 specifies the end of the slice, i.e., that no further CTU follows in the slice.

7.4.5.2 Coding tree unit semantics

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

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

When alf_ctb_flag[cIdx][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize] is not present, it is inferred as follows

– If all of the following conditions are true, alf_ctb_flag[cIdx][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize] is inferred to be equal to 1:
  – alf_chroma_ctb_present_flag is equal to 0,
  – cIdx is greater than 0,
  – alf_chroma_idc is equal to cIdx or alf_chroma_idc is equal to 3
– Otherwise, alf_ctb_flag[cIdx][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize] is inferred to be equal to 0.

7.4.5.3 Sample adaptive offset semantics

sao_merge_left_flag equal to 1 specifies that the syntax elements sao_type_idx_luma, sao_type_idx_chroma, sao_band_position, sao_eo_class_luma, sao_eo_class_chroma, sao_offset_abs and sao_offset_sign are derived from the corresponding syntax elements of the left CTB. sao_merge_left_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the left CTB. When sao_merge_left_flag is not present, it is inferred to be equal to 0.

sao_merge_up_flag equal to 1 specifies that the syntax elements sao_type_idx_luma, sao_type_idx_chroma, sao_band_position, sao_eo_class_luma, sao_eo_class_chroma, sao_offset_abs and sao_offset_sign are derived from the corresponding syntax elements of the above CTB. sao_merge_up_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the above CTB. When sao_merge_up_flag is not present, it is inferred to be equal to 0.

sao_type_idx_luma specifies the offset type for the luma component. The array SaoTypeIdx[cIdx][rx][ry] specifies the offset type as specified in Table 7-3 for the CTB at the location (rx, ry) for the colour component cIdx. The value of SaoTypeIdx[0][rx][ry] is derived as follows:

– If sao_type_idx_luma is present, SaoTypeIdx[0][rx][ry] is set equal to sao_type_idx_luma.
– Otherwise (sao_type_idx_luma is not present), SaoTypeIdx[0][rx][ry] is derived as follows:
  – If sao_merge_left_flag is equal to 1, SaoTypeIdx[0][rx][ry] is set equal to SaoTypeIdx[0][rx − 1][ry].
  – Otherwise, if sao_merge_up_flag is equal to 1, SaoTypeIdx[0][rx][ry] is set equal to SaoTypeIdx[0][rx][ry − 1].
  – Otherwise, SaoTypeIdx[0][rx][ry] is set equal to 0.

sao_type_idx_chroma specifies the offset type for the chroma components. The values of SaoTypeIdx[cIdx][rx][ry] are derived as follows for cIdx equal to 1..2:

– If sao_type_idx_chroma is present, SaoTypeIdx[cIdx][rx][ry] is set equal to sao_type_idx_chroma.

– Otherwise (sao_type_idx_chroma is not present), SaoTypeIdx[cIdx][rx][ry] is derived as follows:

  – If sao_merge_left_flag is equal to 1, SaoTypeIdx[cIdx][rx][ry] is set equal to SaoTypeIdx[cIdx][rx − 1][ry].

  – Otherwise, if sao_merge_up_flag is equal to 1, SaoTypeIdx[cIdx][rx][ry] is set equal to SaoTypeIdx[cIdx][rx][ry − 1].

  – Otherwise, SaoTypeIdx[cIdx][rx][ry] is set equal to 0.

Table 7-3 – Specification of the SAO type

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

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

When sao_offset_abs[cIdx][rx][ry][i] is not present, it is inferred as follows:

– If sao_merge_left_flag is equal to 1, sao_offset_abs[cIdx][rx][ry][i] is inferred to be equal to sao_offset_abs[cIdx][rx − 1][ry][i].

– Otherwise, if sao_merge_up_flag is equal to 1, sao_offset_abs[cIdx][rx][ry][i] is inferred to be equal to sao_offset_abs[cIdx][rx][ry − 1][i].

– Otherwise, sao_offset_abs[cIdx][rx][ry][i] is inferred to be equal to 0.

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

When sao_offset_sign[cIdx][rx][ry][i] is not present, it is inferred as follows:

– If sao_merge_left_flag is equal to 1, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal to sao_offset_sign[cIdx][rx − 1][ry][i].

– Otherwise, if sao_merge_up_flag is equal to 1, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal to sao_offset_sign[cIdx][rx][ry − 1][i].

– Otherwise, if SaoTypeIdx[cIdx][rx][ry] is equal to 2, the following applies:

  – If i is equal to 0 or 1, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal 0.

  – Otherwise (i is equal to 2 or 3), sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal 1.

– Otherwise, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal 0.

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

```
SaoOffsetVal[ cIdx ][ rx ][ ry ][ 0 ] = 0
for( i = 0; i < 4; i++ )
SaoOffsetVal[ cIdx ][ rx ][ ry ][ i + 1 ] = ( 1 − 2 * sao_offset_sign[ cIdx ][ rx ][ ry ][ i ] ) *
        sao_offset_abs[ cIdx ][ rx ][ ry ][ i ]
```
(7-52)

sao_band_position[cIdx][rx][ry] specifies the displacement of the band offset of the sample range when SaoTypeIdx[cIdx][rx][ry] is equal to 1.

When sao_band_position[cIdx][rx][ry] is not present, it is inferred as follows:

– If sao_merge_left_flag is equal to 1, sao_band_position[cIdx][rx][ry] is inferred to be equal to sao_band_position[cIdx][rx − 1][ry].

– Otherwise, if sao_merge_up_flag is equal to 1, sao_band_position[cIdx][rx][ry] is inferred to be equal to sao_band_position[cIdx][rx][ry − 1].

– Otherwise, sao_band_position[cIdx][rx][ry] is inferred to be equal to 0.

sao_eo_class_luma specifies the edge offset class for the luma component. The array SaoEoClass[cIdx][rx][ry] specifies the offset type as specified in Table 7-4 for the CTB at the location (rx, ry) for the colour component cIdx. The value of SaoEoClass[0][rx][ry] is derived as follows:

– If sao_eo_class_luma is present, SaoEoClass[0][rx][ry] is set equal to sao_eo_class_luma.

– Otherwise (sao_eo_class_luma is not present), SaoEoClass[0][rx][ry] is derived as follows:

  – If sao_merge_left_flag is equal to 1, SaoEoClass[0][rx][ry] is set equal to SaoEoClass[0][rx − 1][ry].

  – Otherwise, if sao_merge_up_flag is equal to 1, SaoEoClass[0][rx][ry] is set equal to SaoEoClass[0][rx][ry − 1].

  – Otherwise, SaoEoClass[0][rx][ry] is set equal to 0.

sao_eo_class_chroma specifies the edge offset class for the chroma components. The values of SaoEoClass[cIdx][rx][ry] are derived as follows for cIdx equal to 1..2:

– If sao_eo_class_chroma is present, SaoEoClass[cIdx][rx][ry] is set equal to sao_eo_class_chroma.

– Otherwise (sao_eo_class_chroma is not present), SaoEoClass[cIdx][rx][ry] is derived as follows:

  – If sao_merge_left_flag is equal to 1, SaoEoClass[cIdx][rx][ry] is set equal to SaoEoClass[cIdx][rx − 1][ry].

  – Otherwise, if sao_merge_up_flag is equal to 1, SaoEoClass[cIdx][rx][ry] is set equal to SaoEoClass[cIdx][rx][ry − 1].

  – Otherwise, SaoEoClass[cIdx][rx][ry] is set equal to 0.

Table 7-4 – Specification of the SAO edge offset class

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

7.4.5.4 Coding quadtree semantics

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

When qt_split_cu_flag[x0][y0] is not present, the following applies:

– If one or more of the following conditions are true, the value of qt_split_cu_flag[x0][y0] is inferred to be equal to 1:
  – x0 + (1 << log2CbSize) is greater than pic_width_in_luma_samples and (1 << log2CbSize) is greater than MaxBtSizeY.
  – y0 + (1 << log2CbSize) is greater than pic_height_in_luma_samples and (1 << log2CbSize) is greater than MaxBtSizeY.
– Otherwise, if all of the following conditions are true, the value of qt_split_cu_flag[x0][y0] is inferred to be equal to 1:
  – x0 + (1 << log2CbSize) is greater than pic_width_in_luma_samples.
  – y0 + (1 << log2CbSize) is greater than pic_height_in_luma_samples.
  – (1 << log2CbSize) is greater than MinQtSizeY.
– Otherwise, the value of qt_split_cu_flag[x0][y0] is inferred to be equal to 0.

7.4.5.5 Multi-type tree semantics

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

– The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT_BT_VER, the coding block width cbWidth, the coding block height cbHeight, the location (x0, y0), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, and the current partition index partIdx as inputs, and the output is assigned to allowSplitBtVer.
– The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT_BT_HOR, the coding block height cbHeight, the coding block width cbWidth, the location (x0, y0), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, and the current partition index partIdx as inputs, and the output is assigned to allowSplitBtHor.
– The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT_TT_VER, the coding block width cbWidth, the coding block height cbHeight, the location (x0, y0), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, and the current partition index partIdx as inputs, and the output is assigned to allowSplitTtVer.
– The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT_TT_HOR, the coding block height cbHeight, the coding block width cbWidth, the location (x0, y0), the current multi-type tree depth mttDepth, the current MaxMttDepth offset depthOffset, and the current partition index partIdx as inputs, and the output is assigned to allowSplitTtHor.

mtt_split_cu_flag equal to 0 specifies that a coding unit is not split. mtt_split_cu_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split or into three coding units using a ternary split as indicated by the syntax element mtt_split_cu_binary_flag. The binary or ternary split can be either vertical or horizontal as indicated by the syntax element mtt_split_cu_vertical_flag.

When mtt_split_cu_flag is not present, the value of mtt_split_cu_flag is inferred as follows:

– If one or more of the following conditions are true, the value of mtt_split_cu_flag is inferred to be equal to 1:
  – x0 + cbWidth is greater than pic_width_in_luma_samples.
  – y0 + cbHeight is greater than pic_height_in_luma_samples.
– Otherwise, the value of mtt_split_cu_flag is inferred to be equal to 0.

mtt_split_cu_vertical_flag equal to 0 specifies that a coding unit is split horizontally. mtt_split_cu_vertical_flag equal to 1 specifies that a coding unit is split vertically

When mtt_split_cu_vertical_flag is not present, it is inferred as follows:

– If allowSplitBtHor is equal to TRUE or allowSplitTtHor is equal to TRUE, the value of mtt_split_cu_vertical_flag is inferred to be equal to 0.

– Otherwise, the value of mtt_split_cu_vertical_flag is inferred to be equal to 1.

mtt_split_cu_binary_flag equal to 0 specifies that a coding unit is split into three coding units using a ternary split. mtt_split_cu_binary_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split.

When mtt_split_cu_binary_flag is not present, it is inferred as follows:

– If allowSplitBtVer is equal to FALSE and allowSplitBtHor is equal to FALSE, the value of mtt_split_cu_binary_flag is inferred to be equal to 0.

– Otherwise, if allowSplitTtVer is equal to FALSE and allowSplitTtHor is equal to FALSE, the value of mtt_split_cu_binary_flag is inferred as to be equal to 1.

– Otherwise, if allowSplitBtHor is equal to TRUE and allowSplitTtVer is equal to TRUE, the value of mtt_split_cu_binary_flag is inferred to be equal to !mtt_split_cu_vertical_flag.

– Otherwise (allowSplitBtVer is equal to TRUE and allowSplitTtHor is equal to TRUE), the value of mtt_split_cu_binary_flag is inferred to be equal to mtt_split_cu_vertical_flag.

The variable MttSplitMode[x][y][mttDepth] is derived from the value of mtt_split_cu_vertical_flag and from the value of mtt_split_cu_binary_flag as defined in Table 7-5 for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

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

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

Table 7-5 – Specification of MttSplitMode[x][y][mttDepth] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1

MttSplitMode[x0][y0][mttDepth]	mtt_split_cu_vertical_flag	mtt_split_cu_binary_flag
SPLIT_TT_HOR	0	0
SPLIT_BT_HOR	0	1
SPLIT_TT_VER	1	0
SPLIT_BT_VER	1	1

7.4.5.6 Coding unit semantics

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

CbPosX[x][y] = x0 (7-53)

CbPosY[x][y] = y0 (7-54)

CbWidth[x][y] = cbWidth (7-55)

CbHeight[x][y] = cbHeight (7-56)

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

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

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

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

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

When pred_mode_flag is not present, the variable CuPredMode[x][y] is inferred to be equal to MODE_INTRA for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

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

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

mvp_l0_flag[x0][y0] specifies the motion vector predictor index of list 0 where x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When mvp_l0_flag[x0][y0] is not present, it is inferred to be equal to 0.

mvp_l1_flag[x0][y0] has the same semantics as mvp_l0_flag, with l0 and list 0 replaced by l1 and list 1, respectively.

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

When merge_flag[x0][y0] is not present, it is inferred as follows:

– If cu_skip_flag[x0][y0] is equal to 1, merge_flag[x0][y0] is inferred to be equal to 1.

– Otherwise, merge_flag[x0][y0] is inferred to be equal to 0.

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

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

inter_pred_idc[x0][y0] specifies whether list0, list1, or bi-prediction is used for the current coding unit according to Table 7-6. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7-6 – Name association to inter prediction mode

inter_pred_idc	Name of inter_pred_idc	
	(cbWidth + cbHeight) != 12	(cbWidth + cbHeight) = = 12
0	PRED_L0	PRED_L0
1	PRED_L1	PRED_L1
2	PRED_BI	n.a.

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

ref_idx_l0[x0][y0] specifies the list 0 reference picture index for the current coding unit. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When ref_idx_l0[x0][y0] is not present it is inferred to be equal to 0.

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

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

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

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

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

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

– If merge_flag[x0][y0] is equal to 1, the following applies:

$$MotionModelIdc[x][y] = merge\_affine\_flag[x0][y0] \quad (7\text{-}57)$$

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

$$MotionModelIdc[x][y] = inter\_affine\_flag[x0][y0] + cu\_affine\_type\_flag[x0][y0] \quad (7\text{-}58)$$

Table 7-7 – Interpretation of MotionModelIdc[x0][y0]

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

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

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

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

$$MvdL0[x0][y0][0] = MvdL0[x0][y0][0] << (MvShift + 2) \quad (7\text{-}59)$$

$$MvdL0[x0][y0][1] = MvdL0[x0][y0][1] << (MvShift + 2) \quad (7\text{-}60)$$

$$MvdL1[x0][y0][0] = MvdL1[x0][y0][0] << (MvShift + 2) \quad (7\text{-}61)$$

$$MvdL1[x0][y0][1] = MvdL1[x0][y0][1] << (MvShift + 2) \quad (7\text{-}62)$$

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

When cu_cbf is not present, it is inferred as follows:

– If cu_skip_flag[x0][y0] is equal to 1, cu_cbf is inferred to be equal to 0.

– Otherwise, cu_cbf is inferred to be equal to 1.

7.4.5.7 Motion vector difference semantics

abs_mvd_greater0_flag[compIdx] specifies whether the absolute value of a motion vector component difference is greater than 0.

abs_mvd_greater1_flag[compIdx] specifies whether the absolute value of a motion vector component difference is greater than 1.

When abs_mvd_greater1_flag[compIdx] is not present, it is inferred to be equal to 0.

abs_mvd_minus2[compIdx] plus 2 specifies the absolute value of a motion vector component difference.

When abs_mvd_minus2[compIdx] is not present, it is inferred to be equal to −1.

mvd_sign_flag[compIdx] specifies the sign of a motion vector component difference as follows:

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

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

When mvd_sign_flag[compIdx] is not present, it is inferred to be equal to 0.

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

$$lMvd[compIdx] = abs\_mvd\_greater0\_flag[compIdx] * (abs\_mvd\_minus2[compIdx] + 2) * (1 - 2 * mvd\_sign\_flag[compIdx]) \quad (7\text{-}63)$$

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

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

– If MotionModelIdc[x][y] is equal to 0, the variable MvdLX[x0][y0][compIdx], with X being 0 or 1, specifies the difference between a list X vector component to be used and its prediction. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

  – If refList is equal to 0, MvdL0[x0][y0][compIdx] is set equal to lMvd[compIdx] for compIdx = 0..1.

  – Otherwise (refList is equal to 1), MvdL1[x0][y0][compIdx] is set equal to lMvd[compIdx] for compIdx = 0..1.

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

  – If refList is equal to 0, MvdCpL0[x0][y0][cpIdx][compIdx] is set equal to (lMvd[compIdx] << 2) for compIdx = 0..1.

  – Otherwise (refList is equal to 1), MvdCpL1[x0][y0][cpIdx][compIdx] is set equal to (lMvd[compIdx] << 2) for compIdx = 0..1.

7.4.5.8 Transform tree semantics

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

7.4.5.9 Transform unit semantics

The transform coefficient levels are represented by the arrays TransCoeffLevel[x0][y0][cIdx][xC][yC]. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for Y, 1 for Cb, and 2 for Cr. The array indices xC and yC specify the transform coefficient location (xC, yC) within the current transform block. When the value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is not specified in clause 7.3.4.10, it is inferred to be equal to 0.

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

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

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

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

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

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

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

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

7.4.5.10 Residual coding semantics

The array AbsLevel[xC][yC] represents an array of absolute values of transform coefficient levels for the current transform block and the array AbsLevelPass1[xC][yC] represents an array of partially reconstructed absolute values of transform coefficient levels for the current transform block. The array indices xC and yC specify the transform coefficient location (xC, yC) within the current transform block. When the value of AbsLevel[xC][yC] is not specified in clause 7.3.4.10, it is inferred to be equal to 0. When the value of AbsLevelPass1[xC][yC] is not specified in clause 7.3.4.10, it is inferred to be equal to 0.

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

$$CoeffMin = -(1 \ll 15) \tag{7-64}$$

$$CoeffMax = (1 \ll 15) - 1 \tag{7-65}$$

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

$$QStateTransTable[ ][ ] = \{ \{ 0, 2 \}, \{ 2, 0 \}, \{ 1, 3 \}, \{ 3, 1 \} \} \tag{7-66}$$

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

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

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

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

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

– If last_sig_coeff_x_suffix is not present, the following applies:

$$LastSignificantCoeffX = last\_sig\_coeff\_x\_prefix \tag{7-67}$$

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

$$LastSignificantCoeffX = (1 \ll ((last\_sig\_coeff\_x\_prefix \gg 1) - 1)) * (2 + (last\_sig\_coeff\_x\_prefix \,\&\, 1)) + last\_sig\_coeff\_x\_suffix \tag{7-68}$$

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

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

– If last_sig_coeff_y_suffix is not present, the following applies:

$$LastSignificantCoeffY = last\_sig\_coeff\_y\_prefix \tag{7-69}$$

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

$$LastSignificantCoeffY = (1 << ((last\_sig\_coeff\_y\_prefix >> 1) - 1)) * (2 + (last\_sig\_coeff\_y\_prefix \& 1)) + last\_sig\_coeff\_y\_suffix \tag{7-70}$$

coded_sub_block_flag[xS][yS] specifies the following for the subblock at location (xS, yS) within the current transform block, where a subblock is a (4x4) array of 16 transform coefficient levels:

– If coded_sub_block_flag[xS][yS] is equal to 0, the 16 transform coefficient levels of the subblock at location (xS, yS) are inferred to be equal to 0.

– Otherwise (coded_sub_block_flag[xS][yS] is equal to 1), the following applies:

  – If (xS, yS) is equal to (0, 0) and (LastSignificantCoeffX, LastSignificantCoeffY) is not equal to (0, 0), at least one of the 16 sig_coeff_flag syntax elements is present for the subblock at location (xS, yS).

  – Otherwise, at least one of the 16 transform coefficient levels of the subblock at location (xS, yS) has a non-zero value.

When coded_sub_block_flag[xS][yS] is not present, it is inferred as follows:

– If one or more of the following conditions are true, coded_sub_block_flag[xS][yS] is inferred to be equal to 1:

  – (xS, yS) is equal to (0, 0).

  – (xS, yS) is equal to (LastSignificantCoeffX >> 2, LastSignificantCoeffY >> 2).

– Otherwise, coded_sub_block_flag[xS][yS] is inferred to be equal to 0.

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

– If sig_coeff_flag[xC][yC] is equal to 0, the transform coefficient level at the location (xC, yC) is set equal to 0.

– Otherwise (sig_coeff_flag[xC][yC] is equal to 1), the transform coefficient level at the location (xC, yC) has a non-zero value.

When sig_coeff_flag[xC][yC] is not present, it is inferred as follows:

– If (xC, yC) is the last significant location (LastSignificantCoeffX, LastSignificantCoeffY) in scan order or all of the following conditions are truet[xC][yC] is inferred to be equal to 1:

  – (xC & 3, yC & 3) is equal to (0, 0).

  – inferSbDcSigCoeffFlag is equal to 1.

  – coded_sub_block_flag[xS][yS] is equal to 1.

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

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

rem_abs_gt1_flag[n] specifies whether the syntax element rem_abs_gt2_flag[n] is present for the scanning position n. When rem_abs_gt1_flag[n] is not present, it is inferred to be equal to 0.

rem_abs_gt2_flag[n] specifies whether the syntax element abs_remainder[n] is present for the scanning position n. When rem_abs_gt2_flag[n] is not present, it is inferred to be equal to 0.

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

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

coeff_sign_flag[n] specifies the sign of a transform coefficient level for the scanning position n as follows:

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

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

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

mts_idx[x0][y0][cIdx specifies which transform kernels are applied to theresidual samples along the horizontal and vertical direction of the current transform block. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture the array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb and equal to 2 for Cr.

When mts_idx[x0][y0][cIdx is not present, it is inferred as follows:

– If tu_mts_flag[x0][y0] is equal to 0 or cIdx is greater than 0, mts_idx[x0][y0][cIdx is inferred to be equal to −1.

– Otherwise, mts_idx[x0][y0][cIdx is inferred to be equal to 0.

8 Decoding process

8.1 General decoding process

8.2 Decoding process for coding units coded in intra prediction mode

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

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– a variable treeType specifying whether a single or a dual tree is used and if a dual tree is used, it specifies whether the current tree corresponds to the luma or chroma components.

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

If treeType is equal to SINGLE_TREE or treeType is equal to DUAL_TREE_LUMA, the decoding process for luma samples is specified as follows:

1. The derivation process for the luma intra prediction mode as specified in clause 8.2.2 is invoked with the luma location (xCb, yCb), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.

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

If treeType is equal to SINGLE_TREE or treeType is equal to DUAL_TREE_CHROMA, the decoding process for chroma samples is specified as follows:

1. The derivation process for the chroma intra prediction mode as specified in clause 8.2.3 is invoked with the luma location (xCb, yCb) as input.

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

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

8.2.2 Derivation process for luma intra prediction mode

Input to this process are:

– a luma location (xCb , yCb) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

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

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

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

Intra prediction mode	Associated name
0	INTRA_PLANAR
1	INTRA_DC
2..66	INTRA_ANGULAR2..INTRA_ANGULAR66
77	INTRA_CCLM

NOTE – : The intra prediction mode INTRA_CCLM is only applicable to chroma components.

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

1. The neighbouring locations (xNbA, yNbA) and (xNbB, yNbB) are set equal to (xCb − 1, yCb) and (xCb, yCb − 1), respectively.
2. For X being replaced by either A or B, the variables candIntraPredModeX are derived as follows:
   – The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring location (xNbY, yNbY) set equal to (xNbX, yNbX) as inputs, and the output is assigned to availableX.
   – The candidate intra prediction mode candIntraPredModeX is derived as follows:
     – If one or more of the following conditions are true, candIntraPredModeX is set equal to INTRA_DC.
       – The variable availableX is equal to FALSE.
       – CuPredMode[xNbX][yNbX] is not equal to MODE_INTRA.
       – X is equal to B and yCb − 1 is less than ((yCb >> CtbLog2SizeY) << CtbLog2SizeY).
     – Otherwise, candIntraPredModeX is set equal to IntraPredModeY[xNbX][yNbX].
3. The candModeList[x] with x = 0..2 is derived as follows:
   – If candIntraPredModeB is equal to candIntraPredModeA, the following applies:
     – If candIntraPredModeA is less than 2 (i.e., equal to INTRA_PLANAR or INTRA_DC), candModeList[x] with x = 0..2 is derived as follows:

       candModeList[0] = INTRA_PLANAR (8-1)

       candModeList[1] = INTRA_DC (8-2)

       candModeList[2] = INTRA_ANGULAR50 (8-3)

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

       candModeList[0] = candIntraPredModeA (8-4)

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

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

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

    candModeList[0] = candIntraPredModeA (8-7)

    candModeList[1] = candIntraPredModeB (8-8)

  – If neither of candModeList[0] and candModeList[1] is equal to INTRA_PLANAR, candModeList[2] is set equal to INTRA_PLANAR,
  – Otherwise, if neither of candModeList[0] and candModeList[1] is equal to INTRA_DC, candModeList[2] is set equal to INTRA_DC,
  – Otherwise, candModeList[2] is set equal to INTRA_ANGULAR50.

4. IntraPredModeY[xCb][yCb] is derived by applying the following procedure:
  – If intra_luma_mpm_flag[xCb][yCb] is equal to 1, the IntraPredModeY[xCb][yCb] is set equal to candModeList[intra_luma_mpm_idx[xCb][yCb]].
  – Otherwise, IntraPredModeY[xCb][yCb] is derived by applying the following ordered steps:
    1. The array candModeList[x], x = 0..2 is modified by the following ordered steps:
      i. When candModeList[0] is greater than candModeList[1], both values are swapped as follows:

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

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

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

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

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

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

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

8.2.3 Derivation process for chroma intra prediction mode

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

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

The chroma intra prediction mode IntraPredModeC[xCb][yCb] is derived using intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb][yCb] as specified in Table 8-2 and Table 8-3.

Table 8-2 – Specification of IntraPredModeC[xCb][yCb] depending on intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb][yCb] when sps_cclm_enabled_flag is equal to 0

intra_chroma_pred_mode[xCb][yCb]	IntraPredModeY[xCb][yCb]				
	0	50	18	1	X (0 <= X <= 66)
0	66	0	0	0	0
1	50	66	50	50	50
2	18	18	66	18	18
3	1	1	1	66	1
4	0	50	18	1	X

Table 8-3 – Specification of IntraPredModeC[xCb][yCb] depending on intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb][yCb] when sps_cclm_enabled_flag is equal to 1

intra_chroma_pred_mode[xCb][yCb]	IntraPredModeY[xCb][yCb]				
	0	50	18	1	X (0 <= X <= 66)
0	66	0	0	0	0
1	50	66	50	50	50
2	18	18	66	18	18
3	1	1	1	66	1
4	77	77	77	77	77
5	0	50	18	1	X

8.2.4 Decoding process for intra blocks

8.2.4.1 General decoding process for intra blocks

Inputs to this process are:

– a sample location (xTb0, yTb0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable nTbW specifying the width of the current transform block,

– a variable nTbH specifying the height of the current transform block,

– a variable predModeIntra specifying the intra prediction mode,

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

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

The maximum transform block size maxTbSize is derived as follows:

$$maxTbSize = (cIdx == 0) ? MaxTbSizeY : MaxTbSizeY / 2 \quad (8\text{-}12)$$

The luma sample location is derived as follows:

$$(xTbY, yTbY) = (cIdx == 0) ? (xTb0, yTb0) : (xTb0 * 2, yTb0 * 2) \quad (8\text{-}13)$$

Depending on maxTbSize, the following applies:

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

1. The variables newTbW and newTbH are derived as follows:

$$newTbW = (nTbW > maxTbSize) ? (nTbW / 2) : nTbW \quad (8\text{-}14)$$

$$newTbH = (nTbH > maxTbSize) ? (nTbH / 2) : nTbH \quad (8\text{-}15)$$

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

– Otherwise, the following ordered steps apply:

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

8.2.4.2 Intra sample prediction

8.2.4.2.1 General intra sample prediction

Inputs to this process are:

– a sample location (xTbCmp, yTbCmp) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable predModeIntra specifying the intra prediction mode,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

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

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

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

The variables refW and refH are derived as follows:

$$refW = (nTbH > nTbW) ? (nTbW + (nTbH >> whRatio) + Ceil(nTbH / 32)) : (nTbW * 2) \quad (8\text{-}16)$$

$$refH = (nTbW > nTbH) ? (nTbH + (nTbW >> whRatio) + Ceil(nTbW / 32)) : (nTbH * 2) \quad (8\text{-}17)$$

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

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

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

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

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

– predModeIntra is equal to INTRA_PLANAR
– predModeIntra is equal to INTRA_DC
– predModeIntra is equal to INTRA_ANGULAR18
– predModeIntra is equal to INTRA_ANGULAR50
– predModeIntra is less than or equal to INTRA_ANGULAR10
– predModeIntra is greater than or equal to INTRA_ANGULAR58

8.2.4.2.2 Reference sample availability marking process

Inputs to this process are:

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

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

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

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

$$(xNbCmp, yNbCmp) = (xTbCmp + x, yTbCmp + y) \quad (8\text{-}18)$$

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

$$(xTbY, yTbY) = (cIdx == 0) ? (xTbCmp, yTbCmp) : (xTbCmp << 1, yTbCmp << 1) \quad (8\text{-}19)$$

$$(xNbY, yNbY) = (cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp << 1, yNbCmp << 1) \quad (8\text{-}20)$$

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

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

  – If availableN is equal to FALSE, the sample refUnfilt[x][y] is marked as "not available for intra prediction".

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

8.2.4.2.3 Reference sample substitution process

Inputs to this process are:

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

– reference samples refUnfilt[x][y] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 for intra sample prediction,

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

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

The variable bitDepth is derived as follows:

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

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

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

– If all samples refUnfilt[x][y] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are marked as "not available for intra prediction", all values of refUnfilt[x][y] are set equal to 1 << (bitDepth − 1).

– Otherwise (at least one but not all samples refUnfilt[x][y] are marked as "not available for intra prediction"), the following ordered steps apply:

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

  2. For x = −1, y = refH − 2..−1, when refUnfilt[x][y] is marked as "not available for intra prediction", the value of refUnfilt[x][y] is set equal to the value of refUnfilt[x][y + 1].

  3. For x = 0..refW − 1, y = −1, when refUnfilt[x][y] is marked as "not available for intra prediction", the value of refUnfilt[x][y] is set equal to the value of refUnfilt[x − 1][y].

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

8.2.4.2.4 Reference sample filtering process

Inputs to this process are:

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

– the (unfiltered) neighbouring samples refUnfilt[x][y], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1,

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

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

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

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

The variable wideAngle is derived as follows:

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

The variable filterFlag is derived as follows:

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

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

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

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

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

$$p[-1][-1] = (refUnfilt[-1][0] + 2 * refUnfilt[-1][-1] + refUnfilt[0][-1] + 2) >> 2 \quad (8\text{-}21)$$

$$p[-1][y] = (refUnfilt[-1][y + 1] + 2 * refUnfilt[-1][y] + refUnfilt[-1][y - 1] + 2) >> 2 \quad \text{for } y = 0..refH - 2 \quad (8\text{-}22)$$

$$p[-1][refH - 1] = refUnfilt[-1][refH - 1] \quad (8\text{-}23)$$

$$p[x][-1] = (refUnfilt[x - 1][-1] + 2 * refUnfilt[x][-1] + refUnfilt[x + 1][-1] + 2) >> 2 \quad \text{for } x = 0..refW - 2 \quad (8\text{-}24)$$

$$p[refW - 1][-1] = refUnfilt[refW - 1][-1] \quad (8\text{-}25)$$

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

8.2.4.2.5 Specification of INTRA_PLANAR intra prediction mode

Inputs to this process are:

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

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

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

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

$$predV[x][y] = ((nTbH - 1 - y) * p[x][-1] + (y + 1) * p[-1][nTbH]) << Log2 (nTbW) \quad (8\text{-}26)$$

$$predH[x][y] = ((nTbW - 1 - x) * p[-1][y] + (x + 1) * p[nTbW][-1]) << Log2 (nTbH) \quad (8\text{-}27)$$

$$predSamples[x][y] = (predV[x][y] + predH[x][y] + nTbW * nTbH) >> (Log2 (nTbW) + Log2 (nTbH) + 1) \quad (8\text{-}28)$$

8.2.4.2.6 Specification of INTRA_DC intra prediction mode

Inputs to this process are:

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

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

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

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

1. A variable dcVal is derived as follows:
   – When nTbW is equal to nTbH:

$$dcVal = \left(\sum_{x'=0}^{nTbW-1} p[x'][-1] + \sum_{y'=0}^{nTbH-1} p[-1][y'] + nTbW\right) \gg (Log2(nTbW) + 1) \quad (8\text{-}29)$$

   – When nTbW is greater than nTbH:

$$dcVal = \left(\sum_{x'=0}^{nTbW-1} p[x'][-1] + (nTbW \gg 1)\right) \gg Log2(nTbW) \quad (8\text{-}30)$$

   – When nTbW is less than nTbH:

$$dcVal = \left(\sum_{y'=0}^{nTbH-1} p[-1][y'] + (nTbH \gg 1)\right) \gg Log2(nTbH) \quad (8\text{-}31)$$

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

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

8.2.4.2.7 Specification of INTRA_ANGULAR2..INTRA_ANGULAR66 intra prediction modes

Inputs to this process are:

– the intra prediction mode predModeIntra,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

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

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

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

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

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

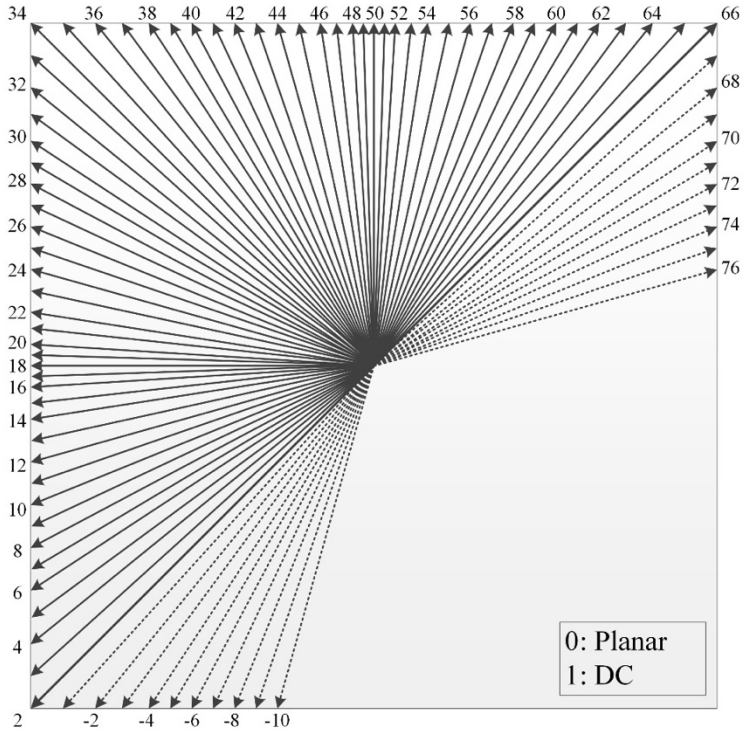


Figure 8-1 – Intra prediction directions (informative)

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

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

Table 8-5 – Specification of intraPredAngle

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

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

$$\text{invAngle} = \text{Round}\left(\frac{256*32}{\text{intraPredAngle}}\right) \tag{8-33}$$

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

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

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

$$\text{ref}[\ x\ ] = p[\ -1 + x\ ][\ -1\ ], \text{ with } x = 0..nTbW \tag{8-34}$$

– If intraPredAngle is less than 0, the main reference sample array is extended as follows:

– When (nTbH * intraPredAngle) >> 5 is less than −1,

$$ref[x] = p[-1][-1 + ((x * invAngle + 128) >> 8)], \text{ with } x = -1..(nTbH * intraPredAngle) >> 5 \quad (8\text{-}35)$$

– Otherwise,

$$ref[x] = p[-1 + x][-1], \text{ with } x = nTbW + 1..refW \quad (8\text{-}36)$$

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

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

$$iIdx = ((y + 1) * intraPredAngle) >> 5 \quad (8\text{-}37)$$

$$iFact = ((y + 1) * intraPredAngle) \& 31 \quad (8\text{-}38)$$

– Depending on the value of iFact, the following applies:

– If iFact is not equal to 0, the value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = ((32 - iFact) * ref[x + iIdx + 1] + iFact * ref[x + iIdx + 2] + 16) >> 5 \quad (8\text{-}39)$$

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

$$predSamples[x][y] = ref[x + iIdx + 1] \quad (8\text{-}40)$$

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

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

– The following applies:

$$ref[x] = p[-1][-1 + x], \text{ with } x = 0..nTbH \quad (8\text{-}41)$$

– If intraPredAngle is less than 0, the main reference sample array is extended as follows:

– When (nTbW * intraPredAngle) >> 5 is less than −1,

$$ref[x] = p[-1 + ((x * invAngle + 128) >> 8)][-1], \text{ with } x = -1..(nTbW * intraPredAngle) >> 5 \quad (8\text{-}42)$$

– Otherwise,

$$ref[x] = p[-1][-1 + x], \text{ with } x = nTbH + 1..refH \quad (8\text{-}43)$$

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

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

$$iIdx = ((x + 1) * intraPredAngle) >> 5 \quad (8\text{-}44)$$

$$iFact = ((x + 1) * intraPredAngle) \& 31 \quad (8\text{-}45)$$

– Depending on the value of iFact, the following applies:

– If iFact is not equal to 0, the value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = ((32 - iFact) * ref[y + iIdx + 1] + iFact * ref[y + iIdx + 2] + 16) >> 5 \quad (8\text{-}46)$$

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

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

8.2.4.2.8 Specification of INTRA_CCLM intra prediction mode

Inputs to this process are:

– a sample location (xTbC, yTbC) of the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– chroma neighbouring samples p[x][y], with x = −1, y = 0..nTbH − 1 and x = 0..nTbW − 1, y = − 1.

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

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

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

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

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

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

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

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

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

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

– Otherwise, the following ordered steps apply:

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

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

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

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

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

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

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

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

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

– pTopDsY[x] with x = 1..nTbW − 1 is derived as follows:

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

– If availTL is equal to TRUE, pTopDsY[0] is derived as follows:

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

– Otherwise, pTopDsY[0] is derived as follows:

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

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

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

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

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

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

k0 = BitDepthC + k1 − 15 (8-61)

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

l = (availL ? $\sum_{y=0}^{nS-1}$ pLeftDsY[$y * yS$] : 0) + (availT ? $\sum_{x=0}^{nS-1}$ pTopDsY[$x * xS$] : 0) (8-62)

c = (availL ? $\sum_{y=0}^{nS-1}$ p [−1][$y * yS$] : 0) + (availT ? $\sum_{x=0}^{nS-1}$ p [$x * xS$][−1] : 0) (8-63)

ll = (availL ? $\sum_{y=0}^{nS-1}$(pLeftDsY[$y * yS$])^2 : 0) + (availT ? $\sum_{x=0}^{nS-1}$(pTopDsY[$x * xS$])^2 : 0) (8-64)

lc = (availL ? $\sum_{y=0}^{nS-1}$ pLeftDsY[$y * yS$] * p[−1][$y * yS$] : 0) + (8-65)
(availT ? $\sum_{x=0}^{nS-1}$ pTopDsY[$x * xS$] * p[$x * xS$][−1] : 0)

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

$$\begin{aligned} l &= (l + (l << (k0 - 1))) >> k0 \\ c &= (c + (c << (k0 - 1))) >> k0 \\ ll &= (ll + (ll << (k0 - 1))) >> k0 \\ lc &= (lc + (lc << (k0 - 1))) >> k0 \\ k1 &= k1 - k0 \end{aligned} \tag{8-66}$$

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

$$k = 0 \tag{8-67}$$

$$a = 0 \tag{8-68}$$

$$b = 1 << (BitDepthC - 1) \tag{8-69}$$

   - Otherwise, the following applies:

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

$$k = 13 - k5 \tag{8-71}$$

$$a = a5 >> k5 \tag{8-72}$$

$$b = avgC - ((a * avgY) >> k) \tag{8-73}$$

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

$$predSamples[x][y] = Clip1C(((pDsY[x][y] * a) >> k) + b) \tag{8-74}$$

8.2.4.2.9 Position-dependent intra prediction sample filtering process

Inputs to this process are:

- the intra prediction mode predModeIntra,
- a variable nTbW specifying the transform block width,
- a variable nTbH specifying the transform block height,
- a variable refW specifying the reference samples width,
- a variable refH specifying the reference samples height,

– the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1,

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

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

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

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

– If cIdx is equal to 0, clip1Cmp is set equal to $Clip1_Y$.

– Otherwise, clip1Cmp is set equal to $Clip1_C$.

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

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

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

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

– If predModeIntra is equal to INTRA_PLANAR, INTRA_DC, INTRA_ANGULAR18, or INTRA_ANGULAR50, the following applies:

```
refL[ x ][ y ] = p[ −1 ][ y ]                                                  (8-76)

refT[ x ][ y ] = p[ x ][ −1 ]                                                  (8-77)

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

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

wTL[ x ][ y ] = ( predModeIntra == INTRA_DC ) ? ( ( wL[ x ] >> 4 ) + ( wT[ y ] >> 4 ) ) : 0   (8-80)
```

– Otherwise, if predModeIntra is equal to INTRA_ANGULAR2 or INTRA_ANGULAR66, the following applies:

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

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

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

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

wTL[ x ][ y ] = 0                                                              (8-85)
```

– Otherwise, if predModeIntra is less than or equal to INTRA_ANGULAR10, the following ordered steps apply:

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

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

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

```
refL[ x ][ y ] = 0                                                             (8-87)

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

$$wT[y] = (dX[x][y] < refW - 1) \ ? \ 32 \ >> \ ((y \ << \ 1) \ >> \ nScale) \ : \ 0 \tag{8-89}$$

$$wL[x] = 0 \tag{8-90}$$

$$wTL[x][y] = 0 \tag{8-91}$$

– Otherwise, if predModeIntra is greater than or equal to INTRA_ANGULAR58, the following ordered steps apply:

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

$$\begin{aligned} dYPos[x] &= ((x + 1) * invAngle + 2) \ >> \ 2 \\ dYFrac[x] &= dYPos[x] \ \& \ 63 \\ dYInt[x] &= dYPos[x] \ >> \ 6 \\ dY[x][y] &= y + dYInt[x] \end{aligned} \tag{8-92}$$

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

$$\begin{aligned} refL[x][y] = (dY[x][y] < refH - 1) \ ? \ ((64 - dYFrac[x]) * sideRef[dY[x][y]] + \\ dYFrac[x] * sideRef[dY[x][y] + 1] + 32) \ >> \ 6 \\ : \ 0 \end{aligned} \tag{8-93}$$

$$refT[x][y] = 0 \tag{8-94}$$

$$wT[y] = 0 \tag{8-95}$$

$$wL[x] = (dY[x][y] < refH - 1) \ ? \ 32 \ >> \ ((x \ << \ 1) \ >> \ nScale) \ : \ 0 \tag{8-96}$$

$$wTL[x][y] = 0 \tag{8-97}$$

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

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

$$\begin{aligned} predSamples[x][y] = clip1Cmp(\ (\ & refL[x][y] * wL[x] + refT[x][y] * wT[y] - \\ & p[-1][-1] * wTL[x][y] + \\ & (64 - wL[x] - wT[y] + wTL[x][y]) * predSamples[x][y] + 32 \ ) \\ & >> \ 6 \ ) \end{aligned} \tag{8-98}$$

8.3 Decoding process for coding units coded in inter prediction mode

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

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

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

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

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

   – If MotionModelIdc[xCb][yCb] is equal to 0, the following applies:

     – The derivation process for motion vector components and reference indices as specified in clause 8.3.2.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx], the chroma motion vectors mvCL0[xSbIdx][ySbIdx] and mvCL1[xSbIdx][ySbIdx], the reference indices refIdxL0 and refIdxL1 and the prediction list utilization

flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 as outputs.

– Otherwise (MotionModelIdc[xCb][yCb] is equal to 1 or 2), the following applies:

  – The derivation process for affine motion vector components and reference indices as specified in clause 8.3.3.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight as inputs, and the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0 and predFlagL1, the luma motion vector array mvLX[xSbIdx][ySbIdx], and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, and ySbIdx = 0..(cbHeight >> 2) − 1, with X being 0 or 1 as outputs.

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

   – The variables numSbX and numSbY are modified as follows

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

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

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

3. The variables NumSbX[xCb][yCb] and NumSbY[xCb][yCb] are set equal to numSbX and numSbY, respectively.

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

   – The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location (xTb0, yTb0) set equal to the luma location (xCb, yCb), the width nTbW set equal to the luma coding block width cbWidth, the height nTbH set equal to the luma coding block height cbHeight and the variable cIdxset equal to 0 as inputs, and the array $resSamples_L$ as output.

   – The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location (xTb0, yTb0) set equal to the chroma location (xCb / 2, yCb / 2), the width nTbW set equal to the chroma coding block width cbWidth / 2, the height nTbH set equal to the chroma coding block height cbHeight / 2 and the variable cIdxset equal to 1 as inputs, and the array $resSamples_{Cb}$ as output.

   – The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location (xTb0, yTb0) set equal to the chroma location (xCb / 2, yCb / 2), the width nTbW set equal to the chroma coding block width cbWidth / 2, the height nTbH set equal to the chroma coding block height cbHeight / 2 and the variable cIdxset equal to 2 as inputs, and the array $resSamples_{Cr}$ as output.

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

   – The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location (xB, yB) set equal to (xCb, yCb), the block width bWidth set equal to cbWidth, the block height bHeight set equal to cbHeight, the variable cIdx set equal to 0, the (cbWidth)x(cbHeight) array predSamples set equal to $predSamples_L$ and the (cbWidth)x(cbHeight) array resSamples set equal to $resSamples_L$ as inputs, and the output is a modified reconstructed picture before in-loop filtering.

   – The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location (xB, yB) set equal to (xCb / 2, yCb / 2), the block width bWidth set equal to cbWidth / 2, the block height bHeight set equal to cbHeight / 2, the variable cIdx set equal to 1, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to $predSamples_{Cb}$ and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to $resSamples_{Cb}$ as inputs, and the output is a modified reconstructed picture before in-loop filtering.

   – The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location (xB, yB) set equal to (xCb / 2, yCb / 2), the block width bWidth set equal to cbWidth / 2, the

block height bHeight set equal to cbHeight / 2, the variable cIdx set equal to 2, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to $predSamples_{Cb}$ and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to $resSamples_{Cb}$ as inputs, and the output is a modified reconstructed picture before in-loop filtering.

8.3.2 Derivation process for motion vector components and reference indices

8.3.2.1 General

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the chroma motion vectors in 1/32 fractional-sample accuracy mvCL0[xSbIdx][ySbIdx] and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

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

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

– If merge_flag[xCb][yCb] is equal to 1, the derivation process for luma motion vectors for merge mode as specified in clause 8.3.2.2 is invoked with the luma location (xCb, yCb), the variables cbWidth and cbHeight inputs, and the output being the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[xSbIdx][ySbIdx], mvL1[xSbIdx][ySbIdx], the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

– Otherwise, for X being replaced by either 0 or 1 in the variables predFlagLX, mvLX and refIdxLX, in PRED_LX, and in the syntax elements ref_idx_lX and MvdLX, the following ordered steps apply:

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

$$numSbX = 1 \tag{8-101}$$

$$numSbY = 1 \tag{8-102}$$

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

   - If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

$$refIdxLX = ref\_idx\_lX[\ xCb\ ][\ yCb\ ] \tag{8-103}$$

$$predFlagLX[\ 0\ ][0\ ] = 1 \tag{8-104}$$

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

$$refIdxLX = -1 \tag{8-105}$$

$$predFlagLX[\ 0\ ][0\ ] = 0 \tag{8-106}$$

3. The variable mvdLX is derived as follows:

$$mvdLX[\ 0\ ] = MvdLX[\ xCb\ ][\ yCb\ ][\ 0\ ] \quad (8\text{-}107)$$

$$mvdLX[\ 1\ ] = MvdLX[\ xCb\ ][\ yCb\ ][\ 1\ ] \quad (8\text{-}108)$$

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

$$uLX[\ 0\ ] = (\ mvpLX[\ 0\ ] + mvdLX[\ 0\ ] + 2^{18}\ )\ \%\ 2^{18} \quad (8\text{-}109)$$

$$mvLX[\ 0\ ][0\ ][\ 0\ ] = (\ uLX[\ 0\ ]\ >=\ 2^{17}\ )\ ?\ (\ uLX[\ 0\ ] - 2^{18}\ )\ :\ uLX[\ 0\ ] \quad (8\text{-}110)$$

$$uLX[\ 1\ ] = (\ mvpLX[\ 1\ ] + mvdLX[\ 1\ ] + 2^{18}\ )\ \%\ 2^{18} \quad (8\text{-}111)$$

$$mvLX[\ 0\ ][0\ ][\ 1\ ] = (\ uLX[\ 1\ ]\ >=\ 2^{17}\ )\ ?\ (\ uLX[\ 1\ ] - 2^{18}\ )\ :\ uLX[\ 1\ ] \quad (8\text{-}112)$$

NOTE 1– The resulting values of mvLX[0] and mvLX[1] as specified above will always be in the range of -2^{17} to $2^{17} - 1$, inclusive.

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

8.3.2.2 Derivation process for luma motion vectors for merge mode

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

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

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

1. The derivation process for merging candidates from neighbouring coding units as specified in clause 8.3.2.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, and the luma coding block height cbHeight as inputs, and the output being the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$ and $availableFlagB_2$, the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$ and $refIdxLXB_2$, the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$ and $predFlagLXB_2$, and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$ and $mvLXB_2$, with X being 0 or 1.
2. The derivation process for subblock-based temporal merging candidates as specified in clause 8.3.2.4 is invoked with the luma location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight , the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$, the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$, the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$ and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$ as inputs and the output being the availability flag availableFlagSbCol, the number of luma coding subblocks in

horizontal direction numSbX and in vertical direction numSbY, the reference indices refIdxLXSbCol, the luma motion vectors mvLXSbCol[xSbIdx][ySbIdx] and the prediction list utilization flags predFlagLXSbCol[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1.

3. The reference indices for the temporal merging candidate, refIdxLXCol, with X being 0 or 1, are set equal to 0.

4. The derivation process for temporal luma motion vector prediction as specified in in clause 8.3.2.11 is invoked with the luma location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL0Col as inputs, and the output being the availability flag availableFlagL0Col and the temporal motion vector mvL0Col. The variables availableFlagCol, predFlagL0Col and predFlagL1Col are derived as follows:

availableFlagCol = availableFlagL0Col (8-113)

predFlagL0Col = availableFlagL0Col (8-114)

predFlagL1Col = 0 (8-115)

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

availableFlagCol = availableFlagL0Col || availableFlagL1Col (8-116)

predFlagL1Col = availableFlagL1Col (8-117)

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

```
i = 0
if( availableFlagA1 )
    mergeCandList[ i++ ] = A1
if( availableFlagB1 )
    mergeCandList[ i++ ] = B1
if( availableFlagB0 )
    mergeCandList[ i++ ] = B0                                   (8-118)
if( availableFlagA0 )
    mergeCandList[ i++ ] = A0
if( availableFlagSbCol )
    mergeCandList[ i++ ] = SbCol
if( availableFlagB2 )
    mergeCandList[ i++ ] = B2
if( availableFlagCol )
    mergeCandList[ i++ ] = Col
```

7. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the mergeCandList.

8. When slice_type is equal to B, the derivation process for combined bi-predictive merging candidates specified in clause 8.3.2.6 is invoked with mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList, numCurrMergeCand and numOrigMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices refIdxL0combCand$_k$ and refIdxL1combCand$_k$, the prediction list utilization flags predFlagL0combCand$_k$ and predFlagL1combCand$_k$ and the motion vectors mvL0combCand$_k$ and mvL1combCand$_k$ of every new candidate combCand$_k$ being added into mergeCandList. The number of candidates being added, numCombMergeCand, is set equal to (numCurrMergeCand − numOrigMergeCand). When numCombMergeCand is greater than 0, k ranges from 0 to numCombMergeCand − 1, inclusive.

9. The derivation process for zero motion vector merging candidates specified in clause 8.3.2.7 is invoked with the mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList and numCurrMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference

indices refIdxL0zeroCand$_m$ and refIdxL1zeroCand$_m$, the prediction list utilization flags predFlagL0zeroCand$_m$ and predFlagL1zeroCand$_m$ and the motion vectors mvL0zeroCand$_m$ and mvL1zeroCand$_m$ of every new candidate zeroCand$_m$ being added into mergeCandList. The number of candidates being added, numZeroMergeCand, is set equal to (numCurrMergeCand − numOrigMergeCand − numCombMergeCand). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand − 1, inclusive.

10. The following applies:

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

refIdxLX = refIdxLXSbCol (8-119)

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

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

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

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

numSbX = 1 (8-123)

numSbY = 1 (8-124)

refIdxLX = refIdxLXN (8-125)

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

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

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

8.3.2.3 Derivation process for spatial merging candidates

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are as follows, with X being 0 or 1:

– the availability flags availableFlagA$_0$, availableFlagA$_1$, availableFlagB$_0$, availableFlagB$_1$ and availableFlagB$_2$ of the neighbouring coding units,

– the reference indices refIdxLXA$_0$, refIdxLXA$_1$, refIdxLXB$_0$, refIdxLXB$_1$ and refIdxLXB$_2$ of the neighbouring coding units,

– the prediction list utilization flags predFlagLXA$_0$, predFlagLXA$_1$, predFlagLXB$_0$, predFlagLXB$_1$ and predFlagLXB$_2$ of the neighbouring coding units,

– the motion vectors in 1/16 fractional-sample accuracy mvLXA$_0$, mvLXA$_1$, mvLXB$_0$, mvLXB$_1$ and mvLXB$_2$ of the neighbouring coding units.

For the derivation of availableFlagA$_1$, refIdxLXA$_1$, predFlagLXA$_1$ and mvLXA$_1$ the following applies:

– The luma location (xNbA$_1$, yNbA$_1$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + cbHeight − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the

neighbouring luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the block availability flag $availableA_1$.

– The variables $availableFlagA_1$, $refIdxLXA_1$, $predFlagLXA_1$ and $mvLXA_1$ are derived as follows:

  – If $availableA_1$ is equal to FALSE, $availableFlagA_1$ is set equal to 0, both components of $mvLXA_1$ are set equal to 0, $refIdxLXA_1$ is set equal to −1 and $predFlagLXA_1$ is set equal to 0, with X being 0 or 1.

  – Otherwise, $availableFlagA_1$ is set equal to 1 and the following assignments are made:

$$mvLXA_1 = MvLX[\ xNbA_1\ ][\ yNbA_1\ ] \tag{8-129}$$

$$refIdxLXA_1 = RefIdxLX[\ xNbA_1\ ][\ yNbA_1\ ] \tag{8-130}$$

$$predFlagLXA_1 = PredFlagLX[\ xNbA_1\ ][\ yNbA_1\ ] \tag{8-131}$$

For the derivation of $availableFlagB_1$, $refIdxLXB_1$, $predFlagLXB_1$ and $mvLXB_1$ the following applies:

– The luma location ($xNbB_1$, $yNbB_1$) inside the neighbouring luma coding block is set equal to (xCb + cbWidth − 1, yCb − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_1$, $yNbB_1$) as inputs, and the output is assigned to the block availability flag $availableB_1$.

– The variables $availableFlagB_1$, $refIdxLXB_1$, $predFlagLXB_1$ and $mvLXB_1$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagB_1$ is set equal to 0, both components of $mvLXB_1$ are set equal to 0, $refIdxLXB_1$ is set equal to −1 and $predFlagLXB_1$ is set equal to 0, with X being 0 or 1:

    – $availableB_1$ is equal to FALSE.

    – $availableA_1$ is equal to TRUE and the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbB_1$, $yNbB_1$) have the same motion vectors and the same reference indices.

  – Otherwise, $availableFlagB_1$ is set equal to 1 and the following assignments are made:

$$mvLXB_1 = MvLX[\ xNbB_1\ ][\ yNbB_1\ ] \tag{8-132}$$

$$refIdxLXB_1 = RefIdxLX[\ xNbB_1\ ][\ yNbB_1\ ] \tag{8-133}$$

$$predFlagLXB_1 = PredFlagLX[\ xNbB_1\ ][\ yNbB_1\ ] \tag{8-134}$$

For the derivation of $availableFlagB_0$, $refIdxLXB_0$, $predFlagLXB_0$ and $mvLXB_0$ the following applies:

– The luma location ($xNbB_0$, $yNbB_0$) inside the neighbouring luma coding block is set equal to (xCb + cbWidth, yCb − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_0$, $yNbB_0$) as inputs, and the output is assigned to the block availability flag $availableB_0$.

– The variables $availableFlagB_0$, $refIdxLXB_0$, $predFlagLXB_0$ and $mvLXB_0$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagB_0$ is set equal to 0, both components of $mvLXB_0$ are set equal to 0, $refIdxLXB_0$ is set equal to −1 and $predFlagLXB_0$ is set equal to 0, with X being 0 or 1:

    – $availableB_0$ is equal to FALSE.

    – $availableB_1$ is equal to TRUE and the luma locations ($xNbB_1$, $yNbB_1$) and ($xNbB_0$, $yNbB_0$) have the same motion vectors and the same reference indices.

  – Otherwise, $availableFlagB_0$ is set equal to 1 and the following assignments are made:

$$mvLXB_0 = MvLX[\ xNbB_0\ ][\ yNbB_0\ ] \tag{8-135}$$

$$refIdxLXB_0 = RefIdxLX[\ xNbB_0\ ][\ yNbB_0\ ] \tag{8-136}$$

$$predFlagLXB_0 = PredFlagLX[\ xNbB_0\ ][\ yNbB_0\ ] \quad (8\text{-}137)$$

For the derivation of $availableFlagA_0$, $refIdxLXA_0$, $predFlagLXA_0$ and $mvLXA_0$ the following applies:

– The luma location ($xNbA_0$, $yNbA_0$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + cbWidth).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbA_0$, $yNbA_0$) as inputs, and the output is assigned to the block availability flag $availableA_0$.

– The variables $availableFlagA_0$, $refIdxLXA_0$, $predFlagLXA_0$ and $mvLXA_0$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagA_0$ is set equal to 0, both components of $mvLXA_0$ are set equal to 0, $refIdxLXA_0$ is set equal to −1 and $predFlagLXA_0$ is set equal to 0, with X being 0 or 1:

    – $availableA_0$ is equal to FALSE.

    – $availableA_1$ is equal to TRUE and the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbA_0$, $yNbA_0$) have the same motion vectors and the same reference indices.

  – Otherwise, $availableFlagA_0$ is set equal to 1 and the following assignments are made:

$$mvLXA_0 = MvLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}138)$$

$$refIdxLXA_0 = RefIdxLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}139)$$

$$predFlagLXA_0 = PredFlagLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}140)$$

For the derivation of $availableFlagB_2$, $refIdxLXB_2$, $predFlagLXB_2$ and $mvLXB_2$ the following applies:

– The luma location ($xNbB_2$, $yNbB_2$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_2$, $yNbB_2$) as inputs, and the output is assigned to the block availability flag $availableB_2$.

– The variables $availableFlagB_2$, $refIdxLXB_2$, $predFlagLXB_2$ and $mvLXB_2$ are derived as follows:

  – If one or more of the following conditions are true, $availableFlagB_2$ is set equal to 0, both components of $mvLXB_2$ are set equal to 0, $refIdxLXB_2$ is set equal to −1 and $predFlagLXB_2$ is set equal to 0, with X being 0 or 1:

    – $availableB_2$ is equal to FALSE.

    – $availableA_1$ is equal to TRUE and the luma locations ($xNbA_1$, $yNbA_1$) and ($xNbB_2$, $yNbB_2$) have the same motion vectors and the same reference indices.

    – $availableB_1$ is equal to TRUE and the luma locations ($xNbB_1$, $yNbB_1$) and ($xNbB_2$, $yNbB_2$) have the same motion vectors and the same reference indices.

    – $availableFlagA_0$ + $availableFlagA_1$ + $availableFlagB_0$ + $availableFlagB_1$ is equal to 4.

  – Otherwise, $availableFlagB_2$ is set equal to 1 and the following assignments are made:

$$mvLXB_2 = MvLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}141)$$

$$refIdxLXB_2 = RefIdxLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}142)$$

$$predFlagLXB_2 = PredFlagLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}143)$$

8.3.2.4 Derivation process for subblock-based temporal merging candidates

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

– the availability flags availableFlagA$_0$, availableFlagA$_1$, availableFlagB$_0$, and availableFlagB$_1$ of the neighbouring coding units,

– the reference indices refIdxLXA$_0$, refIdxLXA$_1$, refIdxLXB$_0$, and refIdxLXB$_1$ of the neighbouring coding units,

– the prediction list utilization flags predFlagLXA$_0$, predFlagLXA$_1$, predFlagLXB$_0$, and predFlagLXB$_1$ of the neighbouring coding units,

– the motion vectors in 1/16 fractional-sample accuracy mvLXA$_0$, mvLXA$_1$, mvLXB$_0$, and mvLXB$_1$ of the neighbouring coding units.

Outputs of this process are:

– the availability flag availableFlagSbCol,

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

– the reference indices refIdxL0SbCol and refIdxL1SbCol,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0SbCol[xSbIdx][ySbIdx] and mvL1SbCol[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the prediction list utilization flags predFlagL0SbCol[xSbIdx][ySbIdx] and predFlagL1SbCol[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

The availability flag availableFlagSbCol is derived as follows.

– If either slice_temporal_mvp_enable_flag or sps_sbtmvp_flag is equal to 0, availableFlagSbCol is set equal to 0.

– Otherwise, the following ordered steps apply.

1. The location (xCtb, yCtb) of the top-left sample of the luma coding tree block that contains the current coding block and the location (xCtr, yCtr) of the below-right center sample of the current luma coding block are derived as follows:

$$xCtb = (xCb >> CtuLog2Size) << CtuLog2Size \quad (8\text{-}144)$$

$$yCtb = (yCb >> CtuLog2Size) << CtuLog2Size \quad (8\text{-}145)$$

$$xCtr = xCb + (cbWidth / 2) \quad (8\text{-}146)$$

$$yCtr = yCb + (cbHeight / 2) \quad (8\text{-}147)$$

2. The luma location (xColCtrCb, yColCtrCb) is set equal to the top-left sample of the collocated luma coding block covering the location given by (xCtr, yCtr) inside ColPic relative to the top-left luma sample of the collocated picture specified by ColPic.

3. The derivation process for subblock-based temporal merging base motion data as specified in clause 8.3.2.5 is invoked with the location (xCtb, yCtb), the location (xColCtrCb, yColCtrCb), the availability flags availableFlagA$_0$, availableFlagA$_1$, availableFlagB$_0$ and availableFlagB$_1$, and the prediction list utilization flags predFlagLXA$_0$, predFlagLXA$_1$, predFlagLXB$_0$ and predFlagLXB$_1$, and the reference indices refIdxLXA$_0$, refIdxLXA$_1$, refIdxLXB$_0$ and refIdxLXB$_1$, and the motion vectors mvLXA$_0$, mvLXA$_1$, mvLXB$_0$ and mvLXB$_1$, with X being 0 and 1 as inputs and the motion vectors ctrMvLX, the prediction list utilization flags ctrPredFlagLX and the reference indices ctrRefIdxLX of the collocated block, with X being 0 and 1, and the temporal motion vector tempMV as outputs.

4. The variable availableFlagSbCol is derived as follows:

   – If both ctrPredFlagL0 and ctrPredFlagL1 are equal to 0, availableFlagSbCol is set equal to 0.

   – Otherwise, availableFlagSbCol is set equal to 1.

When availableFlagSbCol is equal to 1, the following applies:

– The variables numSbX, numSbY, sbWidth, sbHeight and refIdxLXSbCol are derived as follows:

$$numSbX = max(cbWidth >> Log2SbtmvpSize, 1) \quad (8\text{-}148)$$

numSbY = max(cbHeight >> Log2SbtmvpSize, 1) (8-149)

sbWidth = cbWidth / numSbX (8-150)

sbHeight = cbHeight / numSbY (8-151)

refIdxLXSbCol = 0 (8-152)

– For xSbIdx = 0..numSbX − 1 and ySbIdx = 0 .. numSbY − 1, the motion vectors mvLXSbCol[xSbIdx][ySbIdx] and prediction list utilization flags predFlagLXSbCol[xSbIdx][ySbIdx] are derived as follows:

  – The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

    xSb = xCb + xSbIdx * sbWidth (8-153)

    ySb = yCb + ySbIdx * sbHeight (8-154)

  – The location (xColSb, yColSb) of the collocated subblock inside ColPic is derived as follows.

    xColSb = Clip3(xCtb,
    Min(CurPicWidthInSamplesY − 1, xCtb + (1 << CtbLog2SizeY) +3), (8-155)
    xSb + (tempMv[0] >> 4))

    yColSb = Clip3(yCtb,
    Min(CurPicHeightInSamplesY − 1, yCtb + (1 << CtbLog2SizeY) +3), (8-156)
    ySb + (tempMv[1] >> 4))

  – The variable currCb specifies the luma coding block covering the current coding subblock inside the current picture.

  – The variable colCb specifies the luma coding block covering the modified location given by ((xColSb >> 3) << 3, (yColSb >> 3) << 3) inside the ColPic.

  – The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.

  – The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxL0 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL0SbCol[xSbIdx][ySbIdx] and availableFlagL0SbCol.

  – The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxL1 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL1SbCol[xSbIdx][ySbIdx] and availableFlagL1SbCol.

  – When availableFlagL0SbCol and availableFlagL1SbCol are both equal to 0, the following applies:

    mvLXSbCol[xSbIdx][ySbIdx] = ctrMvLX (8-157)

    predFlagLXSbCol[xSbIdx][ySbIdx] = ctrPredFlagLX (8-158)

8.3.2.5 Derivation process for subblock-based temporal merging base motion data

Inputs to this process are:

– the location (xCtb, yCtb) of the top-left sample of the luma coding tree block that contains the current coding block,

– the location (xColCtrCb, yColCtrCb) of the top-left sample of the collocated luma coding block that covers the below-right center sample.

– the availability flags availableFlagA_0, availableFlagA_1, availableFlagB_0, and availableFlagB_1 of the neighbouring coding units,

– the reference indices refIdxLXA_0, refIdxLXA_1, refIdxLXB_0, and refIdxLXB_1 of the neighbouring coding units,

– the prediction list utilization flags predFlagLXA$_0$, predFlagLXA$_1$, predFlagLXB$_0$, and predFlagLXB$_1$ of the neighbouring coding units,

– the motion vectors in 1/16 fractional-sample accuracy mvLXA$_0$, mvLXA$_1$, mvLXB$_0$, and mvLXB$_1$ of the neighbouring coding units.

Outputs of this process are:

– the motion vectors ctrMvL0 and ctrMvL1,

– the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1,

– the reference indices ctrRefIdxL0 and ctrRefIdxL1,

– the temporal motion vector tempMV.

The variables tempMv, numRefLists and bTerminate are set as follows:

tempMv[0] = 0 (8-159)

tempMv[0] = 0 (8-160)

numRefLists = (slice_type = = B) ? 2: 1 (8-161)

bTerminate = FALSE (8-162)

The variable currPic specifies the current picture.

For each reference picture list LX with X ranging from 0 to (numRefList − 1), mvTemp is derived as follows:

1. When all of the following conditions are equal to true, X is set equal to !X:
   – DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice,
   – slice_type is equal to B,
   – collocated_from_l0_flag is equal to 0.
2. When bTerminate is equal to FALSE, and availableFlagA0 and predFlagLXA0 are equal to 1 and DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXA0]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXA0:
3. When bTerminate is equal to FALSE, availableFlagLB0 and predFlagLXB0 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXB0]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXB0.
4. When bTerminate is equal to FALSE, availableFlagB1 and predFlagLXB1 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXB1]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXB1.
5. When bTerminate is equal to FALSE, availableFlagA1 and predFlagLXB1 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXA1]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set to mvLXA1.

The location (xColCb, yColCb) of the collocated block inside ColPic is derived as follows.

```
xColCb = Clip3( xCtb,
                Min( CurPicWidthInSamplesY − 1, xCtb + ( 1 << CtbLog2SizeY ) +3 ),
                xColCtrCb + ( tempMv[0] >> 4 ) )                                    (8-163)

yColCb = Clip3( yCtb,
                Min( CurPicHeightInSamplesY − 1, yCtb + ( 1 << CtbLog2SizeY ) +3 ),
                yColCtrCb + ( tempMv[1] >> 4 ) )                                    (8-164)
```

The array colPredMode is set equal to the prediction mode array CuPredMode of the collocated picture specified by ColPic.

The motion vectors ctrMvL0 and ctrMvL1, the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1, and the reference indices ctrRefIdxL0 and ctrRefIdxL1 are derived as follows:

– If colPredMode[xColCb][yColCb] is equal to MODE_INTER, the following applies:

– The variable currCb specifies the luma coding block covering (xCtrCb ,yCtrCb) inside the current picture.

– The variable colCb specifies the luma coding block covering the modified location given by ((xColCb >> 3) << 3, (yColCb >> 3) << 3) inside the ColPic.

– The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.

– The derivation process for temporal motion vector prediction in subclause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL0, and sbFlag set equal to 1 as inputs and the output being assigned to centerMvL0 and centerPredFlagL0.

– The derivation process for temporal motion vector prediction in subclause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL1, and sbFlag set equal to 1 as inputs and the output being assigned to centerMvL1 and centerPredFlagL1.

– Otherwise, the following applies:

ctrPredFlagL0 = 0 (8-165)

ctrPredFlagL1 = 0 (8-166)

8.3.2.6 Derivation process for combined bi-predictive merging candidates

Inputs to this process are:

– a merging candidate list mergeCandList,

– the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,

– the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,

– the motion vectors in 1/16 fractional-sample accuracy mvL0N and mvL1N of every candidate N in mergeCandList,

– the number of elements numCurrMergeCand within mergeCandList,

– the number of elements numOrigMergeCand within the mergeCandList after the spatial and temporal merge candidate derivation process.

Outputs of this process are:

– the merging candidate list mergeCandList,

– the number of elements numCurrMergeCand within mergeCandList,

– the reference indices refIdxL0combCand$_k$ and refIdxL1combCand$_k$ of every new candidate combCand$_k$ added into mergeCandList during the invocation of this process,

– the prediction list utilization flags predFlagL0combCand$_k$ and predFlagL1combCand$_k$ of every new candidate combCand$_k$ added into mergeCandList during the invocation of this process,

– the motion vectors in 1/16 fractional-sample accuracy mvL0combCand$_k$ and mvL1combCand$_k$ of every new candidate combCand$_k$ added into mergeCandList during the invocation of this process.

When numOrigMergeCand is greater than 1 and less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable combIdx is set equal to 0, the variable combStop is set equal to FALSE and the following ordered steps are repeated until combStop is equal to TRUE:

1. The variables l0CandIdx and l1CandIdx are derived using combIdx as specified in Table 8-6.
2. The following assignments are made, with l0Cand being the candidate at position l0CandIdx and l1Cand being the candidate at position l1CandIdx in the merging candidate list mergeCandList:
   – l0Cand = mergeCandList[l0CandIdx]
   – l1Cand = mergeCandList[l1CandIdx]
3. When all of the following conditions are true:
   – predFlagL0l0Cand = = 1
   – predFlagL1l1Cand = = 1

– (DiffPicOrderCnt(RefPicList0[refIdxL0l0Cand], RefPicList1[refIdxL1l1Cand]) != 0) || (mvL0l0Cand != mvL1l1Cand)

the candidate $combCand_k$ with k equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $combCand_k$, and the reference indices, the prediction list utilization flags and the motion vectors of $combCand_k$ are derived as follows and numCurrMergeCand is incremented by 1:

$refIdxL0combCand_k = refIdxL0l0Cand$ (8-167)

$refIdxL1combCand_k = refIdxL1l1Cand$ (8-168)

$predFlagL0combCand_k = 1$ (8-169)

$predFlagL1combCand_k = 1$ (8-170)

$mvL0combCand_k[0] = mvL0l0Cand[0]$ (8-171)

$mvL0combCand_k[1] = mvL0l0Cand[1]$ (8-172)

$mvL1combCand_k[0] = mvL1l1Cand[0]$ (8-173)

$mvL1combCand_k[1] = mvL1l1Cand[1]$ (8-174)

numCurrMergeCand = numCurrMergeCand + 1 (8-175)

4. The variable combIdx is incremented by 1.
5. When combIdx is equal to (numOrigMergeCand * (numOrigMergeCand − 1)) or numCurrMergeCand is equal to MaxNumMergeCand, combStop is set equal to TRUE.

Table 8-6 – Specification of l0CandIdx and l1CandIdx

combIdx	0	1	2	3	4	5	6	7	8	9	10	11
l0CandIdx	0	1	0	2	1	2	0	3	1	3	2	3
l1CandIdx	1	0	2	0	2	1	3	0	3	1	3	2

8.3.2.7 Derivation process for zero motion vector merging candidates

Inputs to this process are:

– a merging candidate list mergeCandList,
– the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,
– the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,
– the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList,
– the number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

– the merging candidate list mergeCandList,
– the number of elements numCurrMergeCand within mergeCandList,
– the reference indices $refIdxL0zeroCand_m$ and $refIdxL1zeroCand_m$ of every new candidate $zeroCand_m$ added into mergeCandList during the invocation of this process,
– the prediction list utilization flags $predFlagL0zeroCand_m$ and $predFlagL1zeroCand_m$ of every new candidate $zeroCand_m$ added into mergeCandList during the invocation of this process,
– the motion vectors $mvL0zeroCand_m$ and $mvL1zeroCand_m$ of every new candidate $zeroCand_m$ added into mergeCandList during the invocation of this process.

The variable numRefIdx is derived as follows:

– If slice_type is equal to P, numRefIdx is set equal to num_ref_idx_l0_active_minus1 + 1.

– Otherwise (slice_type is equal to B), numRefIdx is set equal to Min(num_ref_idx_l0_active_minus1 + 1, num_ref_idx_l1_active_minus1 + 1).

When numCurrMergeCand is less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable zeroIdx is set equal to 0 and the following ordered steps are repeated until numCurrMergeCand is equal to MaxNumMergeCand:

1. For the derivation of the reference indices, the prediction list utilization flags and the motion vectors of the zero motion vector merging candidate, the following applies:

   – If slice_type is equal to P, the candidate $zeroCand_m$ with m equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $zeroCand_m$, and the reference indices, the prediction list utilization flags and the motion vectors of $zeroCand_m$ are derived as follows and numCurrMergeCand is incremented by 1:

   $$refIdxL0zeroCand_m = (zeroIdx < numRefIdx) ? zeroIdx : 0 \quad (8\text{-}176)$$

   $$refIdxL1zeroCand_m = -1 \quad (8\text{-}177)$$

   $$predFlagL0zeroCand_m = 1 \quad (8\text{-}178)$$

   $$predFlagL1zeroCand_m = 0 \quad (8\text{-}179)$$

   $$mvL0zeroCand_m[0] = 0 \quad (8\text{-}180)$$

   $$mvL0zeroCand_m[1] = 0 \quad (8\text{-}181)$$

   $$mvL1zeroCand_m[0] = 0 \quad (8\text{-}182)$$

   $$mvL1zeroCand_m[1] = 0 \quad (8\text{-}183)$$

   $$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}184)$$

   – Otherwise (slice_type is equal to B), the candidate $zeroCand_m$ with m equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $zeroCand_m$, and the reference indices, the prediction list utilization flags and the motion vectors of $zeroCand_m$ are derived as follows and numCurrMergeCand is incremented by 1:

   $$refIdxL0zeroCand_m = (zeroIdx < numRefIdx) ? zeroIdx : 0 \quad (8\text{-}185)$$

   $$refIdxL1zeroCand_m = (zeroIdx < numRefIdx) ? zeroIdx : 0 \quad (8\text{-}186)$$

   $$predFlagL0zeroCand_m = 1 \quad (8\text{-}187)$$

   $$predFlagL1zeroCand_m = 1 \quad (8\text{-}188)$$

   $$mvL0zeroCand_m[0] = 0 \quad (8\text{-}189)$$

   $$mvL0zeroCand_m[1] = 0 \quad (8\text{-}190)$$

   $$mvL1zeroCand_m[0] = 0 \quad (8\text{-}191)$$

   $$mvL1zeroCand_m[1] = 0 \quad (8\text{-}192)$$

   $$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}193)$$

2. The variable zeroIdx is incremented by 1.

8.3.2.8 Derivation process for luma motion vector prediction

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the prediction mvpLX in 1/16 fractional-sample accuracy of the motion vector mvLX, with X being 0 or 1.

The motion vector predictor mvpLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for motion vector predictor candidate list as specified in clause 8.3.2.9 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1, as output.
2. The motion vector predictor mvpLX with X being 0 or 1 is derived as follows:

$$mvpLX = mvpListLX[\ mvp\_lX\_flag[\ xCb\ ][\ yCb\ ]\ ] \tag{8-194}$$

8.3.2.9 Derivation process for motion vector predictor candidate list

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is motion vector predictor candidate list mvpListLX in 1/16 fractional-sample accuracy with X being 0 or 1.

The motion vector predictor candidate list mvpListLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for spatial motion vector predictor candidates from neighbouring coding unit partitions as specified in clause 8.3.2.10 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the availability flags availableFlagLXN and the motion vectors mvLXN, with N being replaced by A or B, as output.
2. The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLXN, with N being replaced by A or B, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXN, with N being replaced by A or B, as output.
3. If both availableFlagLXA and availableFlagLXB are equal to 1 and mvLXA is not equal to mvLXB, availableFlagLXCol is set equal to 0.
4. Otherwise, the following applies:
   – The derivation process for temporal luma motion vector prediction as specified in clause 8.3.2.11 is with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
   – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLXCol, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXCol as output.
5. The motion vector predictor candidate list, mvpListLX, is constructed as follows:

```
i = 0
if( availableFlagLXA ) {
    mvpListLX[ i++ ] = mvLXA
    if( availableFlagLXB  &&  ( mvLXA  !=  mvLXB ) )
        mvpListLX[ i++ ] = mvLXB
} else if( availableFlagLXB )
    mvpListLX[ i++ ] = mvLXB                                    (8-195)
if( i < 2  &&  availableFlagLXCol )
    mvpListLX[ i++ ] = mvLXCol
while( i < 2 ) {
    mvpListLX[ i ][ 0 ] = 0
    mvpListLX[ i ][ 1 ] = 0
    i++
}
```

8.3.2.10 Derivation process for spatial motion vector predictor candidates

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

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

– the motion vectors mvLXN in 1/16 fractional-sample accuracy of the neighbouring coding units,

– the availability flags availableFlagLXN of the neighbouring coding units.

Figure 8-2 provides an overview of spatial motion vector neighbours.

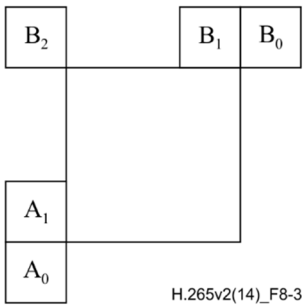


Figure 8-2 – Spatial motion vector neighbours (informative)

The variable currCb specifies the current luma coding block at luma location (xCb, yCb) and the variable currPic specifies the current picture.

The variable isScaledFlagLX, with X being 0 or 1, is set equal to 0.

The motion vector mvLXA and the availability flag availableFlagLXA are derived in the following ordered steps:

1. The sample location ($xNbA_0$, $yNbA_0$) is set equal to (xCb − 1, yCb + cbHeight) and the sample location ($xNbA_1$, $yNbA_1$) is set equal to ($xNbA_0$, $yNbA_0$ − 1).
2. The availability flag availableFlagLXA is set equal to 0 and both components of mvLXA are set equal to 0.
3. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbA_0$, $yNbA_0$) as inputs, and the output is assigned to the block availability flag $availableA_0$.

4. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the block availability flag $availableA_1$.
5. When $availableA_0$ or $availableA_1$ is equal to TRUE, the variable isScaledFlagLX is set equal to 1.
6. The following applies for ($xNbA_k$, $yNbA_k$) from ($xNbA_0$, $yNbA_0$) to ($xNbA_1$, $yNbA_1$):
   - When $availableA_k$ is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
     - If PredFlagLX[$xNbA_k$][$yNbA_k$] is equal to 1 and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbA_k$][$yNbA_k$]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

       $$mvLXA = MvLX[xNbA_k][yNbA_k] \quad (8\text{-}196)$$

     - Otherwise, when PredFlagLY[$xNbA_k$][$yNbA_k$] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[$xNbA_k$][$yNbA_k$]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

       $$mvLXA = MvLY[xNbA_k][yNbA_k] \quad (8\text{-}197)$$

7. When availableFlagLXA is equal to 0, the following applies for ($xNbA_k$, $yNbA_k$) from ($xNbA_0$, $yNbA_0$) to ($xNbA_1$, $yNbA_1$) or until availableFlagLXA is equal to 1:
   - When $availableA_k$ is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
     - If PredFlagLX[$xNbA_k$][$yNbA_k$] is equal to 1 , availableFlagLXA is set equal to 1 and the following assignments are made:

       $$mvLXA = MvLX[xNbA_k][yNbA_k] \quad (8\text{-}198)$$

       $$refIdxA = RefIdxLX[xNbA_k][yNbA_k] \quad (8\text{-}199)$$

       $$refPicListA = RefPicListX \quad (8\text{-}200)$$

     - Otherwise, if PredFlagLY[$xNbA_k$][$yNbA_k$] (with Y = !X) is equal to 1, availableFlagLXA is set equal to 1 and the following assignments are made:

       $$mvLXA = MvLY[xNbA_k][yNbA_k] \quad (8\text{-}201)$$

       $$refIdxA = RefIdxLY[xNbA_k][yNbA_k] \quad (8\text{-}202)$$

       $$refPicListA = RefPicListY \quad (8\text{-}203)$$

   - When availableFlagLXA is equal to 1, DiffPicOrderCnt(refPicListA[refIdxA], RefPicListX[refIdxLX]) is not equal to 0, mvLXA is derived as follows:

     $$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}204)$$

     $$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}205)$$

     $$mvLXA = Clip3(-32768, 32767, Sign(distScaleFactor * mvLXA) * ((Abs(distScaleFactor * mvLXA) + 127) >> 8)) \quad (8\text{-}206)$$

     where td and tb are derived as follows:

     $$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, refPicListA[refIdxA])) \quad (8\text{-}207)$$

     $$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}208)$$

The motion vector mvLXB and the availability flag availableFlagLXB are derived in the following ordered steps:

1. The sample locations ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$) and ($xNbB_2$, $yNbB_2$) are set equal to (xCb + cbWidth, yCb − 1), (xCb + cbWidth − 1, yCb − 1) and (xCb − 1, yCb − 1), respectively.

2. The availability flag availableFlagLXB is set equal to 0 and the both components of mvLXB are set equal to 0.
3. The following applies for ($xNbB_k$, $yNbB_k$) from ($xNbB_0$, $yNbB_0$) to ($xNbB_2$, $yNbB_2$):
   - The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_k$, $yNbB_k$) as inputs, and the output is assigned to the block availability flag $availableB_k$.
   - When $availableB_k$ is equal to TRUE and availableFlagLXB is equal to 0, the following applies:
     - If $PredFlagLX[xNbB_k][yNbB_k]$ is equal to 1, and DiffPicOrderCnt(RefPicListX[$RefIdxLX[xNbB_k][yNbB_k]$], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

       $$mvLXB = MvLX[xNbB_k][yNbB_k] \quad (8\text{-}209)$$

       $$refIdxB = RefIdxLX[xNbB_k][yNbB_k] \quad (8\text{-}210)$$

     - Otherwise, when $PredFlagLY[xNbB_k][yNbB_k]$ (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[$RefIdxLY[xNbB_k][yNbB_k]$], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

       $$mvLXB = MvLY[xNbB_k][yNbB_k] \quad (8\text{-}211)$$

       $$refIdxB = RefIdxLY[xNbB_k][yNbB_k] \quad (8\text{-}212)$$

4. When isScaledFlagLX is equal to 0 and availableFlagLXB is equal to 1, availableFlagLXA is set equal to 1 and the following applies:

   $$mvLXA = mvLXB \quad (8\text{-}213)$$

5. When isScaledFlagLX is equal to 0, availableFlagLXB is set equal to 0 and the following applies for ($xNbB_k$, $yNbB_k$) from ($xNbB_0$, $yNbB_0$) to ($xNbB_2$, $yNbB_2$) or until availableFlagLXB is equal to 1:
   - The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_k$, $yNbB_k$) as inputs, and the output is assigned to the block availability flag $availableB_k$.
   - When $availableB_k$ is equal to TRUE and availableFlagLXB is equal to 0, the following applies:
     - If $PredFlagLX[xNbB_k][yNbB_k]$ is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

       $$mvLXB = MvLX[xNbB_k][yNbB_k] \quad (8\text{-}214)$$

       $$refIdxB = RefIdxLX[xNbB_k][yNbB_k] \quad (8\text{-}215)$$

       $$refPicListB = RefPicListX \quad (8\text{-}216)$$

     - Otherwise, when $PredFlagLY[xNbB_k][yNbB_k]$ (with Y = !X) is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

       $$mvLXB = MvLY[xNbB_k][yNbB_k] \quad (8\text{-}217)$$

       $$refIdxB = RefIdxLY[xNbB_k][yNbB_k] \quad (8\text{-}218)$$

       $$refPicListB = RefPicListY \quad (8\text{-}219)$$

   - When availableFlagLXB is equal to 1, DiffPicOrderCnt(refPicListB[refIdxB], RefPicListX[refIdxLX]) is not equal to 0, mvLXB is derived as follows:

     $$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}220)$$

     $$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}221)$$

$$mvLXB = Clip3(-32768, 32767, Sign(distScaleFactor * mvLXB) * ((Abs(distScaleFactor * mvLXB) + 127) >> 8)) \quad (8\text{-}222)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, refPicListB[refIdxB])) \quad (8\text{-}223)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}224)$$

8.3.2.11 Derivation process for temporal luma motion vector prediction

Inputs to this process are:

- a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a variable cbWidth specifying the width of the current coding block in luma samples,
- a variable cbHeight specifying the height of the current coding block in luma samples,
- a reference index refIdxLX, with X being 0 or 1.

Outputs of this process are:

- the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,
- the availability flag availableFlagLXCol.

The variable currCb specifies the current luma coding block at luma location (xCb, yCb).

The variables mvLXCol and availableFlagLXCol are derived as follows:

- If slice_temporal_mvp_enabled_flag is equal to 0, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
- Otherwise (slice_temporal_mvp_enabled_flag is equal to 1), the following ordered steps apply:

1. The bottom right collocated motion vector is derived as follows:

$$xColBr = xCb + cbWidth \quad (8\text{-}225)$$

$$yColBr = yCb + cbHeight \quad (8\text{-}226)$$

   - If yCb >> CtbLog2SizeY is equal to yColBr >> CtbLog2SizeY, yColBr is less than pic_height_in_luma_samples and xColBr is less than pic_width_in_luma_samples, the following applies:
     - The variable colCb specifies the luma coding block covering the modified location given by ((xColBr >> 3) << 3, (yColBr >> 3) << 3) inside the collocated picture specified by ColPic.
     - The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
     - The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.
   - Otherwise, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

2. When availableFlagLXCol is equal to 0, the central collocated motion vector is derived as follows:

$$xColCtr = xCb + (cbWidth >> 1) \quad (8\text{-}227)$$

$$yColCtr = yCb + (cbHeight >> 1) \quad (8\text{-}228)$$

   - The variable colCb specifies the luma coding block covering the modified location given by ((xColCtr >> 3) << 3, (yColCtr >> 3) << 3) inside the collocated picture specified by ColPic.

- The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
- The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

8.3.2.12 Derivation process for collocated motion vectors

Inputs to this process are:

- a variable currCb specifying the current coding block,
- a variable colCb specifying the collocated coding block inside the collocated picture specified by ColPic,
- a luma location (xColCb, yColCb) specifying the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic,
- a reference index refIdxLX, with X being 0 or 1,
- a flag indicating a subblock temporal merging candidate sbFlag.

Outputs of this process are:

- the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,
- the availability flag availableFlagLXCol.

The variable currPic specifies the current picture.

The arrays predFlagL0Col[x][y], mvL0Col[x][y] and refIdxL0Col[x][y] are set equal to PredFlagL0[x][y], MvL0[x][y] and RefIdxL0[x][y], respectively, of the collocated picture specified by ColPic, and the arrays predFlagL1Col[x][y], mvL1Col[x][y] and refIdxL1Col[x][y] are set equal to PredFlagL1[x][y], MvL1[x][y] and RefIdxL1[x][y], respectively, of the collocated picture specified by ColPic.

[Ed. (BB): Define ColPic NoBackwardPredFlag.]

The variables mvLXCol and availableFlagLXCol are derived as follows:

- If colCb is coded in an intra prediction mode, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
- Otherwise, the motion vector mvCol, the reference index refIdxCol and the reference list identifier listCol are derived as follows:
  - If sbFlag is equal to 0, availableFlagLXCol is set to 1 and the following applies:
    - If predFlagL0Col[xColCb][yColCb] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL1Col[xColCb][yColCb], refIdxL1Col[xColCb][yColCb] and L1, respectively.
    - Otherwise, if predFlagL0Col[xColCb][yColCb] is equal to 1 and predFlagL1Col[xColCb][yColCb] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL0Col[xColCb][yColCb], refIdxL0Col[xColCb][yColCb] and L0, respectively.
    - Otherwise (predFlagL0Col[xColCb][yColCb] is equal to 1 and predFlagL1Col[xColCb][yColCb] is equal to 1), the following assignments are made:
      - If NoBackwardPredFlag is equal to 1, mvCol, refIdxCol and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] and LX, respectively.
      - Otherwise, mvCol, refIdxCol and listCol are set equal to mvLNCol[xColCb][yColCb], refIdxLNCol[xColCb][yColCb] and LN, respectively, with N being the value of collocated_from_l0_flag.
  - Otherwise (sbFlag is equal to 1), the following applies:
    - If PredFlagLXCol[xColCb][yColCb] is equal to 1, mvCol, refIdxCol, and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb], and LX, respectively, availableFlagLXCol is set to 1.
    - Otherwise (PredFlagLXCol[xColCb][yColCb] is equal to 0), the following applies:

– If DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice and PredFlagLYCol[xColCb][yColCb] is equal to 1, mvCol, refIdxCol, and listCol are set to mvLYCol[xColCb][yColCb], refIdxLYCol[xColCb][yColCb] and LY, respectively, with Y being equal to !X where X being the value of X this process is invoked for. availableFlagLXCol is set to 1.

– Both the components of mvLXCol are set to 0 and availableFlagLXCol is set equal to 0.

– When availableFlagLXCol is equal to TRUE, mvLXCol and availableFlagLXCol are derived as follows:

  – If LongTermRefPic(currPic, currCb, refIdxLX, LX) is not equal to LongTermRefPic(ColPic, colCb, refIdxCol, listCol), both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

  – Otherwise, the variable availableFlagLXCol is set equal to 1, refPicListCol[refIdxCol] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the slice containing coding block colCb in the collocated picture specified by ColPic, and the following applies:

$$colPocDiff = DiffPicOrderCnt(ColPic, refPicListCol[refIdxCol]) \quad (8\text{-}229)$$

$$currPocDiff = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) \quad (8\text{-}230)$$

  – If RefPicListX[refIdxLX] is a long-term reference picture, or colPocDiff is equal to currPocDiff, mvLXCol is derived as follows:

$$mvLXCol = mvCol \quad (8\text{-}231)$$

  – Otherwise, mvLXCol is derived as a scaled version of the motion vector mvCol as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}232)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}233)$$

$$mvLXCol = Clip3(-32768, 32767, Sign(distScaleFactor * mvCol) * ((Abs(distScaleFactor * mvCol) + 127) >> 8)) \quad (8\text{-}234)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, colPocDiff) \quad (8\text{-}235)$$

$$tb = Clip3(-128, 127, currPocDiff) \quad (8\text{-}236)$$

8.3.2.13 Derivation process for chroma motion vectors

Input to this process is a luma motion vector in 1/16 fractional-sample accuracy mvLX.

Output of this process is a chroma motion vector in 1/32 fractional-sample accuracy mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvCLX, the following applies:

$$mvCLX[0] = mvLX[0] * 2 / SubWidthC \quad (8\text{-}237)$$

$$mvCLX[1] = mvLX[1] * 2 / SubHeightC \quad (8\text{-}238)$$

8.3.2.14 Rounding process for motion vectors

Inputs to this process are

– the motion vector mvX,

– the right shift parameter rightShift for rounding,

– the left shift parameter leftShift for resolution increase.

Output of this process is the rounded motion vector mvX.

For the rounding of mvX, the following applies:

$$offset = 1 << (rightShift - 1) \quad (8\text{-}239)$$

$$mvX[0] = (mvX[0] >= 0 ? (mvX[0] + offset) >> rightShift : -((- mvX[0] + offset) >>rightShift)) << leftShift \quad (8\text{-}240)$$

$$mvX[1] = (mvX[1] >= 0 ? (mvX[1] + offset) >> rightShift : -((- mvX[1] + offset) >> rightShift)) << leftShift \quad (8\text{-}241)$$

8.3.3 Derivation process for affine motion vector components and reference indices

8.3.3.1 General

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the luma subblock motion vector array in 1/16 fractional-sample accuracy mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1,

– the chroma subblock motion vector array in 1/32 fractional-sample accuracy mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1.

For the derivation of the variables mvLX[xSbIdx][ySbIdx] and mvCLC[xSbIdx][ySbIdx], the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1, the following applies:

– For the derivation of the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, refIdxL0, refIdxL1, predFlagL0 and predFlagL1, the following applies:

  – If merge_affine_flag[xCb][yCb] is equal to 1, the derivation process for affine control point motion vectors and reference indices in affine merge mode as specified in 8.3.3.2 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx], cpMvL1[cpIdx], with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as output.

  – Otherwise (merge_affine_flag[xCb][yCb] is equal to 0), for X being replaced by either 0 or 1 in the variables predFlagLX, cpMvLX, MvdCpLX, and refIdxLX, in PRED_LX, and in the syntax element ref_idx_lX, the following ordered steps apply:

    1. The number of control point motion vectors numCpMv is set equal to MotionModelIdc[xCb][yCb] + 1.

    2. The variables refIdxLX and predFlagLX with X being 0 or 1 are derived as follows:

       - If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

         $$refIdxLX = ref\_idx\_lX[xCb][yCb] \quad (8\text{-}242)$$

         $$predFlagLX = 1 \quad (8\text{-}243)$$

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

         $$refIdxLX = -1 \quad (8\text{-}244)$$

         $$predFlagLX = 0 \quad (8\text{-}245)$$

    3. The variable mvdCpLX[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, is derived as follows:

mvdCpLX[cpIdx][0] = MvdCpLX[xCb][yCb][cpIdx][0] (8-246)

mvdCpLX[cpIdx][1] = MvdCpLX[xCb][yCb][cpIdx][1] (8-247)

4. When predFlagLX is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in clause 8.3.3.4 is invoked with the luma coding block location (xCb, yCb), and the variables cbWidth, cbHeight, refIdxLX, and the number of control point motion vectors numCpMv as inputs, and the output being mvpCpLX[cpIdx] with cpIdx ranging from 0 to numCpMv − 1.

5. When predFlagLX is equal to 1, the luma motion vectors cpMvLX[cpIdx] with cpIdx ranging from 0 to NumCpMv − 1, are derived as follows:

uLX[cpIdx][0] = (mvpCpLX[cpIdx][0] + mvdCpLX[cpIdx][0] + 2^{18}) % 2^{18} (8-248)

cpMvLX[cpIdx][0] = (uLX[cpIdx][0] >= 2^{17}) ? (uLX[cpIdx][0] − 2^{18}) : uLX[cpIdx][0] (8-249)

uLX[cpIdx][1] = (mvpCpLX[cpIdx][1] + mvdCpLX[cpIdx][1] + 2^{18}) % 2^{18} (8-250)

cpMvLX[cpIdx][1] = (uLX[cpIdx][1] >= 2^{17}) ? (uLX[cpIdx][1] − 2^{18}) : uLX[cpIdx][1] (8-251)

– The derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.3.3.7 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma prediction block height cbHeight, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL1[cpIdx], cpMvL0[cpIdx] with cpIdx being 0..2, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, the luma motion vector array mvLX[xSbIdx][ySbIdx] and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1 as output.

– For xSbIdx = 0..(cbWidth >> 2) − 1 and ySbIdx = 0..(cbHeight >> 2) − 1, the motion vectors MvLX with X being 0 and 1 are derived as follows:

  – The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

    (xSb, ySb) = (xCb + xSbIdx * 4, yCb + ySbIdx * 4) (8-252)

  – If xSbIdx and ySbIdx are both equal to 0 (top-left subblock), the following applies for x = 0..3 and y = 0..3:

    MvL0[xSb + x][ySb + y] = cpMvL0[0] (8-253)

    MvL1[xSb + x][ySb + y] = cpMvL1[0] (8-254)

  – Otherwise, if xSbIdx is equal to (cbWidth >> 2) − 1 and ySbIdx is equal to 0 (top-right subblock), the following applies for x = 0..3 and y = 0..3:

    MvL0[xSb + x][ySb + y] = cpMvL0[1] (8-255)

    MvL1[xSb + x][ySb + y] = cpMvL1[1] (8-256)

  – Otherwise, if numCpMv is equal to 3 and xSbIdx is equal to 0 and ySbIdx is equal to (cbHeight >> 2) − 1 (belowleft subblock), the following applies for x = 0..3 and y = 0..3:

    MvL0[xSb + x][ySb + y] = cpMvL0[2] (8-257)

    MvL1[xSb + x][ySb + y] = cpMvL1[2] (8-258)

  – Otherwise, the following applies for x = 0..3 and y = 0..3:

    MvL0[xSb + x][ySb + y] = mvL0[xSbIdx][ySbIdx] (8-259)

    MvL1[xSb + x][ySb + y] = mvL1[xSbIdx][ySbIdx] (8-260)

– For x = 0..cbWidth − 1 and y = 0..cbHeight − 1, the reference indices refIdxLX and the prediction list utilization flags PredFlagLX with X being 0 and 1 are derived as follows:

RefIdxL0[xCb + x][yCb + y] = refIdxL0 (8-261)

RefIdxL1[xCb + x][yCb + y] = refIdxL1 (8-262)

PredFlagL0[xCb + x][yCb + y] = predFlagL0 (8-263)

PredFlagL1[xCb + x][yCb + y] = predFlagL1 (8-264)

8.3.3.2 Derivation process for luma affine control point motion vectors and reference indices in affine merge mode

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

– the number of control point motion vectors numCpMv,

– the luma affine control point motion vectors cpMvL0[cpIdx], cpMvL1[cpIdx], with cpIdx ranging from 0 to numCpMv − 1

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

The derivation process for affine motion information in the neighbouring blocks of the current luma coding block as specified in 8.3.3.5 is invoked with the luma location (xCb, yCb), the variables cbWidth and cbHeight as inputs, the output being availablity flags $availableFlag_k$, luma locations of neighbouring blocks (xNb_k, yNb_k), the neighbouring luma coding block width nbW_k, and the neighbouring luma coding block height nbH_k with k ranging from 0 to 4.

The the number of control point motion vectors numCpMv, luma affine control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 are derived as follows:

– The variable availableFlag is set equal to 0.

– For (xNb_k, yNb_k) with k ranging from 0 to 4, the following applies:

  – When availableFlag is equal to 0 and $availableFlag_k$ is equal to 1 the following applies:

    – The reference indices refIdxLX and the prediction list utilization flags predFlagLX with X being 0 and 1, and numCpMv are derived as follows

      refIdxLX = RefIdxLX[xNb_k][yNb_k] (8-265)

      predFlagLX = PredFlagLX[xNb_k][yNb_k] (8-266)

      numCpMv = MotionModelIdc[xNb_k][yNb_k] + 1 (8-267)

    – The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

      MotionModelIdc[x][y] = MotionModelIdc[xNb_k][yNb_k] (8-268)

    – The derivation process for luma affine control point motion vectors from a neighbouring block in as specified clause 8.3.3.3 is invoked with the coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNb, yNb) set equal to the neighbouring luma coding block location (xNb_k, yNb_k), the variables nNbW and nNbH set equal to the neighbouring luma coding block width nbW_k and the neighbouring luma coding block height nbH_k, and the number of control point motion vectors numCpMv as inputs, and the luma affine control point motion vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 as output.

    – The variable availableFlag is set equal to 1.

8.3.3.3 Derivation process for luma affine control point motion vectors from a neighbouring block

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,

– a luma location (xNb, yNb) specifying the top-left sample of the neighbouring luma coding block relative to the top-left luma sample of the current picture,

– two variables nNbW and nNbH specifying the width and the height of the neighbouring luma coding block,

– the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1.

The variables log2NbW and log2NbH are derived as follows:

log2NbW = Log2(nNbW) (8-269)

log2NbH = Log2(nNbH) (8-270)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

mvScaleHor = MvLX[xNb][yNb][0] << 7 (8-271)

mvScaleVer = MvLX[xNb][yNb][1] << 7 (8-272)

dHorX = (MvLX[xNb + nNbW − 1][yNb][0] − MvLX[xNb][yNb][0]) << (7 − log2NbW) (8-273)

dVerX = (MvLX[xNb + nNbW − 1][yNb][1] − MvLX[xNb][yNb][1]) << (7 − log2NbW) (8-274)

The variables dHorY and dVerY are derived as follows:

– If MotionModelIdc[xNb][yNb] is equal to 2, the following applies:

dHorY = (MvLX[xNb][yNb + nNbH − 1][0] − MvLX[xNb][yNb][0]) << (7 − log2NbH) (8-275)

dVerY = (MvLX[xNb][yNb + nNbH − 1][1] − MvLX[xNb][yNb][1]) << (7 − log2NbH) (8-276)

– Ohterwise (MotionModelIdc[xNb][yNb] is equal to 1), the following apply,

dHorY = − dVerX (8-277)

dVerY = dHorX (8-278)

The luma affine control point motion vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 are derived as follows:

– The first two control point motion vectors cpMvLX[0] and cpMvLX[1] are derived as follows:

cpMvLX[0][0] = (mvScaleHor + dHorX * (xCb − xNb) + dHorY * (yCb − yNb)) (8-279)

cpMvLX[0][1] = (mvScaleVer + dVerX * (xCb − xNb) + dVerY * (yCb − yNb)) 8-280)

cpMvLX[1][0] = (mvScaleHor + dHorX * (xCb + cbWidth − xNb) + dHorY * (yCb − yNb)) (8-281)

cpMvLX[1][1] = (mvScaleVer + dVerX * (xCb + cbWidth − xNb) + dVerY * (yCb − yNb)) (8-282)

– If numCpMv is equal to 3, the third control point vector cpMvLX[2] is derived as follows:

cpMvLX[2][0] = (mvScaleHor + dHorX * (xCb − xNb) + dHorY * (yCb + cbHeight − yNb)) (8-283)

cpMvLX[2][1] = (mvScaleVer + dVerX * (xCb − xNb) + dVerY * (yCb + cbHeight − yNb)) (8-284)

– The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[cpIdx], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[cpIdx] as output, with X being 0 or 1 and cpIdx = 0 .. numCpMv − 1.

8.3.3.4 Derivation process for luma affine control point motion vector predictors

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,

– the reference index of the current coding unit refIdxLX, with X being 0 or 1,

– the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point motion vector predictors mvpCpLX[cpIdx] with X being 0 or 1, and cpIdx = 0 .. numCpMv − 1.

The derivation process for the partitioning information of neighbouring blocks in clause 8.3.3.5 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the output being the availability flags $availableFlag_k$, the luma locations of neighbouring blocks (xNb_k, yNb_k) and the widths nbW_k and the heights nbH_k of the neighbouring coding blocks with k = 0..4.

For the derivation of the control point motion vectors predictor candidate list, cpMvpListLX with X being 0 or 1, the following ordered steps apply:

1. The number of control point motion vector predictor candidates in the list numCpMvpCandLX is set equal to 0.
2. The variable availableFlagLX is set equal to 0.
3. For k ranging from 0 to 4, inclusive, the following applies:
   – When $availableFlag_k$ is equal to 1 and numCpMvpCandLX is less than 2, the following applies:
     – When PredFlagLX[xNb_k][yNb_k] is equal to 1 and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNb_k][yNb_k]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX is set equal to 1, and the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.3.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (xNb_k, yNb_k), the neighbouring luma coding block width and height ($nNbW_k$, $nNbH_k$), and the number of control point motion vectors numCpMv as inputs, the control point motion vector predictor candidates cpMvpLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 as output.
     – When availableFlagLX is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvpLX[cpIdx], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvpLX[cpIdx] as output, with cpIdx = 0 .. numCpMv − 1.
     – If availableFlagLX is equal to 1 and numCpMvpCandLX is equal to 0, the following assignments are made:

       cpMvpListLX[0][0] = cpMvpLX[0] (8-285)

       cpMvpListLX[0][1] = cpMvpLX[1] (8-286)

       cpMvpListLX[0][2] = cpMvpLX[2] (8-287)

       numCpMvpCandLX = 1 (8-288)

     – Otherwise, if availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,
       – numCpMv is equal to 3 and one of the following conditions is true:
         – cpMvpListLX[0][0] is not equal to cpMvpLX[0]
         – cpMvpListLX[0][1] is not equal to cpMvpLX[1]
         – cpMvpListLX[0][2] is not equal to cpMvpLX[2]
       – numCpMv is equal to 2 and one of the following conditions is true:
         – cpMvpListLX[0][0] is not equal to cpMvpLX[0]
         – cpMvpListLX[0][1] is not equal to cpMvpLX[1]

       the following assignments are made:

       cpMvpListLX[1][0] = cpMvpLX[0] (8-289)

       cpMvpListLX[1][1] = cpMvpLX[1] (8-290)

       cpMvpListLX[1][2] = cpMvpLX[2] (8-291)

       numCpMvpCandLX = 2 (8-292)

     – The variable availableFlagLX is set equal to 0.
4. For k ranging from 0 to 4, inclusive, the following applies:
   – When $availableFlag_k$ is equal to 1 and numCpMvpCandLX is less than 2, the following applies:

– When PredFlagLY[xNb_k][yNb_k] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNb_k][yNb_k]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX is set equal to 1, and the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.3.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (xNb_k, yNb_k), the neighbouring luma coding block width and height ($nNbW_k$, $nNbH_k$), and the number of control point motion vectors numCpMv as inputs, the control point motion vector predictor candidates cpMvpLY[cpIdx] with cpIdx = 0 .. numCpMv − 1 as output.

– When availableFlagLX is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvpLY[cpIdx], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvpLY[cpIdx] as output, with cpIdx = 0 .. numCpMv − 1.

– If availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[0][0] = cpMvpLY[0] (8-293)

cpMvpListLX[0][1] = cpMvpLY[1] (8-294)

cpMvpListLX[0][2] = cpMvpLY[2] (8-295)

numCpMvpCandLX = 1 (8-296)

– Otherwise, if availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,

  – numCpMv is equal to 3 and one of the following conditions is true:

    – cpMvpListLX[0][0] is not equal to cpMvpLY[0]

    – cpMvpListLX[0][1] is not equal to cpMvpLY[1]

    – cpMvpListLX[0][2] is not equal to cpMvpLY[2]

  – numCpMv is equal to 2 and one of the following conditions is true:

    – cpMvpListLX[0][0] is not equal to cpMvpLY[0]

    – cpMvpListLX[0][1] is not equal to cpMvpLY[1]

  the following assignments are made:

cpMvpListLX[1][0] = cpMvpLY[0] (8-297)

cpMvpListLX[1][1] = cpMvpLY[1] (8-298)

cpMvpListLX[1][2] = cpMvpLY[2] (8-299)

numCpMvpCandLX = 2 (8-300)

– The variable availableFlagLX is set equal to 0.

5. When numCpMvpCandLX is less than 2, the following applies

– The derivation process for constructed affine control point motion vector prediction candidate as specified in clause 8.3.3.6 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX, with X being 0 or 1 as inputs, and the availability flag availableConsFlagLX and cpMvpLX[cpIdx] with X being 0 or 1, cpIdx = 0..numCpMv − 1 as outputs.

– If availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[0][0] = cpMvpLX[0] (8-301)

cpMvpListLX[0][1] = cpMvpLX[1] (8-302)

cpMvpListLX[0][2] = cpMvpLX[2] (8-303)

numCpMvpCandLX = 1 (8-304)

– Otherwise, if availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,

  – numCpMv is equal to 3 and one of the following conditions is true:

    – cpMvpListLX[0][0] is not equal to cpMvpLX[0]

– cpMvpListLX[0][1] is not equal to cpMvpLX[1]

– cpMvpListLX[0][2] is not equal to cpMvpLX[2]

– numCpMv is equal to 2 and one of the following conditions is true:

– cpMvpListLX[0][0] is not equal to cpMvpLY[0]

– cpMvpListLX[0][1] is not equal to cpMvpLY[1]

the following assignments are made:

cpMvpListLX[1][0] = cpMvpLX[0] (8-305)

cpMvpListLX[1][1] = cpMvpLX[1] (8-306)

cpMvpListLX[1][2] = cpMvpLX[2] (8-307)

numCpMvpCandLX = 2 (8-308)

6. When numCpMvpCandLX is less than 2, the derivation process for luma motion vector predictor candidate list as specified in clause 8.3.2.9 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1 as output. The cpMvpListLX is appended with candidates from mvpListLX as follows:

```
i = 0
while ( numCpMvpCandLX < 2 ) {
    cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvpListLX[ i ]
    cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvpListLX[ i ]
    cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvpListLX[ i ]
    numCpMvpCandLX++
    i++
}
```
(8-309)

The affine control point motion vector predictor cpMvpLX with X being 0 is derived as follows:

cpMvpLX = cpMvpListLX[mvp_lX_flag[xCb][yCb]] (8-310)

8.3.3.5 Derivation process for neighbouring affine blocks partition information

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

– the availability flags $availableFlag_k$ with k = 0..4,

– the luma locations (xNb_k, yNb_k) of the top-left sample of the neighbouring coding blocks relative to the top-left luma sample of the current picture with k = 0..4,

– the widths nbW_k and the heights nbH_k of the neighbouring coding blocks with k = 0..4.

The neighbouring luma locations (xN_k, yN_k) with k = 0..4 inside the neighbouring luma coding block are set as follows:

(xN_0, yN_0) = (xCb − 1, yCb + cbHeight − 1) (8-311)

(xN_1, yN_1) = (xCb + cbWidth − 1, yCb − 1) (8-312)

(xN_2, yN_2) = (xCb + cbWidth − 1, yCb) (8-313)

(xN_3, yN_3) = (xCb − 1, yCb + cbHeight) (8-314)

(xN_4, yN_4) = (xCb − 1, yCb − 1) (8-315)

The variables $availableFlag_k$, nbW_k and nbH_k with k = 0..4 are derived as follows:

– The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block

width cbWidth, the luma coding block height cbHeight, and the luma location (xNb_k, yNb_k) as inputs, and the output is assigned to the coding block availability flag $availableFlag_k$.

– When $availableFlag_k$ is equal to 1, the following applies:

  – When MotionModelIdc[xNb_k][yNb_k] is equal to 0, $availableFlag_k$ is set equal 0.

  – When $availableFlag_k$ is equal to 1, xNb_k, yNb_k, nbW_k and nbH_k are set equal to CbPosX[xN_k][yN_k], CbPosY[xN_k][yN_k], CbWidth[xN_k][yN_k] and CbHeight[xN_k][yN_k], respectively.

8.3.3.6 Derivation process for constructed affine control point motion vector prediction candidates

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,

– the reference index of the current prediction unit partition refIdxLX, with X being 0 or 1,

Output of this process are:

– the availability flag of the constructed affine control point motion vector prediction candidiates availableConsFlagLX with X being 0 or 1,

– the constructed affine control point motion vector prediction candidiates cpMvLX[cpIdx] with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLX[0] and the availability flag availableFlagLX[0] are derived in the following ordered steps:

1. The sample locations ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$) and ($xNbA_2$, $yNbA_2$) are set equal to (xCb − 1, yCb − 1), (xCb , yCb − 1) and (xCb − 1, yCb), respectively.
2. The availability flag availableFlagLX[0] is set equal to 0 and both components of cpMvLX[0] are set equal to 0.
3. The following applies for (xNbTL, yNbTL) withTL being replaced by B_2, B_3, and A_2:
   – The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNbY, yNbY) set equal to (xNbTL, yNbTL) as inputs, and the output is assigned to the coding block availability flag availableTL.
   – When availableTL is equal to TRUE and availableFlagLX[0] is equal to 0, the following applies:
     – If PredFlagLX[xNbTL][yNbTL] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNbTL][yNbTL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[0] is set equal to 1 and the following assignments are made:

$$cpMvLX[0] = MvLX[xNbTL][yNbTL] \quad (8\text{-}316)$$

     – Otherwise, when PredFlagLY[xNbTL][yNbTL] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNbTL][yNbTL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[0] is set equal to 1 and the following assignments are made:

$$cpMvLX[0] = MvLY[xNbTL][yNbTL] \quad (8\text{-}317)$$

     – When availableFlagLX[0] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[0], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[0] as output.

The second (top-right) control point motion vector cpMvLX[1] and the availability flag availableFlagLX[1] are derived in the following ordered steps:

1. The sample locations ($xNbB_1$, $yNbB_1$) and ($xNbB_0$, $yNbB_0$) are set equal to (xCb + cbWidth − 1, yCb − 1) and (xCb + cbWidth, yCb − 1), respectively.
2. The availability flag availableFlagLX[1] is set equal to 0 and both components of cpMvLX[1] are set equal to 0.
3. The following applies for (xNbTR, yNbTR) withTR being replaced by B_1 and B_0:

– The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNbY, yNbY) set equal to (xNbTR, yNbTR) as inputs, and the output is assigned to the coding block availability flag availableTR.

– When availableTR is equal to TRUE and availableFlagLX[0] is equal to 0, the following applies:

  – If PredFlagLX[xNbTR][yNbTR] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNbTR][yNbTR]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[1] is set equal to 1 and the following assignments are made:

    cpMvLX[1] = MvLX[xNbTR][yNbTR] (8-318)

  – Otherwise, when PredFlagLY[xNbTR][yNbTR] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNbTR][yNbTR]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[1] is set equal to 1 and the following assignments are made:

    cpMvLX[1] = MvLY[xNbTR][yNbTR] (8-319)

  – When availableFlagLX[1] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[1], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[1] as output.

The third (bottom-left) control point motion vector cpMvLX[2] and the availability flag availableFlagLX[2] are derived in the following ordered steps:

1. The sample locations ($xNbA_1$, $yNbA_1$) and ($xNbA_0$, $yNbA_0$) are set equal to (xCb − 1, yCb + cbHeight − 1) and (xCb − 1, yCb + cbHeight), respectively.
2. The availability flag availableFlagLX[2] is set equal to 0 and both components of cpMvLX[2] are set equal to 0.
3. The following applies for (xNbBL, yNbBL) with BL being replaced by A_1 and A_0:

   – The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNbY, yNbY) set equal to (xNbBL, yNbBL) as inputs, and the output is assigned to the coding block availability flag availableBL.

   – When availableBL is equal to TRUE and availableFlagLX[2] is equal to 0, the following applies:

     – If PredFlagLX[xNbBL][yNbBL] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNbBL][yNbBL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[2] is set equal to 1 and the following assignments are made:

       cpMvLX[2] = MvLX[xNbBL][yNbBL] (8-320)

     – Otherwise, when PredFlagLY[xNbBL][yNbBL] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNbBL][yNbBL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[2] is set equal to 1 and the following assignments are made:

       cpMvLX[2] = MvLY[xNbBL][yNbBL] (8-321)

     – When availableFlagLX[2] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[2], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[2] as output.

The variables availableConsFlagLX, cpMvLX[cpIdx] are derived as follows:

– The variables log2CbW and log2CbH are derived as follows:

  log2CbW = Log2(cbWidth) (8-322)

  log2CbH = Log2(cbHeight) (8-323)

– If availableFlagLX[0] is equal to 1 and availableFlagLX[1] is equal to 1 and availableFlagLX[2] is equal to 1, availableConsFlagLX is set equal to 1

– Otherwise, if availableFlagLX[0] is equal to 1, and availableFlagLX[1] is equal to 1, and MotionModelIdc[xCb][yCb] is equal to 2, availableConsFlagLX is set equal to 1 and the following applies:

  – The third control point motion vector cpMvLX[2] is derived as follows:

    cpMvLX[2][0] = (cpMvLX[0][0] << 7) − (8-324)
    ((cpMvLX[1][1] − cpMvLX[0][1]) << (7 + log2CbH − log2CbW))

$$cpMvLX[\ 2\ ][\ 1\ ] = (\ cpMvLX[\ 0\ ][\ 1\ ] << 7\ ) + (\ (cpMvLX[\ 1\ ][\ 0\ ] - cpMvLX[\ 0\ ][\ 0\ ]) << (\ 7 + log2CbH - log2CbW\ )\ ) \quad (8\text{-}325)$$

- The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[2], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[2] as output, with X being 0 or 1.
- The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[2], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[2] as output, with X being 0 or 1.

– Otherwise, if availableFlagLX[0] is equal to 1 and availableFlagLX[2] is equal to 1, availableConsFlagLX is set equal to 1, and the following applies:

- The second control point motion vector cpMvLX[1] is derived as follow:

$$cpMvLX[\ 1\ ][\ 0\ ] = (\ cpMvLX[\ 0\ ][\ 0\ ] << 7\ ) + (\ (\ cpMvLX[\ 2\ ][\ 1\ ] - cpMvLX[\ 0\ ][\ 1\ ]\ ) << (\ 7 + log2CbH - log2CbW\ )\ ) \quad (8\text{-}326)$$

$$cpMvLX[\ 1\ ][\ 1\ ] = (\ cpMvLX[\ 0\ ][\ 1\ ] << 7\ ) - (\ (cpMvLX[\ 2\ ][\ 0\ ] - cpMvLX[\ 0\ ][\ 0\ ]) << (\ 7 + log2CbH - log2CbW\ )\ ) \quad (8\text{-}327)$$

- The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[1], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[1] as output, with X being 0 or 1.
- The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[1], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[1] as output, with X being 0 or 1.

– Otherwise, availableConsFlagLX is set equal to 0.

8.3.3.7 Derivation process for motion vector arrays from affine control point motion vectors

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block,

– the number of control point motion vectors numCpMv,

– the control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx], with cpIdx = 0..numCpMv − 1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are:

– the luma subblock motion vector array mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1,

– the chroma subblock motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1.

When predFlagLX is equal to 1, the following applies for X being 0 and 1:

– The variables log2CbW and log2CbH are derived as follows:

$$log2CbW = Log2(\ cbWidth\ ) \quad (8\text{-}328)$$

$$log2CbH = Log2(\ cbHeight\ ) \quad (8\text{-}329)$$

– The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

$$mvScaleHor = cpMvLX[\ 0\ ][\ 0\ ] << 7 \quad (8\text{-}330)$$

$$mvScaleVer = cpMvLX[\ 0\ ][\ 1\ ] << 7 \quad (8\text{-}331)$$

$$dHorX = (\ cpMvLX[\ 1\ ][\ 0\ ] - cpMvLX[\ 0\ ][\ 0\ ]\ ) << (\ 7 - log2CbW\ ) \quad (8\text{-}332)$$

$$dVerX = (\ cpMvLX[\ 1\ ][\ 1\ ] - cpMvLX[\ 0\ ][\ 1\ ]\ ) << (\ 7 - log2CbW\ ) \quad (8\text{-}333)$$

– If numCpMv is equal to 3, the variable dHorY and dVerY are derived as follow:

dHorY = (cpMvLX[2][0] − cpMvLX[0][0]) << (7 − log2CbH) (8-334)

dVerY = (cpMvLX[2][1] − cpMvLX[0][1]) << (7 − log2CbH) (8-335)

– Otherwise (numCpMv is equal to 2), the variable dHorY and dVerY are derived as follows:

dHorY = − dVerX (8-336)

dVerY = dHorX (8-337)

– For xSbIdx = 0..(cbWidth >> 2) − 1 and ySbIdx = 0..(cbHeight >> 2) − 1, the following applies:
  – The luma motion vector mvLX[xSbIdx][ySbIdx] is derived as follows :

xPosCb = 2 + (xSbIdx << 2) (8-338)

yPosCb = 2 + (ySbIdx << 2) (8-339)

mvLX[xSbIdx][ySbIdx][0] = (mvScaleHor + dHorX * xPosCb + dHorY * yPosCb) (8-340)

mvLX[xSbIdx][ySbIdx][1] = (mvScaleVer + dVerX * xPosCb + dVerY * yPosCb) (8-341)

  – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLX[xSbIdx][ySbIdx], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded mvLX[xSbIdx][ySbIdx] as output.
  – The derivation process for chroma motion vectors in clause 8.3.2.13 is invoked with mvLX[xSbIdx][ySbIdx] as input, and the chroma motion vector mvCLX[xSbIdx][ySbIdx] as output.

8.3.4 Decoding process for inter blocks

8.3.4.1 General

This process is invoked when decoding coding unit whose CuPredMode[xCb][yCb] is not equal to MODE_INTRA.

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,

– the chroma motion vectors mvCL0 and mvCL1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are:

– an (cbWidth)x(cbHeight) array $predSamples_L$ of luma prediction samples,

– an (cbWidth / 2)x(cbHeight / 2) array $predSamples_{Cb}$ of chroma prediction samples for the component Cb,

– an (cbWidth / 2)x(cbHeight / 2) array $predSamples_{Cr}$ of chroma prediction samples for the component Cr.

Let $predSamplesL0_L$ and $predSamplesL1_L$ be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, $predSamplesL0_{Cb}$, $predSamplesL1_{Cb}$, $predSamplesL0_{Cr}$ and $predSamplesL1_{Cr}$ be (cbWidth / 2)x(cbHeight / 2) arrays of predicted chroma sample values.

The width and the height of the current luma coding sublock subCbWidth, subCbHeight are derived as follows:

sbWidth = cbWidth / numSbX (8-342)

sbHeight = cbHeight / numSbY (8-343)

For each coding subblock at subblock index (xSbIdx, ySbIdx) with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1, the following applies:

– The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

$$(xSb, ySb) = (xCb + xSbIdx * sbWidth, yCb + ySbIdx * sbHeight) \quad (8\text{-}344)$$

– For X being each of 0 and 1, when predFlagLX is equal to 1, the following applies:

  – The reference picture consisting of an ordered two-dimensional array $refPicLX_L$ of luma samples and two ordered two-dimensional arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$ of chroma samples is derived by invoking the process specified in clause 8.3.4.2 with refIdxLX as input.

  – The array $predSamplesLX_L$ and the arrays $predSamplesLX_{Cb}$ and $predSamplesLX_{Cr}$ are derived by invoking the fractional sample interpolation process specified in clause 8.3.4.3 with the luma location (xCb, yCb), the luma coding subblock width sbWidth, the luma coding subblock height sbHeight, the motion vectors mvLX[xSb][ySb], mvCLX[xSb][ySb], and the reference arrays $refPicLX_L$, $refPicLX_{Cb}$, and $refPicLX_{Cr}$ as inputs.

– The prediction samples inside the current luma coding subblock, $predSamples_L[x_L + xSb][y_L + ySb]$ with $x_L = 0..sbWidth - 1$ and $y_L = 0..sbHeight - 1$, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays $predSamplesL0_L$ and $predSamplesL1_L$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx set equal to 0 as inputs.

– The prediction samples inside the current chroma component Cb coding block, $predSamples_{Cb}[x_C + xCb / 2][y_C + yCb / 2]$ with $x_C = 0..cbWidth / 2 - 1$ and $y_C = 0..cbHeight / 2 - 1$, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays $predSamplesL0_{Cb}$ and $predSamplesL1_{Cb}$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx set equal to 1 as inputs.

– The prediction samples inside the current chroma component Cr coding block, $predSamples_{Cr}[x_C + xCb / 2][y_C + yCb / 2]$ with $x_C = 0..cbWidth / 2 - 1$ and $y_C = 0..cbHeight / 2 - 1$, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays $predSamplesL0_{Cr}$ and $predSamplesL1_{Cr}$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx set equal to 2 as inputs.

– When MotionModelIdc[xCb][yCb] is equal to 0, the following assignments are made for x = 0..sbWidth − 1 and y = 0..sbHeight − 1:

$$MvL0[xSb + x][ySb + y] = mvL0[xSbIdx][ySbIdx] \quad (8\text{-}345)$$

$$MvL1[xSb + x][ySb + y] = mvL1[xSbIdx][ySbIdx] \quad (8\text{-}346)$$

$$RefIdxL0[xSb + x][ySb + y] = refIdxL0 \quad (8\text{-}347)$$

$$RefIdxL1[xSb + x][ySb + y] = refIdxL1 \quad (8\text{-}348)$$

$$PredFlagL0[xSb + x][ySb + y] = predFlagL0 \quad (8\text{-}349)$$

$$PredFlagL1[xSb + x][ySb + y] = predFlagL1 \quad (8\text{-}350)$$

8.3.4.2 Reference picture selection process

Input to this process is a reference index refIdxLX.

Output of this process is a reference picture consisting of a two-dimensional array of luma samples $refPicLX_L$ and two two-dimensional arrays of chroma samples $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

The output reference picture RefPicListX[refIdxLX] consists of a pic_width_in_luma_samples by pic_height_in_luma_samples array of luma samples $refPicLX_L$ and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

The reference picture sample arrays $refPicLX_L$, $refPicLX_{Cb}$ and $refPicLX_{Cr}$ correspond to decoded sample arrays S_L, S_{Cb} and S_{Cr} derived in clause 8.5 for a previously-decoded picture.

8.3.4.3 Fractional sample interpolation process

8.3.4.3.1 General

Inputs to this process are:

– a luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture,

– a variable sbWidth specifying the width of the current coding subblock in luma samples,

– a variable sbHeight specifying the height of the current coding subblock in luma samples,

– a luma motion vector mvLX given in 1/16-luma-sample units,

– a chroma motion vector mvCLX given in 1/32-chroma-sample units,

– the selected reference picture sample array $refPicLX_L$ and the arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

Outputs of this process are:

– an (sbWidth)x(sbHeight) array $predSamplesLX_L$ of prediction luma sample values,

– two (sbWidth / 2)x(sbHeight / 2) arrays $predSamplesLX_{Cb}$ and $predSamplesLX_{Cr}$ of prediction chroma sample values.

Let ($xInt_L$, $yInt_L$) be a luma location given in full-sample units and ($xFrac_L$, $yFrac_L$) be an offset given in 1/16-sample units. These variables are used only in this clause for specifying fractional-sample locations inside the reference sample arrays $refPicLX_L$, $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

For each luma sample location ($x_L = 0..sbWidth - 1$, $y_L = 0..sbHeight - 1$) inside the prediction luma sample array $predSamplesLX_L$, the corresponding prediction luma sample value $predSamplesLX_L[x_L][y_L]$ is derived as follows:

– The variables $xInt_L$, $yInt_L$, $xFrac_L$ and $yFrac_L$ are derived as follows:

$$xInt_L = xSb + (mvLX[0] >> 4) + x_L \quad (8\text{-}351)$$

$$yInt_L = ySb + (mvLX[1] >> 4) + y_L \quad (8\text{-}352)$$

$$xFrac_L = mvLX[0] \& 15 \quad (8\text{-}353)$$

$$yFrac_L = mvLX[1] \& 15 \quad (8\text{-}354)$$

– The prediction luma sample value $predSamplesLX_L[x_L][y_L]$ is derived by invoking the process specified in clause 8.3.4.3.2 with ($xInt_L$, $yInt_L$), ($xFrac_L$, $yFrac_L$) and $refPicLX_L$ as inputs.

Let ($xInt_C$, $yInt_C$) be a chroma location given in full-sample units and ($xFrac_C$, $yFrac_C$) be an offset given in 1/32 sample units. These variables are used only in this clause for specifying general fractional-sample locations inside the reference sample arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

For each chroma sample location ($x_C = 0..sbWidth / 2 - 1$, $y_C = 0..sbHeight / 2 - 1$) inside the prediction chroma sample arrays $predSamplesLX_{Cb}$ and $predSamplesLX_{Cr}$, the corresponding prediction chroma sample values $predSamplesLX_{Cb}[x_C][y_C]$ and $predSamplesLX_{Cr}[x_C][y_C]$ are derived as follows:

– The variables $xInt_C$, $yInt_C$, $xFrac_C$ and $yFrac_C$ are derived as follows:

$$xInt_C = (xSb / SubWidthC) + (mvCLX[0] >> 5) + x_C \quad (8\text{-}355)$$

$$yInt_C = (ySb / SubHeightC) + (mvCLX[1] >> 5) + y_C \quad (8\text{-}356)$$

$$xFrac_C = mvCLX[0] \& 31 \quad (8\text{-}357)$$

$$yFrac_C = mvCLX[1] \& 31 \quad (8\text{-}358)$$

– The prediction sample value $predSamplesLX_{Cb}[x_C][y_C]$ is derived by invoking the process specified in clause 8.3.4.3.3 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cb}$ as inputs.

– The prediction sample value $predSamplesLX_{Cr}[x_C][y_C]$ is derived by invoking the process specified in clause 8.3.4.3.3 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cr}$ as inputs.

8.3.4.3.2 Luma sample interpolation process

Inputs to this process are:

– a luma location in full-sample units ($xInt_L$, $yInt_L$),

– a luma location in fractional-sample units ($xFrac_L$, $yFrac_L$),

– the luma reference sample array $refPicLX_L$.

Output of this process is a predicted luma sample value $predSampleLX_L$

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min(4, $BitDepth_Y$ − 8), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max(2, 14 − $BitDepth_Y$).

– The variable picW is set equal to pic_width_in_luma_samples and the variable picH is set equal to pic_height_in_luma_samples.

The luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p equal to $xFrac_L$ or $yFrac_L$ are specified in Table 8-7.

The luma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..7:

$$xInt_i = Clip3(0, picW - 1, xInt_L + i - 3) \quad (8\text{-}359)$$

$$yInt_i = Clip3(0, picH - 1, yInt_L + i - 3) \quad (8\text{-}360)$$

The predicted luma sample value $predSampleLX_L$ is derived as follows:

– If both $xFrac_L$and $yFrac_L$ are equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = refPicLX_L[xInt_3][yInt_3] << shift3 \quad (8\text{-}361)$$

– Otherwise, if $xFrac_L$ is not equal to 0 and $yFrac_L$ is equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_3]) >> shift1 \quad (8\text{-}362)$$

– Otherwise, if $xFrac_L$ is equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (\sum_{i=0}^{7} f_L[yFrac_L][i] * refPicLX_L[xInt_3][yInt_i]) >> shift1 \quad (8\text{-}363)$$

– Otherwise, if $xFrac_L$ is not equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

  – The sample array temp[n] with n = 0..7, is derived as follows:

$$temp[n] = (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_n]) >> shift1 \quad (8\text{-}364)$$

  – The predicted luma sample value $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (\sum_{i=0}^{7} f_L[yFrac_L][i] * temp[i]) >> shift2 \quad (8\text{-}365)$$

Table 8-7 – Specification of the luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p.

Fractional sample position p	interpolation filter coefficients							
	$f_L[p][0]$	$f_L[p][1]$	$f_L[p][2]$	$f_L[p][3]$	$f_L[p][4]$	$f_L[p][5]$	$f_L[p][6]$	$f_L[p][7]$
1	0	1	−3	63	4	−2	1	0
2	−1	2	−5	62	8	−3	1	0
3	−1	3	−8	60	13	−4	1	0
4	−1	4	−10	58	17	−5	1	0
5	−1	4	−11	52	26	−8	3	−1
6	−1	3	−9	47	31	−10	4	−1
7	−1	4	−11	45	34	−10	4	−1
8	−1	4	−11	40	40	−11	4	−1
9	−1	4	−10	34	45	−11	4	−1
10	−1	4	−10	31	47	−9	3	−1
11	−1	3	−8	26	52	−11	4	−1
12	0	1	−5	17	58	−10	4	−1
13	0	1	−4	13	60	−8	3	−1
14	0	1	−3	8	62	−5	2	−1
15	0	1	−2	4	63	−3	1	0

8.3.4.3.3 Chroma sample interpolation process

Inputs to this process are:

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

– a chroma location in 1/32 fractional-sample units ($xFrac_C$, $yFrac_C$),

– the chroma reference sample array $refPicLX_C$.

Output of this process is a predicted chroma sample value $predSampleLX_C$

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min(4, $BitDepth_C$ − 8), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max(2, 14 − $BitDepth_C$).

– The variable $picW_C$ is set equal to pic_width_in_luma_samples / SubWidthC and the variable $picH_C$ is set equal to pic_height_in_luma_samples / SubHeightC.

The luma interpolation filter coefficients $f_C[p]$ for each 1/32 fractional sample position p equal to $xFrac_C$ or $yFrac_C$ are specified in Table 8-8.

The chroma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..3:

$$xInt_i = Clip3(0, picW_C − 1, xInt_C + i − 1) \quad (8\text{-}366)$$

$$yInt_i = Clip3(0, picH_C − 1, yInt_C + i − 1) \quad (8\text{-}367)$$

The predicted chroma sample value $predSampleLX_C$ is derived as follows:

– If both $xFrac_C$ and $yFrac_C$ are equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = refPicLX_C[xInt_1][yInt_1] << shift3 \quad (8\text{-}368)$$

– Otherwise if $xFrac_C$ is not equal to 0 and $yFrac_C$ is equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = (\sum_{i=0}^{3} f_C[xFrac_C][i] * refPicLX_C[xInt_i][yInt_1]) >> shift1 \quad (8\text{-}369)$$

– Otherwise if $xFrac_C$ is equal to 0 and $yFrac_C$ is not equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = (\sum_{i=0}^{3} f_C[yFrac_C][i] * refPicLX_C[xInt_1][yInt_i]) >> shift1 \quad (8\text{-}370)$$

– Otherwise if $xFrac_C$ is not equal to 0 and $yFrac_C$ is not equal to 0, the value of $predSampleLX_C$ is derived as follows:

  – The sample array temp[n] with n = 0..3, is derived as follows:

$$temp[n] = (\sum_{i=0}^{3} f_C[xFrac_C][i] * refPicLX_C[xInt_i][yInt_n]) >> shift1 \quad (8\text{-}371)$$

  – The predicted chroma sample value $predSampleLX_C$ is derived as follows:

$$\begin{aligned} predSampleLX_C = (& f_C[yFrac_C][0] * temp[0] + \\ & f_C[yFrac_C][1] * temp[1] + \\ & f_C[yFrac_C][2] * temp[2] + \\ & f_C[yFrac_C][3] * temp[3]) >> shift2 \end{aligned} \quad (8\text{-}372)$$

Table 8-8 – Specification of the chroma interpolation filter coefficients $f_C[p]$ for each 1/32 fractional sample position p.

Fractional sample position p	interpolation filter coefficients			
	$f_C[p][0]$	$f_C[p][1]$	$f_C[p][2]$	$f_C[p][3]$
1	−1	63	2	0
2	−2	62	4	0
3	-2	60	7	−1
4	−2	58	10	−2
5	−3	57	12	−2
6	−4	56	14	−2
7	−4	55	15	−2
8	−4	54	16	−2
9	−5	53	18	−2
10	−6	52	20	−2
11	−6	49	24	−3
12	−6	46	28	−4
13	−5	44	29	−4
14	−4	42	30	−4
15	−4	39	33	−4
16	−4	36	36	−4
17	−4	33	39	−4
18	−4	30	42	−4
19	−4	29	44	−5
20	−4	28	46	−6
21	−3	24	49	−6
22	−2	20	52	−6
23	−2	18	53	−5
24	−2	16	54	−4
25	−2	15	55	−4
26	−2	14	56	−4
27	−2	12	57	−3
28	−2	10	58	−2
29	−1	7	60	−2
30	0	4	62	−2
31	0	2	63	−1

8.3.4.4 Weighted sample prediction process

Inputs to this process are:

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

– two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the reference indices refIdxL0 and refIdxL1,

– a variable cIdx specifying colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable bitDepth is derived as follows:

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

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

Variables shift1, shift2, offset1 and offset2 are derived as follows:

– The variable shift1 is set equal to Max(2, 14 − bitDepth) and the variable shift2 is set equal to Max(3, 15 − bitDepth).

– The variable offset1 is set equal to 1 << (shift1 − 1).

– The variable offset2 is set equal to 1 << (shift2 − 1).

Depending on the values of predFlagL0 and predFlagL1, the prediction samples pbSamples[x][y] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

– If predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction sample values are derived as follows:

$$pbSamples[x][y] = Clip3(0, (1 << bitDepth) - 1, (predSamplesL0[x][y] + offset1) >> shift1) \quad (8\text{-}373)$$

– Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

$$pbSamples[x][y] = Clip3(0, (1 << bitDepth) - 1, (predSamplesL1[x][y] + offset1) >> shift1) \quad (8\text{-}374)$$

– Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

$$pbSamples[x][y] = Clip3(0, (1 << bitDepth) - 1, (predSamplesL0[x][y] + predSamplesL1[x][y] + offset2) >> shift2) \quad (8\text{-}375)$$

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

Inputs to this process are:

– a sample location (xTb0, yTb0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable nTbW specifying the width of the current transform block,

– a variable nTbH specifying the height of the current transform block,

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

Output of this process is an (nTbW)x(nTbH) array resSamples.

The maximum transform block size maxTbSize is derived as follows:

$$maxTbSize = (cIdx == 0) ? MaxTbSizeY : MaxTbSizeY / 2 \quad (8\text{-}376)$$

The luma sample location is derived as follows:

$$(xTbY, yTbY) = (cIdx == 0) ? (xTb0, yTb0) : (xTb0 / 2, yTb0 / 2) \quad (8\text{-}377)$$

Depending on maxTbSize, the following applies:

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

1. The variables newTbW and newTbH are derived as follows:

$$newTbW = (nTbW > maxTbSize) ? (nTbW / 2) : nTbW \quad (8\text{-}378)$$

$$newTbH = (nTbH > maxTbSize) ? (nTbH / 2) : nTbH \quad (8\text{-}379)$$

2. The decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
3. When nTbW is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture .
4. When nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0, yTb0 + newTbH), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
5. Wwhen nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0 + newTbH), the transform block width nTbW set equal to newTbW and height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

– Otherwise, the scaling and transformation process as specified in clause 8.4.2 is invoked with the luma location (xTbY, yTbY), the variable cIdx, the transform width nTbW and the transform height nTbH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.

8.4 Scaling, transformation and array construction process

8.4.1 Derivation process for quantization parameters

In this process, the luma quantization parameter Qp'_Y and the chroma quantization parameters Qp'_{Cb} and Qp'_{Cr} are derived.

[Ed. (BB): QP signalling, derivation as well as invokation of this process to be defined.]

8.4.2 Scaling and transformation process

Inputs to this process are:

– a luma location (xTbY, yTbY) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the current picture,

– a variable cIdx specifying the colour component of the current block,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height.

Output of this process is the (nTbW)x(nTbH) array of residual samples resSamples[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables bitDepth, bdShift and tsShift are derived as follows:

$$bitDepth = (cIdx == 0) ? BitDepth_Y : BitDepth_C \quad (8\text{-}380)$$

$$bdShift = Max(22 - bitDepth, 0) \quad (8\text{-}381)$$

$$tsShift = 5 + ((Log2(nTbW) + Log2(nTbH)) / 2) \quad (8\text{-}382)$$

The (nTbW)x(nTbH) array of residual samples resSamples is derived as follows:

1. The scaling process for transform coefficients as specified in clause 8.4.3 is invoked with the transform block location (xTbY, yTbY), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx and the bit depth of the current colour component bitDepth as inputs, and the output is an (nTbW)x(nTbH) array of scaled transform coefficients d.
2. The (nTbW)x(nTbH) array of residual samples r is derived as follows:

– If transform_skip_flag[xTbY][yTbY][cIdx] is equal to 1, the residual sample array values r[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

$$r[x][y] = d[x][y] << tsShift \quad (8\text{-}383)$$

– Otherwise (transform_skip_flag[xTbY][yTbY][cIdx] is equal to 0), the transformation process for scaled transform coefficients as specified in clause 8.4.4.1 is invoked with the transform block location (xTbY, yTbY), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx and the (nTbW)x(nTbH) array of scaled transform coefficients d as inputs, and the output is an (nTbW)x(nTbH) array of residual samples r.

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

$$resSamples[x][y] = (r[x][y] + (1 << (bdShift − 1))) >> bdShift \quad (8\text{-}384)$$

8.4.3 Scaling process for transform coefficients

Inputs to this process are:

– a luma location (xTbY, yTbY) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the current picture,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable cIdx specifying the colour component of the current block,

– a variable bitDepth specifying the bit depth of the current colour component.

Output of this process is the (nTbW)x(nTbH) array d of scaled transform coefficients with elements d[x][y].

The quantization parameter qP is derived as follows:

– If cIdx is equal to 0, the following applies:

$$qP = Qp'_Y \quad (8\text{-}385)$$

– Otherwise, if cIdx is equal to 1, the following applies:

$$qP = Qp'_{Cb} \quad (8\text{-}386)$$

– Otherwise (cIdx is equal to 2), the following applies:

$$qP = Qp'_{Cr} \quad (8\text{-}387)$$

The variables bdShift, rectNorm and bdOffset are derived as follows:

$$bdShift = bitDepth + (((Log2(nTbW) + Log2(nTbH)) \& 1) * 8 + (Log2(nTbW) + Log2(nTbH)) / 2) − 5 + dep\_quant\_enabled\_flag \quad (8\text{-}388)$$

$$rectNorm = ((Log2(nTbW) + Log2(nTbH)) \& 1) == 1 ? 181 : 1 \quad (8\text{-}389)$$

$$bdOffset = (1 << bdShift) >> 1 \quad (8\text{-}390)$$

The list levelScale[ ] is specified as levelScale[k] = { 40, 45, 51, 57, 64, 72 } with k = 0..5.

For the derivation of the scaled transform coefficients d[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1, the following applies:

– The intermediate scaling factor m[x][y] is set equal to 16.

– The scaling factor ls[x][y] is derived as follows:

  – If dep_quant_enabled_flag is equal to 1, the following applies:

$$ls[x][y] = (m[x][y] * levelScale[(qP + 1) \% 6]) << ((qP + 1) / 6) \quad (8\text{-}391)$$

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

ls[x][y] = (m[x][y] * levelScale[qP % 6]) << (qP / 6) (8-392)

– The value dnc[x][y] is derived as follows:

dnc[x][y] = (8-393)
(TransCoeffLevel[xTbY][yTbY][cIdx][x][y] * ls[x][y] * rectNorm +bdOffset) >> bdShift

– The scaled transform coefficient d[x][y] is derived as follows:

d[x][y] = Clip3(CoeffMin, CoeffMax, dnc[x][y]) (8-394)

8.4.4 Transformation process for scaled transform coefficients

8.4.4.1 General

Inputs to this process are:

– a luma location (xTbY, yTbY) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the current picture,

– a variable nTbW specifying the width of the current transform block,

– a variable nTbH specifying the height of the current transform block,

– a variable cIdx specifying the colour component of the current block,

– an (nTbW)x(nTbH) array d[x][y] of scaled transform coefficients with x = 0..nTbW − 1, y = 0..nTbH − 1.

Output of this process is the (nTbW)x(nTbH) array r[x][y] of residual samples with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable trTypeHor specifying the horizontal transform kernel and the variable trTypeVer specifying the vertical transform kernel are derived in Table 8-9 depending on mts_idx[xTbY][yTbY][cIdx and CuPredMode[xTbY][yTbY].

The variables nonZeroW and nonZeroH are derived as follows:

nonZeroW = Min(nTbW, 32) (8-395)

nonZeroH = Min(nTbH, 32) (8-396)

The (nTbW)x(nTbH) array r of residual samples is derived as follows:

1. Each (vertical) column of scaled transform coefficients d[x][y] with x = 0..nonZeroW − 1, y = 0..nonZeroH − 1 is transformed to e[x][y] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.4.4.2 for each column x = 0..nonZeroW − 1 with the height of the transform block nTbH, the non-zero height of the scaled transform coefficients nonZeroH, the list d[x][y] with y = 0..nonZeroH − 1 and the transform type variable trType set equal to trTypeVer as inputs, and the output is the list e[x][y] with y = 0..nTbH − 1.
2. The intermediate sample values g[x][y] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 are derived as follows:

g[x][y] = Clip3(CoeffMin, CoeffMax, (e[x][y] + 256) >> 9) (8-397)

3. Each (horizontal) row of the resulting array g[x][y] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 is transformed to r[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.4.4.2 for each row y = 0..nTbH − 1 with the width of the transform block nTbW, the non-zero width of the resulting array g[x][y] nonZeroW, the list g[x][y] with x = 0..nonZeroW − 1 and the transform type variable trType set equal to trTypeHor as inputs, and the output is the list r[x][y] with x = 0..nTbW − 1.

Table 8-9 – Specification of trTypeHor and trTypeVer depending on mts_idx[x][y][cIdx and CuPredMode[x][y]

mts_idx[xTbY][yTbY][cIdx	CuPredMode[xTbY][yTbY] = = MODE_INTRA		CuPredMode[xTbY][xTbY] = = MODE_INTER	
	trTypeHor	trTypeVer	trTypeHor	trTypeVer

−1	0	0	0	0
0	1	1	2	2
1	2	1	1	2
2	1	2	2	1
3	2	2	1	1

8.4.4.2 Transformation process

Inputs to this process are:

– a variable nTbS specifying the horizontal sample size of transformed samples,

– a variable nonZeroS specifying the horizontal sample size of non-zero scaled transform coefficients,

– a list of scaled transform coefficients x[j] with j = 0..nonZeroS − 1,

– a transform kernel type variable trType.

Output of this process is the list of transformed samples y[i] with i = 0..nTbS − 1.

The transformation matrix derivation process as specified in clause 8.4.4.3 in invoked with the transform size nTbS and the transform kernel Type trType as inputs, and the transformation maxtrix transMatrix as output.

Depending on the value of trType, the following applies:, the list of transformed samples y[i] with i = 0..nTbS − 1 is derived as follows:

– If trType is equal to 0, the following transform matrix multiplication applies:

$$y[i]= \sum_{j=0}^{nonZeroS-1} transMatrix[i][j * 2^{6-Log2(nTbS)}] * x[j] \quad \text{with } i = 0..nTbS - 1 \tag{8-398}$$

– Otherwise (trType is equal to 1 or trType is equal to 2), the following transform matrix multiplication applies:

$$y[i]= \sum_{j=0}^{nonZeroS-1} transMatrix[i][j] * x[j] \quad \text{with } i = 0..nTbS - 1 \tag{8-399}$$

8.4.4.3 Transformation matrix derivation process

Inputs to this process are:

– a variable nTbS specifying the horizontal sample size of scaled transform coefficients,

– the transformation kernel type trType.

Output of this process is the transformation matrix transMatrix.

The transformation matrix transMatrix is derived based on trType and nTbs as follows:

– If trType is equal to 0, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..63 (8-400)

transMatrixCol0to15 = (8-401)

```
{
{ 256  256  256  256  256  256  256  256  256  256  256  256  256  256  256  256 }
{ 362  361  359  357  353  349  344  338  331  323  315  306  296  285  274  262 }
{ 362  358  351  341  327  311  291  268  243  216  186  155  122   88   53   18 }
{ 361  353  338  315  285  250  208  163  114   62    9  -44  -97 -147 -194 -236 }
{ 360  346  319  280  230  171  105   35  -35 -105 -171 -230 -280 -319 -346 -360 }
{ 359  338  296  236  163   79   -9  -97 -178 -250 -306 -344 -361 -357 -331 -285 }
{ 358  327  268  186   88  -18 -122 -216 -291 -341 -362 -351 -311 -243 -155  -53 }
{ 357  315  236  130    9 -114 -223 -306 -353 -359 -323 -250 -147  -27   97  208 }
{ 355  301  201   71  -71 -201 -301 -355 -355 -301 -201  -71   71  201  301  355 }
{ 353  285  163    9 -147 -274 -349 -357 -296 -178  -27  130  262  344  359  306 }
{ 351  268  122  -53 -216 -327 -362 -311 -186  -18  155  291  358  341  243   88 }
{ 349  250   79 -114 -274 -357 -338 -223  -44  147  296  361  323  194    9 -178 }
{ 346  230   35 -171 -319 -360 -280 -105  105  280  360  319  171  -35 -230 -346 }
{ 344  208   -9 -223 -349 -338 -194   27  236  353  331  178  -44 -250 -357 -323 }
{ 341  186  -53 -268 -362 -291  -88  155  327  351  216  -18 -243 -358 -311 -122 }
{ 338  163  -97 -306 -357 -223   27  262  362  274   44 -208 -353 -315 -114  147 }
{ 334  139 -139 -334 -334 -139  139  334  334  139 -139 -334 -334 -139  139  334 }
{ 331  114 -178 -353 -296  -44  236  362  250  -27 -285 -357 -194   97  323  338 }
{ 327   88 -216 -362 -243   53  311  341  122 -186 -358 -268   18  291  351  155 }
{ 323   62 -250 -359 -178  147  353  274  -27 -306 -338  -97  223  362  208 -114 }
{ 319   35 -280 -346 -105  230  360  171 -171 -360 -230  105  346  280  -35 -319 }
{ 315    9 -306 -323  -27  296  331   44 -285 -338  -62  274  344   79 -262 -349 }
{ 311  -18 -327 -291   53  341  268  -88 -351 -243  122  358  216 -155 -362 -186 }
{ 306  -44 -344 -250  130  361  178 -208 -357  -97  274  331    9 -323 -285   79 }
{ 301  -71 -355 -201  201  355   71 -301 -301   71  355  201 -201 -355  -71  301 }
{ 296  -97 -361 -147  262  323  -44 -353 -194  223  344    9 -338 -236  178  357 }
{ 291 -122 -362  -88  311  268 -155 -358  -53  327  243 -186 -351  -18  341  216 }
{ 285 -147 -357  -27  344  194 -250 -315   97  362   79 -323 -236  208  338  -44 }
{ 280 -171 -346   35  360  105 -319 -230  230  319 -105 -360  -35  346  171 -280 }
{ 274 -194 -331   97  359    9 -357 -114  323  208 -262 -285  178  338  -79 -361 }
{ 268 -216 -311  155  341  -88 -358   18  362   53 -351 -122  327  186 -291 -243 }
{ 262 -236 -285  208  306 -178 -323  147  338 -114 -349   79  357  -44 -361    9 }
{ 256 -256 -256  256  256 -256 -256  256  256 -256 -256  256  256 -256 -256  256 }
{ 250 -274 -223  296  194 -315 -163  331  130 -344  -97  353   62 -359  -27  362 }
{ 243 -291 -186  327  122 -351  -53  362  -18 -358   88  341 -155 -311  216  268 }
{ 236 -306 -147  349   44 -362   62  344 -163 -296  250  223 -315 -130  353   27 }
{ 230 -319 -105  360  -35 -346  171  280 -280 -171  346   35 -360  105  319 -230 }
{ 223 -331  -62  361 -114 -306  262  178 -349   -9  353 -163 -274  296  130 -359 }
{ 216 -341  -18  351 -186 -243  327   53 -358  155  268 -311  -88  362 -122 -291 }
{ 208 -349   27  331 -250 -163  359  -79 -306  285  114 -362  130  274 -315  -62 }
{ 201 -355   71  301 -301  -71  355 -201 -201  355  -71 -301  301   71 -355  201 }
{ 194 -359  114  262 -338   27  315 -296  -62  349 -236 -147  362 -163 -223  353 }
{ 186 -362  155  216 -358  122  243 -351   88  268 -341   53  291 -327   18  311 }
{ 178 -362  194  163 -361  208  147 -359  223  130 -357  236  114 -353  250   97 }
{ 171 -360  230  105 -346  280   35 -319  319  -35 -280  346 -105 -230  360 -171 }
{ 163 -357  262   44 -315  331  -79 -236  361 -194 -130  349 -285   -9  296 -344 }
{ 155 -351  291  -18 -268  358 -186 -122  341 -311   53  243 -362  216   88 -327 }
{ 147 -344  315  -79 -208  359 -274    9  262 -361  223   62 -306  349 -163 -130 }
{ 139 -334  334 -139 -139  334 -334  139  139 -334  334 -139 -139  334 -334  139 }
{ 130 -323  349 -194  -62  285 -361  250   -9 -236  359 -296   79  178 -344  331 }
{ 122 -311  358 -243   18  216 -351  327 -155  -88  291 -362  268  -53 -186  341 }
{ 114 -296  362 -285   97  130 -306  361 -274   79  147 -315  359 -262   62  163 }
{ 105 -280  360 -319  171   35 -230  346 -346  230  -35 -171  319 -360  280 -105 }
{  97 -262  353 -344  236  -62 -130  285 -359  331 -208   27  163 -306  362 -315 }
{  88 -243  341 -358  291 -155  -18  186 -311  362 -327  216  -53 -122  268 -351 }
{  79 -223  323 -362  331 -236   97   62 -208  315 -361  338 -250  114   44 -194 }
{  71 -201  301 -355  355 -301  201  -71  -71  201 -301  355 -355  301 -201   71 }
{  62 -178  274 -338  362 -344  285 -194   79   44 -163  262 -331  361 -349  296 }
{  53 -155  243 -311  351 -362  341 -291  216 -122   18   88 -186  268 -327  358 }
{  44 -130  208 -274  323 -353  362 -349  315 -262  194 -114   27   62 -147  223 }
{  35 -105  171 -230  280 -319  346 -360  360 -346  319 -280  230 -171  105  -35 }
{  27  -79  130 -178  223 -262  296 -323  344 -357  362 -359  349 -331  306 -274 }
{  18  -53   88 -122  155 -186  216 -243  268 -291  311 -327  341 -351  358 -362 }
{   9  -27   44  -62   79  -97  114 -130  147 -163  178 -194  208 -223  236 -250 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..63 (8-402)

transMatrixCol16to31 = (8-403)

```
{
{ 256  256  256  256  256  256  256  256  256  256  256  256  256  256  256  256 }
{ 250  236  223  208  194  178  163  147  130  114   97   79   62   44   27    9 }
{ -18  -53  -88 -122 -155 -186 -216 -243 -268 -291 -311 -327 -341 -351 -358 -362 }
{-274 -306 -331 -349 -359 -362 -357 -344 -323 -296 -262 -223 -178 -130  -79  -27 }
{-360 -346 -319 -280 -230 -171 -105  -35   35  105  171  230  280  319  346  360 }
{-223 -147  -62   27  114  194  262  315  349  362  353  323  274  208  130   44 }
{  53  155  243  311  351  362  341  291  216  122   18  -88 -186 -268 -327 -358 }
{ 296  349  361  331  262  163   44  -79 -194 -285 -344 -362 -338 -274 -178  -62 }
{ 355  301  201   71  -71 -201 -301 -355 -355 -301 -201  -71   71  201  301  355 }
{ 194   44 -114 -250 -338 -361 -315 -208  -62   97  236  331  362  323  223   79 }
{ -88 -243 -341 -358 -291 -155   18  186  311  362  327  216   53 -122 -268 -351 }
{-315 -362 -306 -163   27  208  331  359  285  130  -62 -236 -344 -353 -262  -97 }
{-346 -230  -35  171  319  360  280  105 -105 -280 -360 -319 -171   35  230  346 }
{-163   62  262  359  315  147  -79 -274 -361 -306 -130   97  285  362  296  114 }
{ 122  311  358  243   18 -216 -351 -327 -155   88  291  362  268   53 -186 -341 }
{ 331  344  178  -79 -296 -359 -236    9  250  361  285   62 -194 -349 -323 -130 }
{ 334  139 -139 -334 -334 -139  139  334  334  139 -139 -334 -334 -139  139  334 }
{ 130 -163 -349 -306  -62  223  361  262   -9 -274 -359 -208   79  315  344  147 }
{-155 -351 -291  -18  268  358  186 -122 -341 -311  -53  243  362  216  -88 -327 }
{-344 -296   -9  285  349  130 -194 -361 -236   79  331  315   44 -262 -357 -163 }
{-319  -35  280  346  105 -230 -360 -171  171  360  230 -105 -346 -280   35  319 }
{ -97  250  353  114 -236 -357 -130  223  359  147 -208 -361 -163  194  362  178 }
{ 186  362  155 -216 -358 -122  243  351   88 -268 -341  -53  291  327   18 -311 }
{ 353  223 -163 -362 -147  236  349   62 -296 -315   27  338  262 -114 -359 -194 }
{ 301  -71 -355 -201  201  355   71 -301 -301   71  355  201 -201 -355  -71  301 }
{  62 -315 -274  130  362  114 -285 -306   79  359  163 -250 -331   27  349  208 }
{-216 -341   18  351  186 -243 -327   53  358  155 -268 -311   88  362  122 -291 }
{-359 -130  296  274 -163 -353   -9  349  178 -262 -306  114  361   62 -331 -223 }
{-280  171  346  -35 -360 -105  319  230 -230 -319  105  360   35 -346 -171  280 }
{ -27  353  130 -315 -223  250  296 -163 -344   62  362   44 -349 -147  306  236 }
{ 243  291 -186 -327  122  351  -53 -362  -18  358   88 -341 -155  311  216 -268 }
{ 362   27 -359  -62  353   97 -344 -130  331  163 -315 -194  296  223 -274 -250 }
{ 256 -256 -256  256  256 -256 -256  256  256 -256 -256  256  256 -256 -256  256 }
{  -9 -361   44  357  -79 -349  114  338 -147 -323  178  306 -208 -285  236  262 }
{-268 -216  311  155 -341  -88  358   18 -362   53  351 -122 -327  186  291 -243 }
{-361   79  338 -178 -285  262  208 -323 -114  357    9 -359   97  331 -194 -274 }
{-230  319  105 -360   35  346 -171 -280  280  171 -346  -35  360 -105 -319  230 }
{  44  338 -208 -236  323   79 -362   97  315 -250 -194  344   27 -357  147  285 }
{ 291  122 -362   88  311 -268 -155  358  -53 -327  243  186 -351   18  341 -216 }
{ 357 -178 -236  338    9 -344  223  194 -353   44  323 -262 -147  361  -97 -296 }
{ 201 -355   71  301 -301  -71  355 -201 -201  355  -71 -301  301   71 -355  201 }
{ -79 -285  323    9 -331  274   97 -357  208  178 -361  130  250 -344   44  306 }
{-311  -18  327 -291  -53  341 -268  -88  351 -243 -122  358 -216 -155  362 -186 }
{-349  262   79 -344  274   62 -338  285   44 -331  296   27 -323  306    9 -315 }
{-171  360 -230 -105  346 -280  -35  319 -319   35  280 -346  105  230 -360  171 }
{ 114  208 -362  223   97 -338  306  -27 -274  353 -147 -178  359 -250  -62  323 }
{ 327  -88 -216  362 -243  -53  311 -341  122  186 -358  268   18 -291  351 -155 }
{ 338 -323   97  194 -357  285  -27 -250  362 -236  -44  296 -353  178  114 -331 }
{ 139 -334  334 -139 -139  334 -334  139  139 -334  334 -139 -139  334 -334  139 }
{-147 -114  315 -353  208   44 -274  362 -262   27  223 -357  306  -97 -163  338 }
{-341  186   53 -268  362 -291   88  155 -327  351 -216  -18  243 -358  311 -122 }
{-323  357 -250   44  178 -331  353 -236   27  194 -338  349 -223    9  208 -344 }
{-105  280 -360  319 -171  -35  230 -346  346 -230   35  171 -319  360 -280  105 }
{ 178    9 -194  323 -361  296 -147  -44  223 -338  357 -274  114   79 -250  349 }
{ 351 -268  122   53 -216  327 -362  311 -186   18  155 -291  358 -341  243  -88 }
{ 306 -359  344 -262  130   27 -178  296 -357  349 -274  147    9 -163  285 -353 }
{  71 -201  301 -355  355 -301  201  -71  -71  201 -301  355 -355  301 -201   71 }
{-208   97   27 -147  250 -323  359 -353  306 -223  114    9 -130  236 -315  357 }
{-358  327 -268  186  -88  -18  122 -216  291 -341  362 -351  311 -243  155  -53 }
{-285  331 -357  361 -344  306 -250  178  -97    9   79 -163  236 -296  338 -359 }
{ -35  105 -171  230 -280  319 -346  360 -360  346 -319  280 -230  171 -105   35 }
{ 236 -194  147  -97   44    9  -62  114 -163  208 -250  285 -315  338 -353  361 }
{ 362 -358  351 -341  327 -311  291 -268  243 -216  186 -155  122  -88   53  -18 }
{ 262 -274  285 -296  306 -315  323 -331  338 -344  349 -353  357 -359  361 -362 }
},
```

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol16to31[47 −m][n] (8-404)
with m =32..47, n = 0..63

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol0to15[63 −m][n] (8-405)
with m =48..63, n = 0..63

– Otherwise, if trType is equal to 1 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (8-406)

```
{
{ 117  219  296  336 }
{ 296  296    0 -296 }
{ 336 -117 -296  219 }
{ 219 -336  296 -117 }
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 8, the following applies:

transMatrix[m][n] = (8-407)

```
{
{  65  127  185  237  280  314  338  350 }
{ 185  314  350  280  127  -65 -237 -338 }
{ 280  338  127 -185 -350 -237   65  314 }
{ 338  185 -237 -314   65  350  127 -280 }
{ 350  -65 -338  127  314 -185 -280  237 }
{ 314 -280  -65  338 -237 -127  350 -185 }
{ 237 -350  280  -65 -185  338 -314  127 }
{ 127 -237  314 -350  338 -280  185  -65 }
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-408)

```
{
{  34   67  100  133  163  193  220  246  269  290  309  324  337  346  353  356 }
{ 100  193  269  324  353  353  324  269  193  100    0 -100 -193 -269 -324 -353 }
{ 163  290  353  337  246  100  -67 -220 -324 -356 -309 -193  -34  133  269  346 }
{ 220  346  324  163  -67 -269 -356 -290 -100  133  309  353  246   34 -193 -337 }
{ 269  353  193 -100 -324 -324 -100  193  353  269    0 -269 -353 -193  100  324 }
{ 309  309    0 -309 -309    0  309  309    0 -309 -309    0  309  309    0 -309 }
{ 337  220 -193 -346  -34  324  246 -163 -353  -67  309  269 -133 -356 -100  290 }
{ 353  100 -324 -193  269  269 -193 -324  100  353    0 -353 -100  324  193 -269 }
{ 356  -34 -353   67  346 -100 -337  133  324 -163 -309  193  290 -220 -269  246 }
{ 346 -163 -269  290  133 -353   34  337 -193 -246  309  100 -356   67  324 -220 }
{ 324 -269 -100  353 -193 -193  353 -100 -269  324    0 -324  269  100 -353  193 }
{ 290 -337  100  220 -356  193  133 -346  269   34 -309  324  -67 -246  353 -163 }
{ 246 -356  269  -34 -220  353 -290   67  193 -346  309 -100 -163  337 -324  133 }
{ 193 -324  353 -269  100  100 -269  353 -324  193    0 -193  324 -353  269 -100 }
{ 133 -246  324 -356  337 -269  163  -34 -100  220 -309  353 -346  290 -193   67 }
{  67 -133  193 -246  290 -324  346 -356  353 -337  309 -269  220 -163  100  -34 }
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 32, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..31 (8-409)

transMatrixCol0to15 = (8-410)

```
{
{  17   35   52   69   86  103  119  135  151  167  182  197  211  225  238  251 }
{  52  103  151  197  238  275  305  329  347  357  359  354  342  322  296  263 }
{  86  167  238  296  336  357  357  336  296  238  167   86    0  -86 -167 -238 }
{ 119  225  305  351  357  322  251  151   35  -86 -197 -285 -342 -359 -336 -275 }
{ 151  275  347  354  296  182   35 -119 -251 -336 -358 -314 -211  -69   86  225 }
{ 182  314  359  305  167  -17 -197 -322 -358 -296 -151   35  211  329  357  285 }
{ 211  342  342  211    0 -211 -342 -342 -211    0  211  342  342  211    0 -211 }
{ 238  357  296   86 -167 -336 -336 -167   86  296  357  238    0 -238 -357 -296 }
{ 263  358  225  -52 -296 -351 -182  103  322  336  135 -151 -342 -314  -86  197 }
{ 285  347  135 -182 -357 -251   52  314  329   86 -225 -359 -211  103  336  305 }
{ 305  322   35 -285 -336  -69  263  347  103 -238 -354 -135  211  358  167 -182 }
{ 322  285  -69 -347 -238  135  358  182 -197 -357 -119  251  342   52 -296 -314 }
{ 336  238 -167 -357  -86  296  296  -86 -357 -167  238  336    0 -336 -238  167 }
{ 347  182 -251 -314   86  359  103 -305 -263  167  351   17 -342 -197  238  322 }
{ 354  119 -314 -225  238  305 -135 -351   17  357  103 -322 -211  251  296 -151 }
{ 358   52 -351 -103  336  151 -314 -197  285  238 -251 -275  211  305 -167 -329 }
{ 359  -17 -358   35  357  -52 -354   69  351  -86 -347  103  342 -119 -336  135 }
{ 357  -86 -336  167  296 -238 -238  296  167 -336  -86  357    0 -357   86  336 }
{ 351 -151 -285  275  167 -347  -17  354 -135 -296  263  182 -342  -35  357 -119 }
{ 342 -211 -211  342    0 -342  211  211 -342    0  342 -211 -211  342    0 -342 }
{ 329 -263 -119  358 -167 -225  347  -52 -305  296   69 -351  211  182 -357  103 }
{ 314 -305  -17  322 -296  -35  329 -285  -52  336 -275  -69  342 -263  -86  347 }
{ 296 -336   86  238 -357  167  167 -357  238   86 -336  296    0 -296  336  -86 }
{ 275 -354  182  119 -336  314  -69 -225  359 -238  -52  305 -342  135  167 -351 }
{ 251 -359  263  -17 -238  358 -275   35  225 -357  285  -52 -211  354 -296   69 }
{ 225 -351  322 -151  -86  285 -359  275  -69 -167  329 -347  211   17 -238  354 }
{ 197 -329  354 -263   86  119 -285  358 -314  167   35 -225  342 -347  238  -52 }
{ 167 -296  357 -336  238  -86  -86  238 -336  357 -296  167    0 -167  296 -357 }
{ 135 -251  329 -359  336 -263  151  -17 -119  238 -322  358 -342  275 -167   35 }
{ 103 -197  275 -329  357 -354  322 -263  182  -86  -17  119 -211  285 -336  358 }
{  69 -135  197 -251  296 -329  351 -359  354 -336  305 -263  211 -151   86  -17 }
{  35  -69  103 -135  167 -197  225 -251  275 -296  314 -329  342 -351  357 -359 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-411)

transMatrixCol16to31 = (8-412)

```
{
{ 263  275  285  296  305  314  322  329  336  342  347  351  354  357  358  359 }
{ 225  182  135   86   35  -17  -69 -119 -167 -211 -251 -285 -314 -336 -351 -358 }
{-296 -336 -357 -357 -336 -296 -238 -167  -86    0   86  167  238  296  336  357 }
{-182  -69   52  167  263  329  358  347  296  211  103  -17 -135 -238 -314 -354 }
{ 322  359  329  238  103  -52 -197 -305 -357 -342 -263 -135   17  167  285  351 }
{ 135  -52 -225 -336 -354 -275 -119   69  238  342  351  263  103  -86 -251 -347 }
{-342 -342 -211    0  211  342  342  211    0 -211 -342 -342 -211    0  211  342 }
{ -86  167  336  336  167  -86 -296 -357 -238    0  238  357  296   86 -167 -336 }
{ 354  285   35 -238 -359 -251   17  275  357  211  -69 -305 -347 -167  119  329 }
{  35 -263 -354 -167  151  351  275  -17 -296 -342 -119  197  358  238  -69 -322 }
{-359 -197  151  357  225 -119 -351 -251   86  342  275  -52 -329 -296   17  314 }
{  17  329  275  -86 -351 -225  151  359  167 -211 -354 -103  263  336   35 -305 }
{ 357   86 -296 -296   86  357  167 -238 -336    0  336  238 -167 -357  -86  296 }
{ -69 -358 -119  296  275 -151 -354  -35  336  211 -225 -329   52  357  135 -285 }
{-347   35  358   86 -329 -197  263  285 -167 -342   52  359   69 -336 -182  275 }
{ 119  347  -69 -357   17  359   35 -354  -86  342  135 -322 -182  296  225 -263 }
{ 329 -151 -322  167  314 -182 -305  197  296 -211 -285  225  275 -238 -263  251 }
{-167 -296  238  238 -296 -167  336   86 -357    0  357  -86 -336  167  296 -238 }
{-305  251  197 -336  -52  358 -103 -314  238  211 -329  -69  359  -86 -322  225 }
{ 211  211 -342    0  342 -211 -211  342    0 -342  211  211 -342    0  342 -211 }
{ 275 -322  -17  336 -251 -135  359 -151 -238  342  -35 -314  285   86 -354  197 }
{-251 -103  351 -238 -119  354 -225 -135  357 -211 -151  358 -197 -167  359 -182 }
{-238  357 -167 -167  357 -238  -86  336 -296    0  296 -336   86  238 -357  167 }
{ 285  -17 -263  357 -197 -103  329 -322   86  211 -358  251   35 -296  347 -151 }
{ 197 -351  305  -86 -182  347 -314  103  167 -342  322 -119 -151  336 -329  135 }
{-314  135  103 -296  358 -263   52  182 -336  342 -197  -35  251 -357  305 -119 }
{-151  305 -359  296 -135  -69  251 -351  336 -211   17  182 -322  357 -275  103 }
{ 336 -238   86   86 -238  336 -357  296 -167    0  167 -296  357 -336  238  -86 }
{ 103 -225  314 -357  347 -285  182  -52  -86  211 -305  354 -351  296 -197   69 }
{-351  314 -251  167  -69  -35  135 -225  296 -342  359 -347  305 -238  151  -52 }
{ -52  119 -182  238 -285  322 -347  358 -357  342 -314  275 -225  167 -103   35 }
{ 358 -354  347 -336  322 -305  285 -263  238 -211  182 -151  119  -86   52  -17 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (8-413)

```
{
{ 336  296  219  117 }
{ 296    0 -296 -296 }
{ 219 -296 -117  336 }
{ 117 -296  336 -219 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 8, the following applies:

transMatrix[m][n] = (8-414)

```
{
{ 350  338  314  280  237  185  127   65 }
{ 338  237   65 -127 -280 -350 -314 -185 }
{ 314   65 -237 -350 -185  127  338  280 }
{ 280 -127 -350  -65  314  237 -185 -338 }
{ 237 -280 -185  314  127 -338  -65  350 }
{ 185 -350  127  237 -338   65  280 -314 }
{ 127 -314  338 -185  -65  280 -350  237 }
{  65 -185  280 -338  350 -314  237 -127 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-415)

```
{
{ 356  353  346  337  324  309  290  269  246  220  193  163  133  100   67   34 }
{ 353  324  269  193  100    0 -100 -193 -269 -324 -353 -353 -324 -269 -193 -100 }
{ 346  269  133  -34 -193 -309 -356 -324 -220  -67  100  246  337  353  290  163 }
{ 337  193  -34 -246 -353 -309 -133  100  290  356  269   67 -163 -324 -346 -220 }
{ 324  100 -193 -353 -269    0  269  353  193 -100 -324 -324 -100  193  353  269 }
{ 309    0 -309 -309    0  309  309    0 -309 -309    0  309  309    0 -309 -309 }
{ 290 -100 -356 -133  269  309  -67 -353 -163  246  324  -34 -346 -193  220  337 }
{ 269 -193 -324  100  353    0 -353 -100  324  193 -269 -269  193  324 -100 -353 }
{ 246 -269 -220  290  193 -309 -163  324  133 -337 -100  346   67 -353  -34  356 }
{ 220 -324  -67  356 -100 -309  246  193 -337  -34  353 -133 -290  269  163 -346 }
{ 193 -353  100  269 -324    0  324 -269 -100  353 -193 -193  353 -100 -269  324 }
{ 163 -353  246   67 -324  309  -34 -269  346 -133 -193  356 -220 -100  337 -290 }
{ 133 -324  337 -163 -100  309 -346  193   67 -290  353 -220  -34  269 -356  246 }
{ 100 -269  353 -324  193    0 -193  324 -353  269 -100 -100  269 -353  324 -193 }
{  67 -193  290 -346  353 -309  220 -100  -34  163 -269  337 -356  324 -246  133 }
{  34 -100  163 -220  269 -309  337 -353  356 -346  324 -290  246 -193  133  -67 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 32, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..31 (8-416)

transMatrixCol0to15 = (8-417)

```
{
{ 359  358  357  354  351  347  342  336  329  322  314  305  296  285  275  263 }
{ 358  351  336  314  285  251  211  167  119   69   17  -35  -86 -135 -182 -225 }
{ 357  336  296  238  167   86    0  -86 -167 -238 -296 -336 -357 -357 -336 -296 }
{ 354  314  238  135   17 -103 -211 -296 -347 -358 -329 -263 -167  -52   69  182 }
{ 351  285  167   17 -135 -263 -342 -357 -305 -197  -52  103  238  329  359  322 }
{ 347  251   86 -103 -263 -351 -342 -238  -69  119  275  354  336  225   52 -135 }
{ 342  211    0 -211 -342 -342 -211    0  211  342  342  211    0 -211 -342 -342 }
{ 336  167  -86 -296 -357 -238    0  238  357  296   86 -167 -336 -336 -167   86 }
{ 329  119 -167 -347 -305  -69  211  357  275   17 -251 -359 -238   35  285  354 }
{ 322   69 -238 -358 -197  119  342  296   17 -275 -351 -151  167  354  263  -35 }
{ 314   17 -296 -329  -52  275  342   86 -251 -351 -119  225  357  151 -197 -359 }
{ 305  -35 -336 -263  103  354  211 -167 -359 -151  225  351   86 -275 -329  -17 }
{ 296  -86 -357 -167  238  336    0 -336 -238  167  357   86 -296 -296   86  357 }
{ 285 -135 -357  -52  329  225 -211 -336   35  354  151 -275 -296  119  358   69 }
{ 275 -182 -336   69  359   52 -342 -167  285  263 -197 -329   86  358   35 -347 }
{ 263 -225 -296  182  322 -135 -342   86  354  -35 -359  -17  357   69 -347 -119 }
{ 251 -263 -238  275  225 -285 -211  296  197 -305 -182  314  167 -322 -151  329 }
{ 238 -296 -167  336   86 -357    0  357  -86 -336  167  296 -238 -238  296  167 }
{ 225 -322  -86  359  -69 -329  211  238 -314 -103  358  -52 -336  197  251 -305 }
{ 211 -342    0  342 -211 -211  342    0 -342  211  211 -342    0  342 -211 -211 }
{ 197 -354   86  285 -314  -35  342 -238 -151  359 -135 -251  336  -17 -322  275 }
{ 182 -359  167  197 -358  151  211 -357  135  225 -354  119  238 -351  103  251 }
{ 167 -357  238   86 -336  296    0 -296  336  -86 -238  357 -167 -167  357 -238 }
{ 151 -347  296  -35 -251  358 -211  -86  322 -329  103  197 -357  263   17 -285 }
{ 135 -329  336 -151 -119  322 -342  167  103 -314  347 -182  -86  305 -351  197 }
{ 119 -305  357 -251   35  197 -342  336 -182  -52  263 -358  296 -103 -135  314 }
{ 103 -275  357 -322  182   17 -211  336 -351  251  -69 -135  296 -359  305 -151 }
{  86 -238  336 -357  296 -167    0  167 -296  357 -336  238  -86  -86  238 -336 }
{  69 -197  296 -351  354 -305  211  -86  -52  182 -285  347 -357  314 -225  103 }
{  52 -151  238 -305  347 -359  342 -296  225 -135   35   69 -167  251 -314  351 }
{  35 -103  167 -225  275 -314  342 -357  358 -347  322 -285  238 -182  119  -52 }
{  17  -52   86 -119  151 -182  211 -238  263 -285  305 -322  336 -347  354 -358 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-418)

transMatrixCol16to31 = (8-419)

```
{
{ 251  238  225  211  197  182  167  151  135  119  103   86   69   52   35   17 }
{-263 -296 -322 -342 -354 -359 -357 -347 -329 -305 -275 -238 -197 -151 -103  -52 }
{-238 -167  -86    0   86  167  238  296  336  357  357  336  296  238  167   86 }
{ 275  336  359  342  285  197   86  -35 -151 -251 -322 -357 -351 -305 -225 -119 }
{ 225   86  -69 -211 -314 -358 -336 -251 -119   35  182  296  354  347  275  151 }
{-285 -357 -329 -211  -35  151  296  358  322  197   17 -167 -305 -359 -314 -182 }
{-211    0  211  342  342  211    0 -211 -342 -342 -211    0  211  342  342  211 }
{ 296  357  238    0 -238 -357 -296  -86  167  336  336  167  -86 -296 -357 -238 }
{ 197  -86 -314 -342 -151  135  336  322  103 -182 -351 -296  -52  225  358  263 }
{-305 -336 -103  211  359  225  -86 -329 -314  -52  251  357  182 -135 -347 -285 }
{-182  167  358  211 -135 -354 -238  103  347  263  -69 -336 -285   35  322  305 }
{ 314  296  -52 -342 -251  119  357  197 -182 -358 -135  238  347   69 -285 -322 }
{ 167 -238 -336    0  336  238 -167 -357  -86  296  296  -86 -357 -167  238  336 }
{-322 -238  197  342  -17 -351 -167  263  305 -103 -359  -86  314  251 -182 -347 }
{-151  296  251 -211 -322  103  357   17 -351 -135  305  238 -225 -314  119  354 }
{ 329  167 -305 -211  275  251 -238 -285  197  314 -151 -336  103  351  -52 -358 }
{ 135 -336 -119  342  103 -347  -86  351   69 -354  -52  357   35 -358  -17  359 }
{-336  -86  357    0 -357   86  336 -167 -296  238  238 -296 -167  336   86 -357 }
{-119  357  -35 -342  182  263 -296 -135  354  -17 -347  167  275 -285 -151  351 }
{ 342    0 -342  211  211 -342    0  342 -211 -211  342    0 -342  211  211 -342 }
{ 103 -357  182  211 -351   69  296 -305  -52  347 -225 -167  358 -119 -263  329 }
{-347   86  263 -342   69  275 -336   52  285 -329   35  296 -322   17  305 -314 }
{ -86  336 -296    0  296 -336   86  238 -357  167  167 -357  238   86 -336  296 }
{ 351 -167 -135  342 -305   52  238 -359  225   69 -314  336 -119 -182  354 -275 }
{  69 -296  354 -211  -52  285 -357  225   35 -275  358 -238  -17  263 -359  251 }
{-354  238  -17 -211  347 -329  167   69 -275  359 -285   86  151 -322  351 -225 }
{ -52  238 -347  342 -225   35  167 -314  358 -285  119   86 -263  354 -329  197 }
{ 357 -296  167    0 -167  296 -357  336 -238   86   86 -238  336 -357  296 -167 }
{  35 -167  275 -342  358 -322  238 -119  -17  151 -263  336 -359  329 -251  135 }
{-358  336 -285  211 -119   17   86 -182  263 -322  354 -357  329 -275  197 -103 }
{ -17   86 -151  211 -263  305 -336  354 -359  351 -329  296 -251  197 -135   69 }
{ 359 -357  351 -342  329 -314  296 -275  251 -225  197 -167  135 -103   69  -35 }
},
```

8.4.5 Picture reconstruction process

[Ed. (BB): tbd]

8.5 In-loop Filter Process

8.5.1 General

The three in-loop filters, namely deblocking filter, sample adaptive offset and adaptive loop filter, are applied as specified by the following ordered steps:

1. For the deblocking filter, the following applies:
   - The deblocking filter process as specified in clause 8.5.2.1 is invoked with the reconstructed picture sample array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} after deblocking as outputs.
   - The array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} are assigned to the array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.
2. When sps_sao_enabled_flag is equal to 1, the following applies:
   - The sample adaptive offset process as specified in clause 8.5.3.1 is invoked with the reconstructed picture sample array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} after sample adaptive offset as outputs.
   - The array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} are assigned to the array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.
3. When sps_alf_enabled_flag is equal to 1, the following applies:
   - The adaptive loop filter process as specified in clause 8.5.4.1 is invoked with the reconstructed picture sample arrays S_L, S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L, S'_{Cb} and S'_{Cr} after sample adaptive offset as outputs.
   - The arrays S'_L, S'_{Cb} and S'_{Cr} are assigned to the arrays S_L, S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.

8.5.2 Deblocking filter process

8.5.2.1 General

Inputs to this process are the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

Outputs of this process are the modified reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

The vertical edges in a picture are filtered first. Then the horizontal edges in a picture are filtered with samples modified by the vertical edge filtering process as input. The vertical and horizontal edges in the CTBs of each CTU are processed separately on a coding unit basis. The vertical edges of the coding blocks in a coding unit are filtered starting with the edge on the left-hand side of the coding blocks proceeding through the edges towards the right-hand side of the coding blocks in their geometrical order. The horizontal edges of the coding blocks in a coding unit are filtered starting with the edge on the top of the coding blocks proceeding through the edges towards the bottom of the coding blocks in their geometrical order.

NOTE – Although the filtering process is specified on a picture basis in this Specification, the filtering process can be implemented on a coding unit basis with an equivalent result, provided the decoder properly accounts for the processing dependency order so as to produce the same output values.

The deblocking filter process is applied to all coding subblock edges and transform block edges of a picture, except the following types of edges:

– Edges that are at the boundary of the picture,

– Edges that coincide with tile boundaries when loop_filter_across_tiles_enabled_flag is equal to 0,

– Edges that coincide with upper or left boundaries of slices with slice_loop_filter_across_slices_enabled_flag equal to 0 or slice_deblocking_filter_disabled_flag equal to 1,

– Edges within slices with slice_deblocking_filter_disabled_flag equal to 1,

– Edges that do not correspond to 8x8 sample grid boundaries of the considered component,

– Edges within chroma components for which both sides of the edge use inter prediction,

– Edges of chroma transform blocks that are not edges of the associated transform unit.

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

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 8-10.

Table 8-10 – Name of association to edgeType

edgeType	Name of edgeType
0 (vertical edge)	EDGE_VER
1 (horizontal edge)	EDGE_HOR

When slice_deblocking_filter_disabled_flag of the current slice is equal to 0, the following applies:

– The variable treeType is derived as follows:

  – If slice_type is equal to I and qtbtt_dual_tree_intra_flag is equal to 1, treeType is set equal to DUAL_TREE_LUMA.

  – Otherwise, treeType is set equal to SINGLE_TREE.

– The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

– The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the modifed reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

– When slice_type is equal to I and qtbtt_dual_tree_intra_flag is equal to 1, the following applies:

  – The variable treeType is set equal to DUAL_TREE_CHROMA

  – The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

  – The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the modifed reconstructed picture after deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

8.5.2.2 Deblocking filter process for one direction

Inputs to this process are:

– the variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

For each coding unit with coding block width log2CbW, coding block height log2CbH and location of top-left sample of the coding block (xCb, yCb), when edgeType is equal to EDGE_VER and xCb % 8 is equal 0 or when edgeType is equal to EDGE_HOR and yCb % 8 is equal to 0, the edges are filtered by the following ordered steps:

1. The coding block width nCbW is set equal to 1 << log2CbW and the coding block height nCbH is set equal to 1 << log2CbH
2. The variable filterEdgeFlag is derived as follows:
   – If edgeType is equal to EDGE_VER and one or more of the following conditions are true, filterEdgeFlag is set equal to 0:
     – The left boundary of the current coding block is the left boundary of the picture.
     – The left boundary of the current coding block is the left boundary of the tile and loop_filter_across_tiles_enabled_flag is equal to 0.
     – The left boundary of the current coding block is the left boundary of the slice and slice_loop_filter_across_slices_enabled_flag is equal to 0.
   – Otherwise if edgeType is equal to EDGE_HOR and one or more of the following conditions are true, the variable filterEdgeFlag is set equal to 0:
     – The top boundary of the current luma coding block is the top boundary of the picture.
     – The top boundary of the current coding block is the top boundary of the tile and loop_filter_across_tiles_enabled_flag is equal to 0.
     – The top boundary of the current coding block is the top boundary of the slice and slice_loop_filter_across_slices_enabled_flag is equal to 0.
   – Otherwise, filterEdgeFlag is set equal to 1.

   [Ed. (BB): Adapt syntax once tiles are integrated.]
3. All elements of the two-dimensional (nCbW)x(nCbH) array edgeFlags are initialized to be equal to zero.
4. The derivation process of transform block boundary specified in clause 8.5.2.3 is invoked with the location (xB0, yB0) set equal to (0, 0), the block width nTbW set equal to nCbW, the block height nTbH set equal to nCbH, the variable treeType, the variable filterEdgeFlag, the array edgeFlags, and the variable edgeType as inputs, and the modified array edgeFlags as output.
5. The derivation process of coding subblock boundary specified in clause 8.5.2.4 is invoked with the location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the array edgeFlags, and the variable edgeType as inputs, and the modified array edgeFlags as output.
6. The picture sample array recPicture is derived as follows:
   – If treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, recPicture is set equal to the reconstructed luma picture sample array prior to deblocking $recPicture_L$.
   – Otherwise (treeType is equal to DUAL_TREE_CHROMA), recPicture is set equal to the reconstructed chroma picture sample array prior to deblocking $recPicture_{Cb}$.
7. The derivation process of the boundary filtering strength specified in clause 8.5.2.5 is invoked with the picture sample array recPicture, the luma location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the variable edgeType, and the array edgeFlags as inputs, and an (nCbW)x(nCbH) array verBs as output.
8. The edge filtering process is invoked as follows:
   – If edgeType is equal to EDGE_VER, the vertical edge filtering process for a coding unit as specified in clause 8.5.2.6.1 is invoked with the variable treeType, the reconstructed picture prior to deblocking, i.e.,

when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, the location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, and the array verBs as inputs, and the modified reconstructed picture, i.e., when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, as output.

– Otherwise if edgeType is equal to EDGE_HOR, the horizontal edge filtering process for a coding unit as specified in clause 8.5.2.6.2 is invoked with the variable treeType, the modified reconstructed picture prior to deblocking, i.e., when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, the location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, and the array horBs as inputs and the modified reconstructed picture, i.e., when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, as output.

8.5.2.3 Derivation process of transform block boundary

Inputs to this process are:

– a location (xB0, yB0) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable nTbW specifying the width of the current block,

– a variable nTbH specifying the height of the current block,

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– a variable filterEdgeFlag,

– a two-dimensional (nCbW)x(nCbH) array edgeFlags,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbW)x(nCbH) array edgeFlags.

The maximum transform block size maxTbSize is derived as follows:

$$maxTbSize = (treeType == DUAL\_TREE\_CHROMA) ? MaxTbSizeY / 2 : MaxTbSizeY \quad (8\text{-}420)$$

Depending on maxTbSize, the following applies:

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

1. The variables newTbW and newTbH are derived as follows:

$$newTbW = (nTbW > maxTbSize) ? (nTbW / 2) : nTbW \quad (8\text{-}421)$$

$$newTbH = (nTbH > maxTbSize) ? (nTbH / 2) : nTbH \quad (8\text{-}422)$$

2. The derivation process of transform block boundary as specified in this clause is invoked with the location (xB0, yB0), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags, and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

3. If nTbW is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

4. If nTbH is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location (xTb0, yTb0) set equal to (xTb0, yTb0 + newTbH), the variables nTbW set

equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

5. If nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0 + newTbH), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

– Otherwise, the following applies:
  – If edgeType is equal to EDGE_VER, the value of edgeFlags[xB0][yB0 + k] for k = 0..nTbH − 1 is derived as follows:
    – If xB0 is equal to 0, edgeFlags[xB0][yB0 + k] is set equal to filterEdgeFlag.
    – Otherwise, edgeFlags[xB0][yB0 + k] is set equal to 1.
  – Otherwise (edgeType is equal to EDGE_HOR), the value of edgeFlags[xB0 + k][yB0] for k = 0..nTbW − 1 is derived as follows:
    – If yB0 is equal to 0, edgeFlags[xB0 + k][yB0] is set equal to filterEdgeFlag.
    – Otherwise, edgeFlags[xB0 + k][yB0] is set equal to 1.

8.5.2.4 Derivation process of coding subblock boundary

Inputs to this process are:

– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
– a variable nCbW specifying the width of the current coding block,
– a variable nCbH specifying the height of the current coding block,
– a two-dimensional (nCbW)x(nCbH) array edgeFlags,
– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbW)x(nCbH) array edgeFlags.

The number of coding subblock in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

– If CuPredMode[xCb][yCb] = = MODE_INTRA, numSbX and numSbY are both set equal to 1.
– Otherwise, numSbX and numSbY are set equal to NumSbX[xCb][yCb] and NumSbY[xCb][yCb], respectively.

Depending on the value of edgeType the following applies:

– If edgeType is equal to EDGE_VER and numSbX is greater than 1, the following applies for i = 1..min((nCbW / 8) − 1, numSbX − 1), k = 0..nCbH − 1:

$$edgeFlags[i * max(8, nCbW / numSbX)][k] = 1 \quad (8\text{-}423)$$

– Otherwise if edgeType is equal to EDGE_HOR and numSbY is greater than 1, the following applies for j = 1..min((nCbH / 8) − 1, numSbY − 1), k = 0..nCbW − 1:

$$edgeFlags[k][j * max(8, nCbH / numSbY)] = 1 \quad (8\text{-}424)$$

8.5.2.5 Derivation process of boundary filtering strength

Inputs to this process are:

– a picture sample array recPicture,
– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
– a variable nCbW specifying the width of the current coding block,

– a variable nCbH specifying the height of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– a two-dimensional (nCbW)x(nCbH) array edgeFlags.

Output of this process is a two-dimensional (nCbW)x(nCbH) array bS specifying the boundary filtering strength.

The variables xD_i, yD_j, xN and yN are derived as follows:

– If edgeType is equal to EDGE_VER, xD_i is set equal to (i << 3), yD_j is set equal to (j << 2), xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to (nCbH / 4) − 1.

– Otherwise (edgeType is equal to EDGE_HOR), xD_i is set equal to (i << 2), yD_j is set equal to (j << 3), xN is set equal to (nCbW / 4) − 1 and yN is set equal to Max(0, (nCbH / 8) − 1).

For xD_i with i = 0..xN and yD_j with j = 0..yN, the following applies:

– If edgeFlags[xD_i][yD_j] is equal to 0, the variable bS[xD_i][yD_j] is set equal to 0.

– Otherwise, the following applies:

– The sample values p_0 and q_0 are derived as follows:

– If edgeType is equal to EDGE_VER, p_0 is set equal to recPicture [xCb + xD_i − 1][yCb + yD_j] and q_0 is set equal to recPicture [xCb + xD_i][yCb + yD_j].

– Otherwise (edgeType is equal to EDGE_HOR), p_0 is set equal to recPicture [xCb + xD_i][yCb + yD_j − 1] and q_0 is set equal to recPicture [xCb + xD_i][yCb + yD_j].

– The variable bS[xD_i][yD_j] is derived as follows:

– If the sample p_0 or q_0 is in the coding block of a coding unit coded with intra prediction mode, bS[xD_i][yD_j] is set equal to 2.

– Otherwise, if the block edge is also a transform block edge and the sample p_0 or q_0 is in a transform block which contains one or more non-zero transform coefficient levels, bS[xD_i][yD_j] is set equal to 1.

– Otherwise, if one or more of the following conditions are true, bS[xD_i][yD_j] is set equal to 1:

– For the prediction of the coding subblock containing the sample p_0 different reference pictures or a different number of motion vectors are used than for the prediction of the coding subblock containing the sample q_0.

NOTE 1 – The determination of whether the reference pictures used for the two coding sublocks are the same or different is based only on which pictures are referenced, without regard to whether a prediction is formed using an index into reference picture list 0 or an index into reference picture list 1, and also without regard to whether the index position within a reference picture list is different.

NOTE 2 – The number of motion vectors that are used for the prediction of a coding subblock with top-left sample covering (xSb, ySb), is equal to PredFlagL0[xSb][ySb] + PredFlagL1[xSb][ySb].

– One motion vector is used to predict the coding subblock containing the sample p_0 and one motion vector is used to predict the coding subblock containing the sample q_0, and the absolute difference between the horizontal or vertical component of the motion vectors used is greater than or equal to 4 in units of quarter luma samples.

– Two motion vectors and two different reference pictures are used to predict the coding subblock containing the sample p_0, two motion vectors for the same two reference pictures are used to predict the coding subblock containing the sample q_0 and the absolute difference between the horizontal or vertical component of the two motion vectors used in the prediction of the two coding subblocks for the same reference picture is greater than or equal to 4 in units of quarter luma samples.

– Two motion vectors for the same reference picture are used to predict the coding subblock containing the sample p_0, two motion vectors for the same reference picture are used to predict the coding subblock containing the sample q_0 and both of the following conditions are true:

– The absolute difference between the horizontal or vertical component of list 0 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in quarter luma samples, or the absolute difference between the horizontal or vertical component of the

list 1 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in units of quarter luma samples.

– The absolute difference between the horizontal or vertical component of list 0 motion vector used in the prediction of the coding subblock containing the sample p_0 and the list 1 motion vector used in the prediction of the coding subblock containing the sample q_0 is greater than or equal to 4 in units of quarter luma samples, or the absolute difference between the horizontal or vertical component of the list 1 motion vector used in the prediction of the coding subblock containing the sample p_0 and list 0 motion vector used in the prediction of the coding subblock containing the sample q_0 is greater than or equal to 4 in units of quarter luma samples.

– Otherwise, the variable bS[xD_i][yD_j] is set equal to 0.

8.5.2.6 Edge filtering process

8.5.2.6.1 Vertical edge filtering process

Inputs to this process are:

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$,

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

– a variable nCbW specifying the width of the current coding block,

– a variable nCbH specifying the height of the current coding block.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

When treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to (nCbH / 4) − 1.
2. For xD_k equal to k << 3 with k = 0..nN and yD_m equal to m << 2 with m = 0..yN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
     a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER,the boundary filtering strength bS[xD_k][yD_m], and the bit depth bD set equal to $BitDepth_Y$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.
     b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified luma picture sample array $recPicture_L$ as output.

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE the filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to Max(0, (nCbH / 8) − 1).

2. The variable edgeSpacing is set equal to 8 / SubWidthC.
3. The variable edgeSections is set equal to yN * (2 / SubHeightC).
4. For xD_k equal to k * edgeSpacing with k = 0..xN and yD_m equal to m << 2 with m = 0..edgeSections, the following applies:
   – When bS[xD_k * SubWidthC][yD_m * SubHeightC] is equal to 2 and (((xCb / SubWidthC + xD_k) >> 3) << 3) is equal to xCb / SubWidthC + xD_k, the following ordered steps apply:
     a. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER and a variable cQpPicOffset set equal to pps_cb_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
     b. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER and a variable cQpPicOffset set equal to pps_cr_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

When treeType is equal to DUAL_TREE_CHROMA, the filtering process for edges in the two chroma coding blocks of the current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to (nCbH / 4) − 1.
2. For xD_k equal to k << 3 with k = 0..xN and yD_m equal to m << 2 with m = 0..yN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
     a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the location of the chroma block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the boundary filtering strength bS[xD_k][yD_m], and the bit depth bD set equal to $BitDepth_C$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.
     b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
     c. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

8.5.2.6.2 Horizontal edge filtering process

Inputs to this process are:

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,
– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$,
– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$,
– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
– a variable nCbW specifying the width of the current coding block,

– a variable nCbH specifying the height of the current coding block.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

When treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable yN is set equal to Max(0, (nCbH / 8) − 1) and xN is set equal to (nCbW / 4) − 1.
2. For yD_m equal to m << 3 with m = 0..yN and xD_k equal to k << 2 with k = 0..xN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
      a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the boundary filtering strength bS[xD_k][yD_m], and the bit depth bD set equal to $BitDepth_Y$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.
      b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the decisions dEp, dEp and dEq, and the variable t_C as inputs, and the modified luma picture sample array $recPicture_L$ as output.

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE the filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to Max(0, (nCbH / 8) − 1).
2. The variable edgeSpacing is set equal to 8 / SubHeightC.
3. The variable edgeSections is set equal to xN * (2 / SubWidthC).
4. For yD_m equal to m * edgeSpacing with m = 0..yN and xD_k equal to k << 2 with k = 0..edgeSections, the following applies:
   – When bS[xD_k * SubWidthC][yD_m * SubHeightC] is equal to 2 and (((yCb / SubHeightC + yD_m) >> 3) << 3) is equal to yCb / SubHeightC + yD_m, the following ordered steps apply:
      a. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR and a variable cQpPicOffset set equal to pps_cb_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
      b. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR and a variable cQpPicOffset set equal to pps_cr_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

When treeType is equal to DUAL_TREE_CHROMA, the filtering process for edges in the two chroma coding blocks of the current coding unit consists of the following ordered steps:

1. The variable yN is set equal to Max(0, (nCbH / 8) − 1) and xN is set equal to (nCbW / 4) − 1.
2. For yD_m equal to m << 3 with m = 0..yN and xD_k equal to k << 2 with k = 0..xN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
      a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the location of the chroma block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the boundary filtering strength bS[xD_k][yD_m], and

the bit depth bD set equal to $BitDepth_C$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.

b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.

c. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

8.5.2.6.3 Decision process for block edges

Inputs to this process are:

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– a picture sample array recPicture,

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

– a location (xBl, yBl) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– a variable bS specifying the boundary filtering strength,

– a variable bD specifying the bit depth of the current component.

Outputs of this process are:

– the variables dE, dEp and dEq containing decisions,

– the variable t_C.

If edgeType is equal to EDGE_VER, the sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0 and 3 are derived as follows:

$$q_{i,k} = recPicture_L[\ xCb + xBl + i\ ][\ yCb + yBl + k\ ] \quad (8\text{-}429)$$

$$p_{i,k} = recPicture_L[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] \quad (8\text{-}430)$$

Otherwise (edgeType is equal to EDGE_HOR), the sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0 and 3 are derived as follows:

$$q_{i,k} = recPicture[\ xCb + xBl + k\ ][\ yCb + yBl + i\ ] \quad (8\text{-}431)$$

$$p_{i,k} = recPicture[\ xCb + xBl + k\ ][\ yCb + yBl - i - 1\ ] \quad (8\text{-}432)$$

The variable qpOffset is derived as follows:

– If sps_ladf_enabled_flag is equal to 1 and treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the following applies:

  – The variable lumaLevel of the reconstructed luma level is derived as follow:

$$lumaLevel = (\ (\ p0,0 + p0,3 + q0,0 + q0,3\ ) >> 2\ ), \quad (8\text{-}433)$$

  – The variable qpOffset is set equal to sps_ladf_lowest_interval_qp_offset and modified as follows:

```
for( i = 0; i < sps_num_ladf_intervals_minus2 + 1; i++ ) {
    if( lumaLevel > SpsLadfIntervalLowerBound[ i + 1 ] )
```

```
            qpOffset = sps_ladf_qp_offset[ i ]                                    (8-434)
        else
            break
    }
```

– Otherwise (treeType is equal to DUAL_TREE_CHROMA), qpOffset is set equal to 0.

The variables Qp_Q and Qp_P are derived as follows:

– If treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

– Otherwise (treeType is equal to DUAL_TREE_CHROMA), Qp_Q and Qp_P are set equal to the Qp_C values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

The variable qP is derived as follows:

$$qP = ((Qp_Q + Qp_P + 1) >> 1) + qpOffset \quad (8\text{-}435)$$

The value of the variable β′ is determined as specified in Table 8-11 based on the quantization parameter Q derived as follows:

$$Q = Clip3(0, 51, qP + (slice\_beta\_offset\_div2 << 1)) \quad (8\text{-}436)$$

where slice_beta_offset_div2 is the value of the syntax element slice_beta_offset_div2 for the slice that contains sample $q_{0,0}$.

The variable β is derived as follows:

$$\beta = \beta' * (1 << (bD - 8)) \quad (8\text{-}437)$$

The value of the variable t_C' is determined as specified in Table 8-11 based on the quantization parameter Q derived as follows:

$$Q = Clip3(0, 53, qP + 2 * (bS - 1) + (slice\_tc\_offset\_div2 << 1)) \quad (8\text{-}438)$$

where slice_tc_offset_div2 is the value of the syntax element slice_tc_offset_div2 for the slice that contains sample $q_{0,0}$.

The variable t_C is derived as follows:

$$t_C = t_C' * (1 << (bD - 8)) \quad (8\text{-}439)$$

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE_VER, the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

$$dp0 = Abs(p_{2,0} - 2 * p_{1,0} + p_{0,0}) \quad (8\text{-}440)$$

$$dp3 = Abs(p_{2,3} - 2 * p_{1,3} + p_{0,3}) \quad (8\text{-}441)$$

$$dq0 = Abs(q_{2,0} - 2 * q_{1,0} + q_{0,0}) \quad (8\text{-}442)$$

$$dq3 = Abs(q_{2,3} - 2 * q_{1,3} + q_{0,3}) \quad (8\text{-}443)$$

$$dpq0 = dp0 + dq0 \quad (8\text{-}444)$$

$$dpq3 = dp3 + dq3 \quad (8\text{-}445)$$

$$dp = dp0 + dp3 \quad (8\text{-}446)$$

$$dq = dq0 + dq3 \quad (8\text{-}447)$$

$$d = dpq0 + dpq3 \quad (8\text{-}448)$$

2. The variables dE, dEp and dEq are set equal to 0.

3. When d is less than β, the following ordered steps apply:
   a. The variable dpq is set equal to 2 * dpq0.
   b. For the sample location (xCb + xBl, yCb + yBl), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,0}$, $p_{3,0}$, $q_{0,0}$, and $q_{3,0}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam0.
   c. The variable dpq is set equal to 2 * dpq3.
   d. For the sample location (xCb + xBl, yCb + yBl + 3), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,3}$, $p_{3,3}$, $q_{0,3}$, and $q_{3,3}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam3.
   e. The variable dE is set equal to 1.
   f. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
   g. When dp is less than (β + (β >> 1)) >> 3, the variable dEp is set equal to 1.
   h. When dq is less than (β + (β >> 1)) >> 3, the variable dEq is set equal to 1.

– Otherwise (edgeType is equal to EDGE_HOR), the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

$$dp0 = Abs(p_{2,0} - 2 * p_{1,0} + p_{0,0}) \quad (8\text{-}449)$$

$$dp3 = Abs(p_{2,3} - 2 * p_{1,3} + p_{0,3}) \quad (8\text{-}450)$$

$$dq0 = Abs(q_{2,0} - 2 * q_{1,0} + q_{0,0}) \quad (8\text{-}451)$$

$$dq3 = Abs(q_{2,3} - 2 * q_{1,3} + q_{0,3}) \quad (8\text{-}452)$$

$$dpq0 = dp0 + dq0 \quad (8\text{-}453)$$

$$dpq3 = dp3 + dq3 \quad (8\text{-}454)$$

$$dp = dp0 + dp3 \quad (8\text{-}455)$$

$$dq = dq0 + dq3 \quad (8\text{-}456)$$

$$d = dpq0 + dpq3 \quad (8\text{-}457)$$

2. The variables dE, dEp and dEq are set equal to 0.
3. When d is less than β, the following ordered steps apply:
   a. The variable dpq is set equal to 2 * dpq0.
   b. For the sample location (xCb + xBl, yCb + yBl), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,0}$, $p_{3,0}$, $q_{0,0}$ and $q_{3,0}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam0.
   c. The variable dpq is set equal to 2 * dpq3.
   d. For the sample location (xCb + xBl + 3, yCb + yBl), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,3}$, $p_{3,3}$, $q_{0,3}$ and $q_{3,3}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam3.
   e. The variable dE is set equal to 1.
   f. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
   g. When dp is less than (β + (β >> 1)) >> 3, the variable dEp is set equal to 1.
   h. When dq is less than (β + (β >> 1)) >> 3, the variable dEq is set equal to 1.

Table 8-11 – Derivation of threshold variables β′ and t_C′ from input Q

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
β′	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
t_C′	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Q	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
β′	7	8	9	10	11	12	13	14	15	16	17	18	20	22	24	26	28
t_C′	0	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3
Q	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
β′	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
t_C′	3	4	4	4	5	5	6	6	7	8	9	10	11	13	14	16	18
Q	51	52	53														
β′	64	-	-														
t_C′	20	22	25	28	31	35	39	44	50	56	63	70	79	88	99		

8.5.2.6.4 Filtering process for block edges

Inputs to this process are:

– a picture sample array recPicture,

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

– a location (xBl, yBl) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– the variables dE, dEp and dEq containing decisions,

– the variable t_C.

Output of this process is the modified picture sample array recPicture.

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE_VER, the following ordered steps apply:

1. The sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0..3 are derived as follows:

$$q_{i,k} = recPicture_L[\ xCb + xBl + i\ ][\ yCb + yBl + k\ ] \quad (8\text{-}458)$$

$$p_{i,k} = recPicture_L[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] \quad (8\text{-}459)$$

2. When dE is not equal to 0, for each sample location (xCb + xBl, yCb + yBl + k), k = 0..3, the following ordered steps apply:

   a. The filtering process for a sample as specified in clause 8.5.2.6.7 is invoked with the sample values $p_{i,k}$, $q_{i,k}$ with i = 0..3, the locations (xP_i, yP_i) set equal to (xCb + xBl − i − 1, yCb + yBl + k) and (xQ_i, yQ_i) set equal to (xCb + xBl + i, yCb + yBl + k) with i = 0..2, the decision dE, the variables dEp and dEq and the variable t_C as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values p_i' and q_j' as outputs.

   b. When nDp is greater than 0, the filtered sample values p_i' with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] = p_i' \quad (8\text{-}460)$$

   c. When nDq is greater than 0, the filtered sample values q_j' with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[\ xCb + xBl + j\ ][\ yCb + yBl + k\ ] = q_j' \quad (8\text{-}461)$$

– Otherwise (edgeType is equal to EDGE_HOR), the following ordered steps apply:

1. The sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0..3 are derived as follows:

$$q_{i,k} = recPicture_L[\ xCb + xBl + k\ ][\ yCb + yBl + i\ ] \quad (8\text{-}462)$$

$$p_{i,k} = recPicture_L[\ xCb + xBl + k\ ][\ yCb + yBl - i - 1\ ] \quad (8\text{-}463)$$

2. When dE is not equal to 0, for each sample location (xCb + xBl + k, yCb + yBl), k = 0..3, the following ordered steps apply:

   a. The filtering process for a sample as specified in clause 8.5.2.6.7 is invoked with the sample values $p_{i,k}$, $q_{i,k}$ with i = 0..3, the locations (xP_i, yP_i) set equal to (xCb + xBl + k, yCb + yBl − i − 1) and (xQ_i, yQ_i) set equal to (xCb + xBl + k, yCb + yBl + i) with i = 0..2, the decision dE, the variables dEp and dEq, and the variable t_C as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values p_i' and q_j' as outputs.

   b. When nDp is greater than 0, the filtered sample values p_i' with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[\ xCb + xBl + k\ ][\ yCb + yBl - i - 1\ ] = p_i' \quad (8\text{-}464)$$

   c. When nDq is greater than 0, the filtered sample values q_j' with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[\ xCb + xBl + k\ ][\ yCb + yBl + j\ ] = q_j' \quad (8\text{-}465)$$

8.5.2.6.5 Filtering process for chroma block edges

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

– a chroma picture sample array s′,

– a chroma location (xCb, yCb) specifying the top-left sample of the current chroma coding block relative to the top-left chroma sample of the current picture,

– a chroma location (xBl, yBl) specifying the top-left sample of the current chroma block relative to the top-left sample of the current chroma coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– a variable cQpPicOffset specifying the picture-level chroma quantization parameter offset.

Output of this process is the modified chroma picture sample array s′.

If edgeType is equal to EDGE_VER, the values p_i and q_i with i = 0..1 and k = 0..3 are derived as follows:

$$q_{i,k} = s'[\ xCb + xBl + i\ ][\ yCb + yBl + k\ ] \quad (8\text{-}466)$$

$$p_{i,k} = s'[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] \quad (8\text{-}467)$$

Otherwise (edgeType is equal to EDGE_HOR), the sample values p_i and q_i with i = 0..1 and k = 0..3 are derived as follows:

$$q_{i,k} = s'[\ xCb + xBl + k\ ][\ yCb + yBl + i\ ] \quad (8\text{-}468)$$

$$p_{i,k} = s'[\ xCb + xBl + k\ ][\ yCb + yBl - i - 1\ ] \quad (8\text{-}469)$$

The variables Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

If ChromaArrayType is equal to 1, the variable Qp_C is determined as specified in [Ed. (BB): Add reference when table is added in 8.4.1] based on the index qPi derived as follows:

$$qPi = (\ (\ Qp_Q + Qp_P + 1\ )\ >>\ 1\ ) + cQpPicOffset \quad (8\text{-}470)$$

Otherwise (ChromaArrayType is greater than 1), the variable Qp_C is set equal to Min(qPi, 51).

NOTE – The variable cQpPicOffset provides an adjustment for the value of pps_cb_qp_offset or pps_cr_qp_offset, according to whether the filtered chroma component is the Cb or Cr component. However, to avoid the need to vary the amount of the adjustment within the picture, the filtering process does not include an adjustment for the value of slice_cb_qp_offset or slice_cr_qp_offset, nor (when chroma_qp_offset_list_enabled_flag is equal to 1) for the value of $CuQpOffset_{Cb}$ or $CuQpOffset_{Cr}$.

The value of the variable t_C' is determined as specified in Table 8-11 based on the chroma quantization parameter Q derived as follows:

$$Q = Clip3(0, 53, Qp_C + 2 + (slice\_tc\_offset\_div2 << 1)) \quad (8\text{-}471)$$

where slice_tc_offset_div2 is the value of the syntax element slice_tc_offset_div2 for the slice that contains sample $q_{0,0}$.

The variable t_C is derived as follows:

$$t_C = t_C' * (1 << (BitDepth_C - 8)) \quad (8\text{-}472)$$

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE_VER, for each sample location (xCb + xBl, yCb + yBl + k), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.5.2.6.8 is invoked with the sample values $p_{i,k}$, $q_{i,k}$, with i = 0..1, the locations (xCb + xBl − 1, yCb + yBl + k) and (xCb + xBl, yCb + yBl + k) and the variable t_C as inputs, and the filtered sample values p_0' and q_0' as outputs.
2. The filtered sample values p_0' and q_0' replace the corresponding samples inside the sample array s′ as follows:

$$s'[xCb + xBl][yCb + yBl + k] = q_0' \quad (8\text{-}473)$$

$$s'[xCb + xBl - 1][yCb + yBl + k] = p_0' \quad (8\text{-}474)$$

– Otherwise (edgeType is equal to EDGE_HOR), for each sample location (xCb + xBl + k, yCb + yBl), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.5.2.6.8 is invoked with the sample values $p_{i,k}$, $q_{i,k}$, with i = 0..1, the locations (xCb + xBl + k, yCb + yBl − 1) and (xCb + xBl + k, yCb + yBl), and the variable t_C as inputs, and the filtered sample values p_0' and q_0' as outputs.
2. The filtered sample values p_0' and q_0' replace the corresponding samples inside the sample array s′ as follows:

$$s'[xCb + xBl + k][yCb + yBl] = q_0' \quad (8\text{-}475)$$

$$s'[xCb + xBl + k][yCb + yBl - 1] = p_0' \quad (8\text{-}476)$$

8.5.2.6.6 Decision process for a sample

Inputs to this process are:

– the sample values p_0, p_3, q_0 and q_3,

– the variables dpq, β and t_C.

Output of this process is the variable dSam containing a decision.

The variable dSam is specified as follows:

– If dpq is less than (β >> 2), Abs($p_3 - p_0$) + Abs($q_0 - q_3$) is less than (β >> 3) and Abs($p_0 - q_0$) is less than (5 * t_C + 1) >> 1, dSam is set equal to 1.

– Otherwise, dSam is set equal to 0.

8.5.2.6.7 Filtering process for a sample

Inputs to this process are:

– the sample values p_i and q_i with i = 0..3,

– the locations of p_i and q_i, (xP_i, yP_i) and (xQ_i, yQ_i) with i = 0..2,

– a variable dE,

– the variables dEp and dEq containing decisions to filter samples p1 and q1, respectively,

– a variable t_C.

Outputs of this process are:

– the number of filtered samples nDp and nDq,

– the filtered sample values p_i' and q_j' with i = 0..nDp − 1, j = 0..nDq − 1.

Depending on the value of dE, the following applies:

– If the variable dE is equal to 2, nDp and nDq are both set equal to 3 and the following strong filtering applies:

$$p_0' = Clip3(p_0 - 2 * t_C, p_0 + 2 * t_C, (p_2 + 2 * p_1 + 2 * p_0 + 2 * q_0 + q_1 + 4) >> 3) \quad (8\text{-}477)$$

$$p_1' = Clip3(p_1 - 2 * t_C, p_1 + 2 * t_C, (p_2 + p_1 + p_0 + q_0 + 2) >> 2) \quad (8\text{-}478)$$

$$p_2' = Clip3(p_2 - 2 * t_C, p_2 + 2*t_C, (2 * p_3 + 3 * p_2 + p_1 + p_0 + q_0 + 4) >> 3) \quad (8\text{-}479)$$

$$q_0' = Clip3(q_0 - 2 * t_C, q_0 + 2 * t_C, (p_1 + 2 * p_0 + 2 * q_0 + 2 * q_1 + q_2 + 4) >> 3) \quad (8\text{-}480)$$

$$q_1' = Clip3(q_1 - 2 * t_C, q_1 + 2 * t_C, (p_0 + q_0 + q_1 + q_2 + 2) >> 2) \quad (8\text{-}481)$$

$$q_2'= Clip3(q_2 - 2 * t_C, q_2 + 2 * t_C, (p_0 + q_0 + q_1 + 3 * q_2 + 2 * q_3 + 4) >> 3) \quad (8\text{-}482)$$

– Otherwise, nDp and nDq are set both equal to 0 and the following weak filtering applies:

  – The following applies:

$$\Delta = (9 * (q_0 - p_0) - 3 * (q_1 - p_1) + 8) >> 4 \quad (8\text{-}483)$$

  – When Abs(Δ) is less than $t_C * 10$, the following ordered steps apply:

    1. The filtered sample values p_0' and q_0' are specified as follows:

$$\Delta = Clip3(-t_C, t_C, \Delta) \quad (8\text{-}484)$$

$$p_0' = Clip1_Y(p_0 + \Delta) \quad (8\text{-}485)$$

$$q_0' = Clip1_Y(q_0 - \Delta) \quad (8\text{-}486)$$

    2. When dEp is equal to 1, the filtered sample value p_1' is specified as follows:

$$\Delta p = Clip3(-(t_C >> 1), t_C >> 1, (((p_2 + p_0 + 1) >> 1) - p_1 + \Delta) >> 1) \quad (8\text{-}487)$$

$$p_1' = Clip1_Y(p_1 + \Delta p) \quad (8\text{-}488)$$

    3. When dEq is equal to 1, the filtered sample value q_1' is specified as follows:

$$\Delta q = Clip3(-(t_C >> 1), t_C >> 1, (((q_2 + q_0 + 1) >> 1) - q_1 - \Delta) >> 1) \quad (8\text{-}489)$$

$$q_1' = Clip1_Y(q_1 + \Delta q) \quad (8\text{-}490)$$

    4. nDp is set equal to dEp + 1 and nDq is set equal to dEq + 1.

When nDp is greater than 0 and one or more of the following conditions are true, nDp is set equal to 0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xP_0][yP_0] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample p_0 is equal to 1.

When nDq is greater than 0 and one or more of the following conditions are true, nDq is set equal to 0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xQ_0][yQ_0] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample q_0 is equal to 1.

8.5.2.6.8 Filtering process for a chroma sample

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

– the chroma sample values p_i and q_i with i = 0..1,

– the chroma locations of p_0 and q_0, (xP_0, yP_0) and (xQ_0, yQ_0),

– a variable t_C.

Outputs of this process are the filtered sample values p_0' and q_0'.

The filtered sample values p_0' and q_0' are derived as follows:

$$\Delta = \text{Clip3}(-t_C, t_C, ((((q_0 - p_0) << 2) + p_1 - q_1 + 4) >> 3)) \quad (8\text{-}491)$$

$$p_0' = \text{Clip1}_C(p_0 + \Delta) \quad (8\text{-}492)$$

$$q_0' = \text{Clip1}_C(q_0 - \Delta) \quad (8\text{-}493)$$

When one or more of the following conditions are true, the filtered sample value, p_0' is substituted by the corresponding input sample value p_0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xP_0 * SubWidthC][yP_0 * SubHeightC] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample p_0 is equal to 1.

When one or more of the following conditions are true, the filtered sample value, q_0' is substituted by the corresponding input sample value q_0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xQ_0 * SubWidthC][yQ_0 * SubHeightC] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample q_0 is equal to 1.

8.5.3 Sample adaptive offset process

8.5.3.1 General

Inputs to this process are the reconstructed picture sample array prior to sample adaptive offset recPicture_L and, when ChromaArrayType is not equal to 0, the arrays recPicture_{Cb} and recPicture_{Cr}.

Outputs of this process are the modified reconstructed picture sample array after sample adaptive offset saoPicture_L and, when ChromaArrayType is not equal to 0, the arrays saoPicture_{Cb} and saoPicture_{Cr}.

This process is performed on a CTB basis after the completion of the deblocking filter process for the decoded picture.

The sample values in the modified reconstructed picture sample array saoPicture_L and, when ChromaArrayType is not equal to 0, the arrays saoPicture_{Cb} and saoPicture_{Cr} are initially set equal to the sample values in the reconstructed picture sample array recPicture_L and, when ChromaArrayType is not equal to 0, the arrays recPicture_{Cb} and recPicture_{Cr}, respectively.

For every CTU with CTB location (rx, ry), where rx = 0..PicWidthInCtbsY − 1 and ry = 0..PicHeightInCtbsY − 1, the following applies:

– When slice_sao_luma_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.5.3.2 is invoked with recPicture set equal to recPicture_L, cIdx set equal to 0, (rx, ry), and both nCtbSw and nCtbSh set equal to CtbSizeY as inputs, and the modified luma picture sample array saoPicture_L as output.

– When ChromaArrayType is not equal to 0 and slice_sao_chroma_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.5.3.2 is invoked with recPicture set equal to recPicture_{Cb}, cIdx set equal to 1, (rx, ry), nCtbSw set equal to (1 << CtbLog2SizeY) / SubWidthC and nCtbSh set equal to (1 << CtbLog2SizeY) / SubHeightC as inputs, and the modified chroma picture sample array saoPicture_{Cb} as output.

– When ChromaArrayType is not equal to 0 and slice_sao_chroma_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.5.3.2 is invoked with recPicture set equal to recPicture_{Cr}, cIdx set equal to 2, (rx, ry), nCtbSw set equal to (1 << CtbLog2SizeY) / SubWidthC and nCtbSh set equal to (1 << CtbLog2SizeY) / SubHeightC as inputs, and the modified chroma picture sample array saoPicture_{Cr} as output.

8.5.3.2 CTB modification process

Inputs to this process are:

– the picture sample array recPicture for the colour component cIdx,

– a variable cIdx specifying the colour component index,

– a pair of variables (rx, ry) specifying the CTB location,

– the CTB width nCtbSw and height nCtbSh.

Output of this process is a modified picture sample array saoPicture for the colour component cIdx.

The variable bitDepth is derived as follows:

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

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

The location (xCtb, yCtb), specifying the top-left sample of the current CTB for the colour component cIdx relative to the top-left sample of the current picture component cIdx, is derived as follows:

$$(xCtb, yCtb) = (rx * nCtbSw, ry * nCtbSh) \quad (8\text{-}494)$$

The sample locations inside the current CTB are derived as follows:

$$(xS_i, yS_j) = (xCtb + i, yCtb + j) \quad (8\text{-}495)$$

$$(xY_i, yY_j) = (cIdx == 0) ? (xS_i, yS_j) : (xS_i * SubWidthC, yS_j * SubHeightC) \quad (8\text{-}496)$$

For all sample locations (xS_i, yS_j) and (xY_i, yY_j) with i = 0..nCtbSw − 1 and j = 0..nCtbSh − 1, depending on the values of pcm_loop_filter_disabled_flag, pcm_flag[xY_i][yY_j] and cu_transquant_bypass_flag of the coding unit which includes the coding block covering recPicture[xS_i][yS_j], the following applies:

– If one or more of the following conditions are true, saoPicture[xS_i][yS_j] is not modified:

  – pcm_loop_filter_disabled_flag and pcm_flag[xY_i][yY_j] are both equal to 1.

  – cu_transquant_bypass_flag is equal to 1.

  – SaoTypeIdx[cIdx][rx][ry] is equal to 0.

[Ed. (BB): Modify highlighted sections prending on future decision on PCM transform/quantizaion bypass.]

– Otherwise, if SaoTypeIdx[cIdx][rx][ry] is equal to 2, the following ordered steps apply:

1. The values of hPos[k] and vPos[k] for k = 0..1 are specified in Table 8-12 based on SaoEoClass[cIdx][rx][ry].

2. The variable edgeIdx is derived as follows:

  – The modified sample locations (xS_{ik}', yS_{jk}') and (xY_{ik}', yY_{jk}') are derived as follows:

$$(xS_{ik}', yS_{jk}') = (xS_i + hPos[k], yS_j + vPos[k]) \quad (8\text{-}497)$$

$$(xY_{ik}', yY_{jk}') = (cIdx == 0) ? (xS_{ik}', yS_{jk}') : (xS_{ik}' * SubWidthC, yS_{jk}' * SubHeightC) \quad (8\text{-}498)$$

  – If one or more of the following conditions for all sample locations (xS_{ik}', yS_{jk}') and (xY_{ik}', yY_{jk}') with k = 0..1 are true, edgeIdx is set equal to 0:

    – The sample at location (xS_{ik}', yS_{jk}') is outside the picture boundaries.

    – The sample at location (xS_{ik}', yS_{jk}') belongs to a different slice and one of the following two conditions is true:

      – MinTbAddrZs[xY_{ik}' >> MinTbLog2SizeY][yY_{jk}' >> MinTbLog2SizeY] is less than MinTbAddrZs[xY_i >> MinTbLog2SizeY][yY_j >> MinTbLog2SizeY] and slice_loop_filter_across_slices_enabled_flag in the slice which the sample recPicture[xS_i][yS_j] belongs to is equal to 0.

      – MinTbAddrZs[xY_i >> MinTbLog2SizeY][yY_j >> MinTbLog2SizeY] is less than MinTbAddrZs[xY_{ik}' >> MinTbLog2SizeY][yY_{jk}' >> MinTbLog2SizeY] and slice_loop_filter_across_slices_enabled_flag in the slice which the sample recPicture[xS_{ik}'][yS_{jk}'] belongs to is equal to 0.

– loop_filter_across_tiles_enabled_flag is equal to 0 and the sample at location (xS_{ik}', yS_{jk}') belongs to a different tile.

[Ed. (BB): Modify highlighted sections when tiles without slices are incorporated]

– Otherwise, edgeIdx is derived as follows:

– The following applies:

$$edgeIdx = 2 + Sign(recPicture[xS_i][yS_j] - recPicture[xS_i + hPos[0]][yS_j + vPos[0]]) + Sign(recPicture[xS_i][yS_j] - recPicture[xS_i + hPos[1]][yS_j + vPos[1]]) \quad (8\text{-}499)$$

– When edgeIdx is equal to 0, 1, or 2, edgeIdx is modified as follows:

$$edgeIdx = (edgeIdx == 2) ? 0 : (edgeIdx + 1) \quad (8\text{-}500)$$

3. The modified picture sample array saoPicture[xS_i][yS_j] is derived as follows:

$$saoPicture[xS_i][yS_j] = Clip3(0, (1 << bitDepth) - 1, recPicture[xS_i][yS_j] + SaoOffsetVal[cIdx][rx][ry][edgeIdx]) \quad (8\text{-}501)$$

– Otherwise (SaoTypeIdx[cIdx][rx][ry] is equal to 1), the following ordered steps apply:

1. The variable bandShift is set equal to bitDepth − 5.
2. The variable saoLeftClass is set equal to sao_band_position[cIdx][rx][ry].
3. The list bandTable is defined with 32 elements and all elements are initially set equal to 0. Then, four of its elements (indicating the starting position of bands for explicit offsets) are modified as follows:

```
for( k = 0; k < 4; k++ )
    bandTable[ ( k + saoLeftClass ) & 31 ] = k + 1
```
(8-502)

4. The variable bandIdx is set equal to bandTable[recPicture[xS_i][yS_j] >> bandShift].
5. The modified picture sample array saoPicture[xS_i][yS_j] is derived as follows:

$$saoPicture[xS_i][yS_j] = Clip3(0, (1 << bitDepth) - 1, recPicture[xS_i][yS_j] + SaoOffsetVal[cIdx][rx][ry][bandIdx]) \quad (8\text{-}503)$$

Table 8-12 – Specification of hPos and vPos according to the sample adaptive offset class

SaoEoClass[cIdx][rx][ry]	0	1	2	3
hPos[0]	−1	0	−1	1
hPos[1]	1	0	1	−1
vPos[0]	0	−1	−1	−1
vPos[1]	0	1	1	1

8.5.4 Adaptive loop filter process

8.5.4.1 General

Inputs of this process are the reconstructed picture sample arrays prior to adaptive loop filter $recPicture_L$, $recPicture_{Cb}$ and $recPicture_{Cr}$.

Outputs of this process are the modified reconstructed picture sample arrays after adaptive loop filter $alfPicture_L$, $alfPicture_{Cb}$ and $alfPicture_{Cr}$.

The sample values in the modified reconstructed picture sample arrays after adaptive loop filter $alfPicture_L$, $alfPicture_{Cb}$ and $alfPicture_{Cr}$ are initially set equal to the sample values in the reconstructed picture sample arrays prior to adaptive loop filter $recPicture_L$, $recPicture_{Cb}$ and $recPicture_{Cr}$, respectively.

When slice_alf_enabled_flag is equal to 1, for every coding tree unit with luma coding tree block location (rx, ry), where rx = 0..PicWidthInCtbs − 1 and ry = 0..PicHeightInCtbs − 1, the following applies:

– When alf_ctb_flag[0][rx][ry] is equal to 1, the coding tree block filtering process for luma samples as specified in clause 8.5.4.2 is invoked with $recPicture_L$, $alfPicture_L$, and the luma coding tree block location (xCtb, yCtb) set equal to (rx << CtbLog2SizeY, ry << CtbLog2SizeY) as inputs, and the output is the modified filtered picture $alfPicture_L$.

– When alf_ctb_flag[1][rx][ry] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.5.4.4 is invoked with recPicture set equal to $recPicture_{Cb}$, alfPicture set equal to $alfPicture_{Cb}$, and the chroma coding tree block location (xCtbC, yCtbC) set equal to (rx << (CtbLog2SizeY − 1), ry << (CtbLog2SizeY − 1)) as inputs, and the output is the modified filtered picture $alfPicture_{Cb}$.

– When alf_ctb_flag[2][rx][ry] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.5.4.4 is invoked with recPicture set equal to $recPicture_{Cr}$, alfPicture set equal to $alfPicture_{Cr}$, and the chroma coding tree block location (xCtbC, yCtbC) set equal to (rx << (CtbLog2SizeY − 1), ry << (CtbLog2SizeY − 1)) as inputs, and the output is the modified filtered picture $alfPicture_{Cr}$.

8.5.4.2 Coding tree block filtering process for luma samples

Inputs of this process are:

– a reconstructed luma picture sample array $recPicture_L$ prior to the adaptive loop filtering process,

– a filtered reconstructed luma picture sample array $alfPicture_L$,

– a luma location (xCtb, yCtb) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture.

Output of this process is the modified filtered reconstructed luma picture sample array $alfPicture_L$.

The derivation process for filter index clause 8.5.4.3 is invoked with the location (xCtb, yCtb) and the reconstructed luma picture sample array $recPicture_L$ as inputs, and filtIdx[x][y] and transposeIdx[x][y] with x, y = 0..CtbSizeY − 1 as outputs.

For the derivation of the filtered reconstructed luma samples $alfPicture_L$[x][y], each reconstructed luma sample inside the current luma coding tree block $recPicture_L$[x][y] is filtered as follows with x, y = 0..CtbSizeY − 1:

– The array of luma filter coefficients f[j] corresponding to the filter specified by filtIdx[x][y] is derived as follows with j = 0..12:

$$f[j] = AlfCoeff_L[filtIdx[x][y]][j] \quad (8\text{-}504)$$

– The luma filter coefficients filterCoeff are derived depending on transposeIdx[x][y] as follows:

  – If transposeIndex[x][y] = = 1,

$$filterCoeff[ ] = \{ f[9], f[4], f[10], f[8], f[1], f[5], f[11], f[7], f[3], f[0], f[2], f[6], f[12] \} \quad (8\text{-}505)$$

  – Otherwise, if transposeIndex[x][y] = = 2,

$$filterCoeff[ ] = \{ f[0], f[3], f[2], f[1], f[8], f[7], f[6], f[5], f[4], f[9], f[10], f[11], f[12] \} \quad (8\text{-}506)$$

  – Otherwise, if transposeIndex[x][y] = = 3,

$$filterCoeff[ ] = \{ f[9], f[8], f[10], f[4], f[3], f[7], f[11], f[5], f[1], f[0], f[2], f[6], f[12] \} \quad (8\text{-}507)$$

  – Otherwise,

$$filterCoeff[ ] = \{ f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10], f[11], f[12] \} \quad (8\text{-}508)$$

– The locations (h_x, v_y) for each of the corresponding luma samples (x, y) inside the given array recPicture of luma samples are derived as follows:

$$h_x = Clip3(0, pic\_width\_in\_luma\_samples − 1, xCtb + x) \quad (8\text{-}509)$$

$$v_y = Clip3(0, pic\_height\_in\_luma\_samples − 1, yCtb + y) \quad (8\text{-}510)$$

– The variable sum is derived as follows:

$$\begin{aligned}sum = {} & filterCoeff[\,0\,] * (recPicture_L[\,h_x, v_{y+3}\,] + recPicture_L[\,h_x, v_{y-3}\,]) + \\ & filterCoeff[\,1\,] * (recPicture_L[\,h_{x+1}, v_{y+2}\,] + recPicture_L[\,h_{x-1}, v_{y-2}\,]) + \\ & filterCoeff[\,2\,] * (recPicture_L[\,h_x, v_{y+2}\,] + recPicture_L[\,h_x, v_{y-2}\,]) + \\ & filterCoeff[\,3\,] * (recPicture_L[\,h_{x-1}, v_{y+2}\,] + recPicture_L[\,h_{x+1}, v_{y-2}\,]) + \\ & filterCoeff[\,4\,] * (recPicture_L[\,h_{x+2}, v_{y+1}\,] + recPicture_L[\,h_{x-2}, v_{y-1}\,]) + \\ & filterCoeff[\,5\,] * (recPicture_L[\,h_{x+1}, v_{y+1}\,] + recPicture_L[\,h_{x-1}, v_{y-1}\,]) + \\ & filterCoeff[\,6\,] * (recPicture_L[\,h_x, v_{y+1}\,] + recPicture_L[\,h_x, v_{y-1}\,]) + \\ & filterCoeff[\,7\,] * (recPicture_L[\,h_{x-1}, v_{y+1}\,] + recPicture_L[\,h_{x+1}, v_{y-1}\,]) + \\ & filterCoeff[\,8\,] * (recPicture_L[\,h_{x-2}, v_{y+1}\,] + recPicture_L[\,h_{x+2}, v_{y-1}\,]) + \\ & filterCoeff[\,9\,] * (recPicture_L[\,h_{x+3}, v_y\,] + recPicture_L[\,h_{x-3}, v_y\,]) + \\ & filterCoeff[\,10\,] * (recPicture_L[\,h_{x+2}, v_y\,] + recPicture_L[\,h_{x-2}, v_y\,]) + \\ & filterCoeff[\,11\,] * (recPicture_L[\,h_{x+1}, v_y\,] + recPicture_L[\,h_{x-1}, v_y\,]) + \\ & filterCoeff[\,12\,] * \; recPicture_L[\,h_x, v_y\,]\end{aligned} \tag{8-511}$$

$$sum = (sum + 64) >> 7 \tag{8-512}$$

– The modified filtered reconstructed luma picture sample $alfPicture_L[\,xCtb + x\,][\,yCtb + y\,]$ is derived as follows:

$$alfPicture_L[\,xCtb + x\,][\,yCtb + y\,] = Clip3(0, (1 << BitDepth_Y) - 1, sum) \tag{8-513}$$

8.5.4.3 Derivation process for ALF transpose and filter index for luma samples

Inputs of this process are:

– a luma location (xCtb, yCtb) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture,

– a reconstructed luma picture sample array $recPicture_L$ prior to the adaptive loop filtering process.

Outputs of this process are

– the classification filter index array filtIdx[x][y] with x, y = 0..CtbSizeY − 1,

– the transpose index array transposeIdx[x][y] with x, y = 0..CtbSizeY − 1.

The locations (h_x, v_y) for each of the corresponding luma samples (x, y) inside the given array recPicture of luma samples are derived as follows:

$$h_x = Clip3(0, pic\_width\_in\_luma\_samples - 1, x) \tag{8-514}$$

$$v_y = Clip3(0, pic\_height\_in\_luma\_samples - 1, y) \tag{8-515}$$

The classification filter index array filtIdx and the transpose index array transposeIdx are derived by the following ordered steps:

1. The variables filtH[x][y], filtV[x][y], filtD0[x][y] and filtD1[x][y] with x, y = − 2..CtbSizeY + 1 are derived as follows:

   – If both x and y are even numbers or both x and y are uneven numbers, the following applies:

$$filtH[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x-1}, v_{yCtb+y}\,] - recPicture[\,h_{xCtb+x+1}, v_{yCtb+y}\,]) \tag{8-516}$$

$$filtV[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x}, v_{yCtb+y-1}\,] - recPicture[\,h_{xCtb+x}, v_{yCtb+y+1}\,]) \tag{8-517}$$

$$filtD0[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x-1}, v_{yCtb+y-1}\,] - recPicture[\,h_{xCtb+x+1}, v_{yCtb+y+1}\,]) \tag{8-518}$$

$$filtD1[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x+1}, v_{yCtb+y-1}\,] - recPicture[\,h_{xCtb+x-1}, v_{yCtb+y+1}\,]) \tag{8-519}$$

   – Otherwise, filtH[x][y], filtV[x][y], filtD0[x][y] and filtD1[x][y] are set equal to 0.

2. The variables varTempH1[x][y], varTempV1[x][y], varTempD01[x][y], varTempD11[x][y] and varTemp[x][y] with x, y = 0..(CtbSizeY − 1) >> 2 are derived as follows:

$$sumH[x][y] = \Sigma_i\Sigma_j\ filtH[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}520)$$

$$sumV[x][y] = \Sigma_i\Sigma_j\ filtV[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}521)$$

$$sumD0[x][y] = \Sigma_i\Sigma_j\ filtD0[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}522)$$

$$sumD1[x][y] = \Sigma_i\Sigma_j\ filtD1[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}523)$$

$$sumOfHV[x][y] = sumH[x][y] + sumV[x][y] \quad (8\text{-}524)$$

3. The variables dir1[x][y], dir2[x][y] and dirS[x][y] with x, y = 0..CtbSizeY − 1 are derived as follows:
   - The variables hv1, hv0 and dirHV are derived as follows:
     - If sumV[x >> 2][y >> 2] is greater than sumH[x >> 2][y >> 2], the following applies:

$$hv1 = sumV[x >> 2][y >> 2] \quad (8\text{-}525)$$

$$hv0 = sumH[x >> 2][y >> 2] \quad 8\text{-}526)$$

$$dirHV = 1 \quad (8\text{-}527)$$

     - Otherwise, the following applies:

$$hv1 = sumH[x >> 2][y >> 2] \quad (8\text{-}528)$$

$$hv0 = sumV[x >> 2][y >> 2] \quad (8\text{-}529)$$

$$dirHV = 3 \quad (8\text{-}530)$$

   - The variables d1, d0 and dirD are derived as follows:
     - If sumD0[x >> 2][y >> 2] is greater than sumD1[x >> 2][y >> 2], the following applies:

$$d1 = sumD0[x >> 2][y >> 2] \quad (8\text{-}531)$$

$$d0 = sumD1[x >> 2][y >> 2] \quad (8\text{-}532)$$

$$dirD = 0 \quad (8\text{-}533)$$

     - Otherwise, the following applies:

$$d1 = sumD1[x >> 2][y >> 2] \quad (8\text{-}534)$$

$$d0 = sumD0[x >> 2][y >> 2] \quad (8\text{-}535)$$

$$dirD = 2 \quad (8\text{-}536)$$

   - The variables hvd1, hvd0, are derived as follows:

$$hvd1 = (d1 * hv0 > hv1 * d0) \ ?\ d1\ :\ hv1 \quad (8\text{-}537)$$

$$hvd0 = (d1 * hv0 > hv1 * d0) \ ?\ d0\ :\ hv0 \quad (8\text{-}538)$$

   - The variables dirS[x][y], dir1[x][y] and dir2[x][y] derived as follows:

$$dir1[x][y] = (d1 * hv0 > hv1 * d0) \ ?\ dirD\ :\ dirHV \quad (8\text{-}539)$$

$$dir2[x][y] = (d1 * hv0 > hv1 * d0) \ ?\ dirHV\ :\ dirD \quad (8\text{-}540)$$

$$dirS[x][y] = (hvd1 > 2 * hvd0) \ ?\ 1\ :\ ((hvd1 * 2 > 9 * hvd0) \ ?\ 2\ :\ 0) \quad (8\text{-}541)$$

4. The variable avgVar[x][y] with x, y = 0..CtbSizeY − 1 is derived as follows:

$$varTab[ ] = \{ 0, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4 \} \quad (8\text{-}542)$$

$$avgVar[x][y] = varTab[Clip3(0, 15, (sumOfHV[x >> 2][y >> 2] * 64) >> (3 + BitDepth_Y))] \quad (8\text{-}543)$$

5. The classification filter index array filtIdx[x][y] and the transpose index array transposeIdx[x][y] with x = y = 0..CtbSizeY − 1 are derived as follows:

   transposeTable[ ] = { 0, 1, 0, 2, 2, 3, 1, 3 }

   transposeIdx[x][y] = transposeTable[dir1[x][y] * 2 + (dir2[x][y] >> 1)]

   filtIdx[x][y] = avgVar[x][y]

   When dirS[x][y] is not equal 0, filtIdx[x][y] is modified as follows:

   filtIdx[x][y] += (((dir1[x][y] & 0x1) << 1) + dirS[x][y]) * 5 (8-544)

8.5.4.4 Coding tree block filtering process for chroma samples

Inputs of this process are:

– a reconstructed chroma picture sample array recPicture prior to the adaptive loop filtering process,

– a filtered reconstructed chroma picture sample array alfPicture,

– a chroma location (xCtbC, yCtbC) specifying the top-left sample of the current chroma coding tree block relative to the top left sample of the current picture.

Output of this process is the modified filtered reconstructed chroma picture sample array alfPicture.

The size of the current chroma coding tree block ctbSizeC is derived as follows:

ctbSizeC = CtbSizeY / SubWidthC (8-545)

For the derivation of the filtered reconstructed chroma samples alfPicture[x][y], each reconstructed chroma sample inside the current chroma coding tree block recPicture[x][y] is filtered as follows with x, y = 0..ctbSizeC − 1:

– The locations (h_x, v_y) for each of the corresponding chroma samples (x, y) inside the given array recPicture of chroma samples are derived as follows:

$$h_x = Clip3(0, pic\_width\_in\_luma\_samples / SubWidthC - 1, xCtbC + x) \quad (8\text{-}546)$$

$$v_y = Clip3(0, pic\_height\_in\_luma\_samples / SubHeightC - 1, yCtbC + y) \quad (8\text{-}547)$$

– The variable sum is derived as follows:

$$\begin{aligned} sum = {} & AlfCoeff_C[0] * (recPicture[h_x, v_{y+2}] + recPicture[h_x, v_{y-2}]) + \\ & AlfCoeff_C[1] * (recPicture[h_{x+1}, v_{y+1}] + recPicture[h_{x-1}, v_{y-1}]) + \\ & AlfCoeff_C[2] * (recPicture[h_x, v_{y+1}] + recPicture[h_x, v_{y-1}]) + \\ & AlfCoeff_C[3] * (recPicture[h_{x-1}, v_{y+1}] + recPicture[h_{x+1}, v_{y-1}]) + \\ & AlfCoeff_C[4] * (recPicture[h_{x+2}, v_y] + recPicture[h_{x-2}, v_y]) + \\ & AlfCoeff_C[5] * (recPicture[h_{x+1}, v_y] + recPicture[h_{x-1}, v_y]) + \\ & AlfCoeff_C[6] * recPicture[h_x, v_y] \end{aligned} \quad (8\text{-}548)$$

$$sum = (sum + 64) >> 7 \quad (8\text{-}549)$$

– The modified filtered reconstructed chroma picture sample alfPicture[xCtbC + x][yCtbC + y] is derived as follows:

$$alfPicture[xCtbC + x][yCtbC + y] = Clip3(0, (1 << BitDepth_C) - 1, sum) \quad (8\text{-}550)$$

9 Parsing process

9.1 General

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), se(v), uek(v) (see clause 9.2), tu(v) (see clause 9.3), tb(v) (see clause 9.4), or ae(v) (see clause 9.5).

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

9.2.1 General

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), uek(v) or se(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v) or se(v) are Exp-Golomb-coded with order k equal to 0 and syntax elements coded as uek(v) are Exp-Golomb-coded with order k. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

```
leadingZeroBits = −1
for( b = 0; !b; leadingZeroBits++ )                                    (9-1)
    b = read_bits( 1 )
```

The variable codeNum is then assigned as follows:

$$codeNum = (2^{leadingZeroBits} - 1) * 2^{k} + read\_bits(leadingZeroBits + k) \quad (9\text{-}2)$$

where the value returned from read_bits(leadingZeroBits) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

Table 9-1 illustrates the structure of the 0-th order Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed as specified above for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 9-1. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as x_i in Table 9-1, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each x_i is equal to either 0 or 1.

Table 9-1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative)

Bit string form	Range of codeNum
1	0
0 1 x_0	1..2
0 0 1 x_1 x_0	3..6
0 0 0 1 x_2 x_1 x_0	7..14
0 0 0 0 1 x_3 x_2 x_1 x_0	15..30
0 0 0 0 0 1 x_4 x_3 x_2 x_1 x_0	31..62
...	...

Table 9-2 illustrates explicitly the assignment of bit strings to codeNum values.

Table 9-2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative)

Bit string	codeNum
1	0
0 1 0	1
0 1 1	2
0 0 1 0 0	3
0 0 1 0 1	4
0 0 1 1 0	5
0 0 1 1 1	6
0 0 0 1 0 0 0	7
0 0 0 1 0 0 1	8
0 0 0 1 0 1 0	9
...	...

Depending on the descriptor, the value of a syntax element is derived as follows:

– If the syntax element is coded as ue(v), the value of the syntax element is equal to codeNum.

– Otherwise (the syntax element is coded as se(v)), the value of the syntax element is derived by invoking the mapping process for signed Exp-Golomb codes as specified in clause 9.2.2 with codeNum as input.

9.2.2 Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in clause 9.2.

Output of this process is a value of a syntax element coded as se(v).

The syntax element is assigned to the codeNum by ordering the syntax element by its absolute value in increasing order and representing the positive value for a given absolute value with the lower codeNum. Table 9-3 provides the assignment rule.

Table 9-3 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v)

codeNum	syntax element value
0	0
1	1
2	−1
3	2
4	−2
5	3
6	−3
k	$(-1)^{k+1}$ Ceil(k ÷ 2)

9.3 Parsing process for truncated unary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tu(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tu(v) are truncated unary coded. The range of possible values for the syntax element is determined first. The range of this syntax element is 0 to maxValinclusive, with maxVal being greater than or equal to 1. codeNum which is equal to the value of the syntax element is given by a process specified as follows:

```
codeNum = 0
keepGoing = 1
for(i = 0; i < maxVal && keepGoing; i++){
      keepGoing = read_bits( 1 )                                    (9-3)
      if( keepGoing )
            codeNum ++
}
```

9.4 Parsing process for truncated binary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tb(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tb(v) are truncated binary coded. The range of possible values for the syntax element is determined first. The range of this syntax element is 0 to maxValinclusive, with maxVal being greater than or equal to 1. synVal which is equal to the value of the syntax element is given by a process specified as follows:

```
thVal = 1 << 8
th = 8
while( thVal <= maxVal ) {
      th++
      thVal <<= 1
}
th− −
val = 1 << th                                                       (9-4)
b = n − val
synVal = read_bits( 1 )
if( synVal >= val - b ) {
      synVal <<= 1
      synVal += read_bits( 1 )
      synVal −= val − b
}
```

9.5 CABAC parsing process for slice segment data

9.5.1 General

9.5.2 Initialization process

9.5.3 Binarization process

9.5.3.1 General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 9-4 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the k-th order Exp-Golomb (EGk) binarization process and the fixed-length (FL) binarization process are given in clauses 9.5.3.3 through 9.5.3.5, respectively.

Table 9-4 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
coding_quadtree( )	qt_split_cu_flag[ ][ ]	FL	cMax = 1
multi_type_tree( )	mtt_split_cu_flag	FL	cMax = 1
	mtt_split_cu_vertical_flag	FL	cMax = 1
	mtt_split_cu_binary_flag	FL	cMax = 1
coding_tree_unit( )	alf_ctb_flag[ ][ ][ ]	FL	cMax = 1
coding_unit( )	cu_skip_flag[ ][ ]	FL	cMax = 1
	pred_mode_flag	FL	cMax = 1
	intra_luma_mpm_flag[ ][ ]	FL	cMax = 1
	intra_luma_mpm_idx[ ][ ]	TR	cMax = 2, cRiceParam = 0
	intra_luma_mpm_remainder[ ][ ]	FL	cMax = 63
	intra_chroma_pred_mode[ ][ ]	9.5.3.6	-
	merge_affine_flag[ ][ ]	FL	cMax = 1
	merge_flag[ ][ ]	FL	cMax = 1
	merge_idx[ ][ ]	TR	cMax = MaxNumMergeCand − 1, cRiceParam = 0
	inter_pred_idc[x0][y0]	9.5.3.7	-
	inter_affine_flag[ ][ ]	FL	cMax = 1
	cu_affine_type_flag[ ][ ]	FL	cMax = 1
	ref_idx_l0[ ][ ]	TR	cMax = num_ref_idx_l0_active_minus1, cRiceParam = 0
	mvp_l0_flag[ ][ ]	FL	cMax = 1
	ref_idx_l1[ ][ ]	TR	cMax = num_ref_idx_l1_active_minus1, cRiceParam = 0
	mvp_l1_flag[ ][ ]	FL	cMax = 1
	amvr_mode[ ][ ]	TR	cMax = 2, cRiceParam = 0
	cu_cbf	FL	cMax = 1
mvd_coding( )	abs_mvd_greater0_flag[ ]	FL	cMax = 1
	abs_mvd_greater1_flag[ ]	FL	cMax = 1
	abs_mvd_minus2[ ]	EG1	-
	mvd_sign_flag[ ]	FL	cMax = 1

Table 9-4 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
transform_unit( )	tu_cbf_luma[ ][ ][ ]	FL	cMax = 1
	tu_cbf_cb[ ][ ][ ]	FL	cMax = 1
	tu_cbf_cr[ ][ ][ ]	FL	cMax = 1
	tu_mts_flag[ ][ ]	FL	cMax = 1
residual_coding( )	transform_skip_flag[ ][ ][ ]	FL	cMax = 1
	last_sig_coeff_x_prefix	TR	cMax = (log2TrafoSize << 1) − 1, cRiceParam = 0
	last_sig_coeff_y_prefix	TR	cMax = (log2TrafoSize << 1) − 1, cRiceParam = 0
	last_sig_coeff_x_suffix	FL	cMax = (1 << ((last_sig_coeff_x_prefix >> 1) − 1) − 1)
	last_sig_coeff_y_suffix	FL	cMax = (1 << ((last_sig_coeff_y_prefix >> 1) − 1) − 1)
	coded_sub_block_flag[ ][ ]	FL	cMax = 1
	sig_coeff_flag[ ][ ]	FL	cMax = 1
	par_level_flag[ ]	FL	cMax = 1
	rem_abs_gt1_flag[ ]	FL	cMax = 1
	rem_abs_gt2_flag[ ]	FL	cMax = 1
	abs_remainder[ ]	9.5.3.8	cIdx, x0, y0, xC, yC, log2TbWidth, log2TbHeight
	coeff_sign_flag[ ]	FL	cMax = 1
	mts_idx[ ][ ][	FL	cMax = 3

9.5.3.2 Rice parameter derivation process

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the Rice parameter cRiceParam.

Given the syntax elements sig_coeff_flag[x][y] and the array AbsLevel[x][C] for the transform block with component index cIdx and the top-left luma location (x0, y0), the variable locSumAbs is derived as specified by the following pseudo code:

```
locSumAbs = 0
if( xC < (1 << log2TbWidth) − 1 ) {
    locSumAbs += AbsLevel[ xC + 1 ][ yC ] − sig_coeff_flag[ xC + 1 ][ yC ]
    if( xC < (1 << log2TbWidth) − 2 )
        locSumAbs += AbsLevel[ xC + 2 ][ yC ] − sig_coeff_flag[ xC + 2 ][ yC ]
    if( yC < (1 << log2TbHeight) − 1 )
        locSumAbs += AbsLevel[ xC + 1 ][ yC + 1 ] − sig_coeff_flag[ xC + 1 ][ yC + 1 ]                (9-5)
}
if( yC < (1 << log2TbHeight) − 1 ) {
    locSumAbs += AbsLevel[ xC ][ yC + 1 ] − sig_coeff_flag[ xC ][ yC + 1 ]
    if( yC < (1 << log2TbHeight) − 2 )
        locSumAbs += AbsLevelPass1 [ xC ][ yC + 2 ] − sig_coeff_flag[ xC ][ yC + 2 ]
}
```

The Rice parameter cRiceParam is derived as follows:

– If locSumAbs is less than 12, cRiceParam is set equal to 0;

– Otherwise, if locSumAbs is less than 25, cRiceParam is set equal to 1;

– Otherwise (locSumAbs is greater than or equal to 25), cRiceParam is set equal to 2.

9.5.3.3 Truncated Rice binarization process

Input to this process is a request for a truncated Rice (TR) binarization, cMax and cRiceParam.

Output of this process is the TR binarization associating each value symbolVal with a corresponding bin string.

A TR bin string is a concatenation of a prefix bin string and, when present, a suffix bin string.

For the derivation of the prefix bin string, the following applies:

– The prefix value of symbolVal, prefixVal, is derived as follows:

$$prefixVal = symbolVal \gg cRiceParam \tag{9-6}$$

– The prefix of the TR bin string is specified as follows:

  – If prefixVal is less than cMax >> cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0. Table 9-5 illustrates the bin strings of this unary binarization for prefixVal.

  – Otherwise, the bin string is a bit string of length cMax >> cRiceParam with all bins being equal to 1.

Table 9-5 – Bin string of the unary binarization (informative)

prefixVal	Bin string					
0	0					
1	1	0				
2	1	1	0			
3	1	1	1	0		
4	1	1	1	1	0	
5	1	1	1	1	1	0
...						
binIdx	0	1	2	3	4	5

When cMax is greater than symbolVal and cRiceParam is greater than 0, the suffix of the TR bin string is present and it is derived as follows:

– The suffix value suffixVal is derived as follows:

$$suffixVal = symbolVal - ((prefixVal) \ll cRiceParam) \tag{9-7}$$

– The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified in clause 9.5.3.5 for suffixVal with a cMax value equal to (1 << cRiceParam) − 1.

NOTE – For the input parameter cRiceParam = 0, the TR binarization is exactly a truncated unary binarization and it is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

9.5.3.4 k-th order Exp-Golomb binarization process

Inputs to this process is a request for a k-th order Exp-Golomb (EGk) binarization.

Output of this process is the EGk binarization associating each value symbolVal with a corresponding bin string.

The bin string of the EGk binarization process for each value symbolVal is specified as follows, where each call of the function put(X), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

```
absV = Abs( symbolVal )
stopLoop = 0
do
    if( absV >= ( 1 << k ) ) {
        put( 1 )
        absV = absV − ( 1 << k )
        k++
    } else {
        put( 0 )
        while( k− − )
            put( ( absV >> k ) & 1 )
        stopLoop = 1
    }
while( !stopLoop )
```

(9-8)

NOTE – The specification for the k-th order Exp-Golomb (EGk) code uses 1's and 0's in reverse meaning for the unary part of the Exp-Golomb code of k-th order as specified in clause 9.2.

9.5.3.5 Fixed-length binarization process

Inputs to this process are a request for a fixed-length (FL) binarization and cMax.

Output of this process is the FL binarization associating each value symbolVal with a corresponding bin string.

FL binarization is constructed by using the fixedLength-bit unsigned integer bin string of the symbol value symbolVal, where fixedLength = Ceil(Log2(cMax + 1)). The indexing of bins for the FL binarization is such that the binIdx = 0 relates to the most significant bit with increasing values of binIdx towards the least significant bit.

9.5.3.6 Binarization process for intra_chroma_pred_mode

Input to this process is a request for a binarization for the syntax element intra_chroma_pred_mode.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element intra_chroma_pred_mode is specified in Table 9-6 and Table 9-7.

Table 9-6 – Binarization for intra_chroma_pred_mode when sps_cclm_enabled_flag is equal to 0

Value of intra_chroma_pred_mode	Bin string
4	0
0	100
1	101
2	110
3	111

Table 9-7 – Binarization for intra_chroma_pred_mode when sps_cclm_enabled_flag is equal to 1

Value of intra_chroma_pred_mode	Bin string
5	0
4	10
0	1100
1	1101
2	1110
3	1111

9.5.3.7 Binarization process for inter_pred_idc

Input to this process is a request for a binarization for the syntax element inter_pred_idc.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element inter_pred_idc is specified in Table 9-8.

Table 9-8 – Binarization for inter_pred_idc

Value of inter_pred_idc	Name of inter_pred_idc	Bin string
0	PRED_L0	00
1	PRED_L1	01
2	PRED_BI	1

9.5.3.8 Binarization process for abs_remainder[ ]

Input to this process is a request for a binarization for the syntax element abs_remainder[n], the colour component cIdx, the luma location (x0, y0) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the picture), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the binarization of the syntax element.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process as specified in clause 9.5.3.2 with the colour component index cIdx, the luma location (x0, y0), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as inputs.

The variable cMax is derived from cRiceParam as:

$$cMax = (cRiceParam == 1 \ ? \ 6 \ : \ 7) << cRiceParam \tag{9-9}$$

The binarization of the syntax element abs_remainder[n] is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

– The prefix value of abs_remainder[n], prefixVal, is derived as follows:

$$prefixVal = Min(cMax, abs\_remainder[n]) \tag{9-10}$$

– The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with the variables cMax and cRiceParam as inputs.

When the prefix bin string is equal to the bit string of length 4 with all bits equal to 1, the suffix bin string is present and it is derived as follows:

– The suffix value of abs_remainder[n], suffixVal, is derived as follows:

$$suffixVal = abs\_remainder[n] - cMax \tag{9-11}$$

– The suffix bin string is specified by invoking the k-th order EGk binarization process as specified in clause 9.5.3.4 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1.

9.5.4 Decoding process flow

9.5.4.1 General

9.5.4.2 Derivation process for ctxTable, ctxIdx and bypassFlag

9.5.4.2.1 General

9.5.4.2.2 Derivation process of ctxInc for the syntax element tu_cbf_cr[ ][ ]

Inputs to this process is the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture.

Outputs of this process is the variable ctxInc.

The variable ctxInc is set equal to tu_cbf_cb[x0][y0].

9.5.4.2.3 Derivation process for the variables locNumSig, locSumAbsPass1

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Outputs of this process are the variables locNumSig and locSumAbsPass1.

Given the syntax elements sig_coeff_flag[x][y] and the array AbsLevelPass1[x][C] for the transform block with component index cIdx and the top-left luma location (x0, y0), the variables locNumSig and locSumAbsPass1 are derived as specified by the following pseudo code:

```
locNumSig      = 0
locSumAbsPass1 = 0
if( xC < (1 << log2TbWidth) − 1 ) {
    locNumSig      += sig_coeff_flag[ xC + 1 ][ yC ]
    locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC ]
    if( xC < (1 << log2TbWidth) − 2 ) {
        locNumSig      += sig_coeff_flag[ xC + 2 ][ yC ]
        locSumAbsPass1 += AbsLevelPass1[ xC + 2 ][ yC ]
    }
    if( yC < (1 << log2TbHeight) − 1 ) {
        locNumSig      += sig_coeff_flag[ xC + 1 ][ yC + 1 ]                    (9-12)
        locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC + 1 ]
    }
}
if( yC < (1 << log2TbHeight) − 1 ) {
    locNumSig      += sig_coeff_flag[ xC ][ yC + 1 ]
    locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 1 ]
    if( yC < (1 << log2TbHeight) − 2 ) {
        locNumSig      += sig_coeff_flag[ xC ][ yC + 2 ]
        locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 2 ]
    }
}
```

9.5.4.2.4 Derivation process of ctxInc for the syntax element sig_coeff_flag

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.3 with colour component index cIdx, the luma location (x0, y0), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

– If cIdx is equal to 0, ctxInc is derived as follows:

$$ctxInc = 18 * Max(0, QState - 1) + Min(locSumAbsPass1, 5) + (d < 2 \ ? \ 12 \ : \ (d < 5 \ ? \ 6 \ : \ 0)) \quad (9\text{-}13)$$

– Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

$$ctxInc = 54 + 12 * Max(0, QState - 1) + Min(locSumAbsPass1, 5) + (d < 2 \ ? \ 6 \ : \ 0) \quad (9\text{-}14)$$

9.5.4.2.5 Derivation process of ctxInc for the syntax elements par_level_flag, rem_abs_gt1_flag, and rem_abs_level_gt2_flag

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variablea locNumSig and locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.3 with colour component index cIdx, the luma location (x0, y0), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable ctxOffset is set equal to Min(locSumAbsPass1 − locNumSig, 4).

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

– If xC is equal to LastSignificantCoeffX and yC is equal to LastSignificantCoeffY, ctxInc is derived as follows:

$$ctxInc = (cIdx \ == \ 0 \ ? \ 0 \ : \ 21) \quad (9\text{-}15)$$

– Otherwise, if cIdx is equal to 0, ctxInc is derived as follows:

$$ctxInc = 1 + ctxOffset + (d \ == \ 0 \ ? \ 15 \ : \ (d < 3 \ ? \ 10 \ : \ (d < 10 \ ? \ 5 \ : \ 0))) \quad (9\text{-}16)$$

– Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

$$ctxInc = 22 + ctxOffset + (d \ == \ 0 \ ? \ 5 \ : \ 0) \quad (9\text{-}17)$$

Joint Video Experts Team (JVET)
of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11
12th Meeting: Macao, CN, 3–12 Oct. 2018

Document: JVET-L0072

Title: **CE11: Long deblocking filters for luma (CE11.1.1) and for both luma and chroma (CE11.1.9)**

Status: Input document to JVET

Purpose: Proposal

Author(s) or Contact(s): K. Andersson, Z. Zhang, R. Sjöberg

Tel: +46730371390
Email: kenneth.r.andersson@ericsson.com

Source: Ericsson

Abstract

This contribution proposes long filters and decisions for luma in CE11.1.1 and for both luma and chroma using HEVC luma deblocking for chroma deblocking in CE11.1.9. Deblocking is applied on an 8x8 grid. Long filters for luma can be selected for boundary segments of 16 samples, symmetrically deblocking 7 or 5 samples on respective side of the block boundary for block sizes equal to or larger than 32x32 respectively 16x16. The long filters are designed to not modify a plane/ramp and to be computationally efficient to implement.

The proposed solution in CE11.1.1 is claimed to improve subjective quality at low bitrates. It also provides a luma BD rate impact of -0.01%/-0.01%/0.02%/0.02% for AI/RA/LD/LDP compared to VTM-2.0.1 and -0.03%/-0.05%/0.06%/-0.07% for AI/RA/LD/LDP compared to VTM-2.0.1 when ALF is turned off. Decoding time 99%/100%/100%/101% for AI/RA/LD/LDP compared to VTM-2.0.1 and 99%/101%/101%/101% for AI/RA/LD compared to VTM-2.0.1 when ALF is turned off.

The proposed solution in CE11.1.9 is claimed to improve subjective quality at low bitrates. It also provides a luma BD rate impact of -0.01%/-0.03%/-0.05%/-0.09% for AI/RA/LD/LDP compared to VTM-2.0.1 and -0.03%/-0.07%/-0.04%/-0.15% for AI/RA/LD/LDP compared to VTM-2.0.1 when ALF is turned off. Decoding time 103%/104%/105%/104% for AI/RA/LD/LDP compared to VTM-2.0.1 and 103%/106%/107%/105% for AI/RA/LD compared to VTM-2.0.1 when ALF is turned off.

1 Introduction

The deblocking in the current test model is based on HEVC deblocking and is applied on an 8x8 grid. The proposed long filters and decisions for luma in CE11.1.1 and CE11.1.9 [1] were originally proposed in CfP response JVET-J0012 [2], JVET-K0307 [3], JVET-K0318 [4]. The long filters and decisions on top of the HEVC filters and decisions are described in section 2. In section 3 the use of the HEVC luma filters and decisions for chroma deblocking is described, originally proposed in JVET-K0318 [4]. In section 4 objective results in comparison to VTM-2.0.1 and VTM-2.0.1 without ALF are shown. Finally some conclusions in section 5.

2 Long filters and decisions for luma

The approach is to apply a long filter from a set of long filters to symmetrically filter 5 or 7 samples on respective side of a block boundary for larger blocks and smooth regions. The filters are designed to not modify a plane/ramp and to be computationally efficient to implement.

The filtering of N samples on each side of a line of samples across the block boundary is performed as follows. First pre-filter to obtain three virtual samples, one centered at a midpoint refMiddle between

current block Q and neighbouring block P, one refP centered between two samples in block P and one refQ centered between two samples in block Q. Then two tap linear interpolation between refMiddle and refP to determined filtered samples in block P, and between refMiddle and refQ to determine filtered samples in block Q. This procedure facilitates that a plan/ramp not is modified but that discontinuities between block P and Q can be reduced.

One of the longer filters is used when the strong weak filter decision (identical with HEVC/VTM/BMS) indicates strong filter for 8 lines of a 16 sample segment of a boundary to be filtered and that samples near the boundary are smooth. This gives less frequent use of long filters and also less frequent switching between long filters and weaker filters which are determined for segments of four samples. Filter decisions are shown in the Table 1. The 7 sample filter is used if the current and the neighbouring CUs have a width and height equal or greater than 32 and when 8 samples in current and neighbouring block are smooth. If that is not the case the 5 sample filter is used if the current and neighbouring CUs have a width and height equal or greater than 16 and when 6 samples in current and neighbouring block are smooth. If that is not the case filters in VTM/BMS are used.

Table 1: Filter decisions

Size	Conditions for each of the 8 lines of a 16 sample segment (line 0, 3, 4, 7, 8, 11, 12, 15)
7	$HEVC_{strong\_weak}$&& $\left(\lvert p_0 - 2*p_2 + p_4\rvert + \lvert q_0 - 2*q_2 + q_4\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - p_2 - p_3 + p_5\rvert + \lvert q_0 - q_2 - q_3 + q_5\rvert < \frac{3*\beta}{32}\right)$&& $\left(\lvert p_0 - 2*p_3 + p_6\rvert + \lvert q_0 - 2*q_3 + q_6\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - p_3 - p_4 + p_7\rvert + \lvert q_0 - q_3 - q_4 + q_7\rvert < \frac{3*\beta}{32}\right)$
5	$HEVC_{strong\_weak}$&& $\left(\lvert p_0 - p_4\rvert + \lvert q_0 - q_4\rvert < \frac{\beta}{8}\right)$ && $\left(\lvert p_0 - p_5\rvert + \lvert q_0 - q_5\rvert < \frac{\beta}{8}\right)$
0-3	HEVC filter and decisions apply

Block boundary samples p_i and q_i for i=0 to S-1 are then replaced by linear interpolation as follows:

- $p_i' = (f_i * Middle_s + (64 - f_i) * P_s) + 32) \gg 6, clipped\ to\ p_i \pm \text{tc}$
- $q_i' = (f_i * Middle_s + (64 - f_i) * Q_s + 32) \gg 6), clipped\ to\ q_i \pm \text{tc}$

where f, Middle, P and Q are given below:

Table 2: Filter kernels

S	Filter kernels
7	$f_i = 59 - i * 9,$ $can\ also\ be\ described\ as\ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $Middle_7 = (2 * (p_o + q_o) + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1,$ $Q_7 = (q_6 + q_7 + 1) \gg 1$
5	$f_i = 58 - i * 13,$ $can\ also\ be\ described\ as\ \boldsymbol{f} = \{58,45,32,19,6\}$ $Middle_5 = (2 * (p_o + q_o + p_1 + q_1 + p_2 + q_2) + p_3 + q_3 + p_4 + q_4 + 8) \gg 4$ $P_5 = (p_4 + p_5 + 1) \gg 1,$ $Q_5 = (q_4 + q_5 + 1) \gg 1$

2.1 Complexity analysis

Filtering is performed on an 8x8 grid as for the anchor. All filtering can be performed in parallel. Two cases of long filters are supported, either 7 or 5 samples are modified on each of a block boundary. At most 8 samples are read from both sides of a block boundary when 7 samples of filtering is selected which gives a line buffer requirement of 8 lines. The size of the blocks adjacent to the block boundary is at least 32x32 in the case of 7+7 sample filtering and at least 16x16 for the case of 5+5 sample filtering. Long filtering is performed on segments of 16 samples.

2.1.1 Filtering

First two tap filterings and one 14 or 10 tap filtering. Then bi-linear filtering on respective side of the boundary.

7+7 sample filtering of one line requires in total 120 operations including rounding (24 multiplications, 46 additions, 22 shifts, 28 cmp).

5+5 sample filtering of one line requires in total 88 operations including rounding (16 multiplications, 34 additions, 18 shifts, 20 cmp).

Worst case: 120 operations for block size P and Q of 32x32.

2.1.2 Decision

In case HEVC strong filter decision is true, long filter decisions are checked.

7+7 sample filtering decision for a line (8 of 16 lines are checked):

- Compare width and height for both current and neighbouring block with a threshold.
- Smoothness: 20 operations (12 cmp, 24 additions, 4 shifts)/2
- Total operations per line: 24

5+5 sample filtering decisions for a line (8 of 16 lines are checked):

- Compare width and height for both current and neighbouring block with a threshold.
- Smoothness: 6 operations (6 cmp, 6 additions)/2
- Total operations per line: 10

Worst case: 30 operations per line for block size P and Q of 32x32.

3 Deblocking for chroma

The HEVC luma filters and decisions are used to deblock chroma with some adaptations but with same 8x8 granularity as before.

First adaptation is to filter chroma also for inter blocks when bs=1, re-using the luma decision for motion check but adding specific cbf check for respective chroma component.

Second adaptation is to limit the maximum deblocking filter length on P side (neighbouring CU) with the length of neighbouring block orthogonal to the boundary divided by 4 and for Q side (current CU) the length of the current block orthogonal to the boundary divided by 4.

lengthP = edge==EDGE_HOR ? (CUP_height_chroma >> 2) : (CUP_width_chroma >>2)

lengthQ = edge==EDGE_HOR ? (CUQ_height_chroma >> 2) : (CUQ_width_chroma >>2)

Filtering of the boundary is only applied if both sides have a maximum deblocking filter length larger than 0. The HEVC strong filter (reading 4 samples and modifying 3 samples on respective side) is only applied if both sides have a maximum deblocking filter length larger than 2.

Thirdly the decision for strong/weak filter and number of samples to filter is determined for each line instead of for a group of 4 sample segment as in HEVC. Inter blocks are only filtered if Laplacian

gradients near the boundary on respective side is below the structure threshold “beta” as in HEVC but for intra blocks that check is by passed.

Fourthly is “beta” parameter divided by 8 compared to what is used for luma. This makes the use of more than 1 pixel to filter only happening when there is very little structure near the boundary.

3.1 Complexity analysis

Filtering is performed on an 8x8 grid as for the anchor. All filtering can be performed in parallel. Same filter combinations (HEVC strong and weak filter) as for the anchor but for luma. At most 4 samples is read from both sides of a block boundary when 3 samples of filtering is selected (HEVC strong filter) which gives a line buffer requirement of 4 lines. The size of the blocks adjacent to the block boundary is at least 4xN in the case of 1 sample weak filtering, 8xN in the case of 2 sample weak filtering and at least 16xN for the case of 3 sample strong filtering. Filtering is performed for each line of chroma samples.

3.1.1 Filtering

3 sample filtering: HEVC strong filter: 52 operations (2 multiplications, 28 additions, 10 shifts, 12 cmp)

1 or 2 simple filtering: HEVC weak filter: 33 operations (2 multiplications, 16 additions, 5 shifts, 8 cmp)

Worst case: 52 operations per line for block size P 16xN and Q 16xN

3.1.2 Decisions

HEVC d decision: 4 operations (5 additions, 3 cmp)/2

HEVC strong weak decision: 10 operations (4 additions, 6 cmp)

HEVC weak decision: 4 operations (4 cmp)

Worst case: 18 operations per line for block size P 16xN and Q 16xN

4 Results

The simulations were performed using the CTC with the inclusion of LDP. We also tested the performance with ALF disabled. The decoding times for the anchor and test configurations were measured using the same machines, but encoding times were measured on different machines which make the encoding times unreliable. However, since the decoding times are magnitudes shorter and the deblocking filter is done once per picture for both encoder and decoder, we conclude the encoding time difference is negligible.

The results indicate that that the objective performance of the proposed long filters for luma as in CE11.1.1 is similar to the anchor. It has slightly better objective performance when ALF is disabled than when it is enabled. Encoding time is about 100% and decoding time is about 101% compared to the anchor.

For the case of both long luma filters and longer chroma filters as in CE11.1.9 the objective performance is slightly better than the anchor. Also here the objective performance is slightly better when ALF is disabled than when it is enabled. Encoding time is about 100% and decoding time 106% compared to the anchor.

4.1 Objective performance for CE11.1.1

	All Intra Main10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.02%	-0.01%	0.00%	106%	98%	-0.09%	-0.02%	0.00%	103%	98%
Class A2	-0.01%	0.00%	0.00%	108%	99%	-0.01%	-0.01%	0.00%	94%	100%

Class B	0.00%	0.00%	0.00%	108%	98%	-0.01%	0.00%	0.00%	100%	99%
Class C	0.00%	0.00%	0.00%	107%	101%	0.00%	0.00%	0.00%	101%	99%
Class E	-0.06%	-0.01%	0.00%	98%	100%	-0.06%	-0.01%	0.00%	98%	100%
Overall	-0.01%	0.00%	0.00%	106%	99%	-0.03%	-0.01%	0.00%	99%	99%
Class D	0.00%	0.00%	0.00%	104%	101%	0.00%	0.00%	0.00%	104%	101%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Random Access Main 10									
	VTM-2.0.1					**VTM-2.0.1 ALF=0**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.12%	0.01%	0.03%	99%	100%	-0.24%	-0.01%	0.02%	99%	104%
Class A2	0.00%	-0.10%	-0.08%	99%	101%	-0.04%	0.05%	0.04%	101%	98%
Class B	0.03%	-0.01%	-0.09%	97%	100%	-0.01%	0.04%	0.00%	100%	102%
Class C	0.02%	0.06%	-0.03%	100%	102%	0.02%	0.02%	-0.07%	101%	101%
Class E										
Overall	-0.01%	-0.01%	-0.05%	99%	100%	-0.05%	0.03%	-0.01%	100%	101%
Class D	0.03%	-0.04%	0.01%	97%	100%	-0.01%	0.00%	0.08%	99%	101%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay B Main10									
	VTM-2.0.1					**VTM-2.0.1 ALF=0**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.08%	-0.17%	0.00%	103%	100%	0.06%	0.09%	-0.10%	102%	101%
Class C	0.03%	-0.13%	-0.10%	102%	100%	0.01%	-0.15%	-0.13%	100%	102%
Class E	-0.07%	0.37%	-0.07%	95%	102%	0.14%	-0.09%	0.29%	96%	102%
Overall	0.02%	-0.02%	-0.05%	101%	100%	0.06%	-0.04%	-0.01%	100%	101%
Class D	0.00%	0.03%	-0.49%	104%	100%	-0.03%	-0.48%	-0.06%	103%	99%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay P Main10									
	VTM-2.0.1					**VTM-2.0.1 ALF=0**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.01%	0.17%	0.18%	102%	100%	-0.06%	-0.03%	-0.09%	101%	100%
Class C	0.02%	0.02%	0.03%	100%	101%	0.03%	0.12%	0.00%	101%	100%
Class E	0.03%	0.00%	0.60%	107%	101%	-0.22%	0.23%	-0.20%	100%	102%
Overall	0.02%	0.08%	0.24%	102%	101%	-0.07%	0.08%	-0.09%	101%	101%
Class D	0.04%	0.05%	-0.32%	103%	101%	-0.01%	-0.40%	-0.18%	102%	101%
Class F	#VALU	#VALU	#VALU	#NU	#NUM	#VALU	#VALU	#VALU	#NUM	#NUM

(optional)	E!	E!	E!	M!	!	E!	E!	E!	!	!

4.2 Objective performance for CE11.1.9

	All Intra Main10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.02%	-0.90%	-0.93%	101%	105%	-0.09%	-1.36%	-1.24%	104%	103%
Class A2	0.00%	-0.41%	-0.32%	104%	103%	0.00%	-0.32%	-0.18%	97%	103%
Class B	0.00%	-0.72%	-0.80%	105%	103%	-0.01%	-0.66%	-0.69%	95%	103%
Class C	0.00%	-0.45%	-0.46%	102%	102%	0.00%	-0.20%	-0.18%	102%	102%
Class E	-0.05%	-0.68%	-0.53%	100%	104%	-0.06%	-0.78%	-0.65%	97%	104%
Overall	-0.01%	-0.63%	-0.62%	103%	103%	-0.03%	-0.64%	-0.58%	99%	103%
Class D	0.00%	-0.42%	-0.50%	106%	101%	0.00%	-0.10%	-0.09%	97%	103%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Random Access Main 10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.13%	-2.41%	-3.28%	98%	103%	-0.27%	-3.46%	-4.09%	99%	108%
Class A2	-0.03%	-1.62%	-1.47%	99%	105%	-0.06%	-1.63%	-1.35%	101%	105%
Class B	0.01%	-1.97%	-2.18%	97%	104%	-0.02%	-2.10%	-2.32%	99%	106%
Class C	-0.01%	-1.38%	-1.55%	99%	105%	0.01%	-1.79%	-1.96%	99%	105%
Class E										
Overall	-0.03%	-1.83%	-2.09%	98%	104%	-0.07%	-2.20%	-2.39%	99%	106%
Class D	0.00%	-1.21%	-1.25%	99%	105%	0.02%	-1.63%	-1.69%	95%	104%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay B Main10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.00%	-1.72%	-1.98%	101%	104%	-0.01%	-1.66%	-2.24%	103%	107%
Class C	-0.06%	-1.46%	-1.57%	105%	105%	-0.07%	-1.29%	-1.44%	100%	107%
Class E	-0.13%	-3.09%	-3.03%	95%	107%	-0.04%	-2.84%	-2.45%	95%	108%
Overall	-0.05%	-1.98%	-2.10%	101%	105%	-0.04%	-1.83%	-2.03%	100%	107%
Class D	-0.01%	-0.34%	-0.82%	100%	104%	-0.06%	-0.80%	-0.84%	102%	103%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay P Main10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT

Class A1										
Class A2										
Class B	-0.12%	-2.22%	-2.16%	101%	103%	-0.15%	-2.65%	-2.98%	103%	106%
Class C	-0.07%	-1.34%	-1.38%	99%	105%	-0.06%	-1.17%	-1.49%	100%	105%
Class E	-0.09%	-3.15%	-2.47%	101%	104%	-0.26%	-3.28%	-3.57%	96%	103%
Overall	-0.09%	-2.16%	-1.98%	100%	104%	-0.15%	-2.32%	-2.63%	100%	105%
Class D	-0.01%	-1.03%	-1.26%	101%	104%	-0.08%	-1.71%	-1.32%	102%	98%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#NUM!	#VALUE!	#VALUE!	#VALUE!	#NUM!	#NUM!

5 Conclusions

We propose to adopt the proposed long filters for luma to combat very visual artifacts for large blocks. We also propose to use HEVC luma deblocking for chroma deblocking but that mainly improves subjective performance on the Campfire sequence.

6 References

[1] A. Norkin, A. Meher Kotra, "Description of Core Experiment 11 (CE11): Deblocking", JVET-K1031, Ljubljana, July 2018

[2] R. Sjöberg, K. Andersson, R. Yu, Z. Zhang, P. Wennersten, J. Lainema, A. Hallapuro, A. Aminlou, M. Hannuksela, R. Ghaznavi-Youvalari, J. Ridge, "Description of SDR and HDR video coding technology proposal by Ericsson and Nokia", JVET-J0012, San Diego, April 2018

[3] K. Andersson, Z. Zhang, R. Sjöberg, "CE2-2.1.1: Long deblocking filters and fixes", JVET-K0307, Ljubljana, July 2018

[4] K. Andersson, Z. Zhang, R. Sjöberg, "CE2-2.1.1 related: HEVC luma filters and decisions for chroma deblocking", JVET-K0318, Ljubljana, July 2018

[5] J. Boyce, K. Suehring, X. Li, V. Seregin, "JVET common test conditions and software reference configurations", JVET-K1010, Ljubljana, July 2018

7 Patent rights declaration(s)

Ericsson may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).

Joint Video Experts Team (JVET)
of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11
12th Meeting: Macao, CN, 3–12 Oct. 2018

Document: JVET-L1031-v3

Title: **Description of Core Experiment 11 (CE11): Deblocking**

Status: Output Document to JVET

Purpose: Core Experiment description

Author(s) or Contact(s): Andrey Norkin
Anand Meher Kotra

Tel:
Email: anorkin@netflix.com
anand.meher.kotra@huawei.com

Source: CE coordinators

Abstract

The goal of this Core Experiment (CE) is to improve and/or enhance deblocking filtering techniques in the test model (VTM). Techniques covered by this CE are related long deblocking filters, filtering on the 4x4 boundaries, and other modifications to deblocking.

The deblocking filter techniques are evaluated according to subjective picture quality as well as BD-rate gains. These are organized in CE11.1 and CE11.2 which address aspects of the long deblocking filters, and filtering at 4x4 block boundaries. Participants in this activity are Ericsson, HHI, Huawei, KDDI, MediaTek, Panasonic, Samsung, Qualcomm, SHARP, and Sony.

The software basis for this CE is VTM-3.0 [2]. For the test sequences, configurations and test conditions, in JVET-L1010 [1] for SDR sequences are used. For subjective viewing, additional results with sequences encoded with higher QPs may be required.

1 Participants

Nr.	Name	Company	Email
1	Alexis Tourapis	Apple	atourapis@apple.com
2	Li Zhang	Bytedance	lizhang.idm@bytedance.com
3	Patrice Onno	Canon	patrice.onno@crf.canon.fr
4	Jonathan Taquet	Canon	jonathan.taquet@crf.canon.fr
5	Christophe Gisquet	Canon	christophe.gisquet@crf.canon.fr
6	Dan Grois	Comcast	grois@ieee.org
7	Yongjo Ahn	Digital Insights	yjahn@digitalinsights.co.kr
8	Xiaozhen Zheng	DJI	xiaozhen.zheng@dji.com
9	Peng Yin	Dolby	pyin@dolby.com
10	Taoran Lu	Dolby	tlu@dolby.com
11	Kenneth Andersson	Ericsson	kenneth.r.andersson@ericsson.com
12	Jungwon Kang	ETRI	jungwon@etri.re.kr
13	Hahyun Lee	ETRI	hanilee@etri.re.kr
14	Yao-Jen Chang	Foxconn	yaojen.chang.foxconn@gmail.com

15	Tim Li	Foxconn	tim410177@gmail.com
16	Hong-Jheng Jhu	Foxconn	ted0958.am96@g2.nctu.edu.tw
17	Benjamin Bross	HHI	benjamin.bross@hhi.fraunhofer.de
18	Christian Helmrich	HHI	christian.helmrich@hhi.fraunhofer.de
19	Adam Wieckowski	HHI	adam.wieckowski@hhi.fraunhofer.de
20	Jianhua Zheng	HiSilicon	zhengjianhua@hisilicon.com
21	Xu Chen	HiSilicon	anderson.chen@hisilicon.com
22	Na Zhang	HiSilicon	zhangna25@hisilicon.com
23	Haechul Choi	HNU	choihc@hanbat.ac.kr
24	Heeji Han	HNU	sarhakor@naver.com
25	Daehyeok Gwon	HNU	skyeye0530@naver.com
26	Jaeryun Choe	HNU	tjswo9395@naver.com
27	Sergey Ikonin	Huawei	sergey.ikonin@huawei.com
28	Filippov Alexey	Huawei	alexey.filippov@huawei.com
29	Vasily Rufitskiy	Huawei	vasily.rufitskiy@huawei.com
30	Biao Wang	Huawei	biao.wang@huawei.com
31	Jianle Chen	Huawei	Jianle.chen@huawei.com
32	Grace Yu	Intel	grace.yu@intel.com
33	Samuel Wong	Intel	swong10h266@gmail.com
34	Rahul Vanam	InterDigital	Rahul.vanam@interdigital.com
35	Kyohei Unno	KDDI	ky-unno@kddi-research.jp
36	Kei Kawamura	KDDI	ki-kawamura@kddi.com
37	Yi-Wen Chen	Kwai	yiwenchen@kwai.com
38	Xianglin Wang	Kwai	xianglinwang@kwai.com
39	Seung Hwan Kim	LG	seunghwan3.kim@lge.com
40	Junghak Nam	LG	junghak.nam@lge.com
41	Hyeongmun Jang	LG	hm.jang@lge.com
42	Jie Zhao	LG	jie.zhao@lge.com
43	Jaehyun Lim	LG	jaehyun.lim@lge.com
44	Chih-Wei Hsu	MediaTek	cw.hsu@mediatek.com
45	Chia-ming Tsai	MediaTek	Chia-ming.Tsai@mediatek.com
46	Ching-Yeh Chen	MediaTek	chingyeh.chen@mediatek.com
47	Tzu-Der Chuang	MediaTek	Peter.Chuang@mediatek.com
48	Yu-Wen Huang	MediaTek	yuwen.huang@mediatek.com
49	Atsuro Ichigaya	NHK	ichigaya.a-go@nhk.or.jp
50	Shunsuke Iwamura	NHK	iwamura.s-gc@nhk.or.jp
51	Shimpei Nemoto	NHK	nemoto.s-fy@nhk.or.jp
52	Fannie Yu	Oppo	yuyuanfang@oppo.com
53	Tadamasa Toma	Panasonic	toma.tadamasa@jp.panasonic.com

54	Kiyofumi Abe	Panasonic	abe.kiyo@jp.panasonic.com
55	Dmytro Rusanovskyy	Qualcomm	dmytror@qti.qualcomm.com
56	Wei-Jung Chien	Qualcomm	wchien@qti.qualcomm.com
57	Chun-Chi Chen	Qualcomm	chunchic@qti.qualcomm.com
58	Yan Zhang	Qualcomm	yzh@qti.qualcomm.com
59	Chao-Hsiung Hung	Qualcomm	chaohsiu@qti.qualcomm.com
60	Kiho Choi	Samsung	kiho14.choi@samsung.com
61	Woongil Choi	Samsung	woongil.choi@samsung.com
62	Weijia Zhu	Sharp	zhuw@sharplabs.com
63	Andrew Segall	Sharp	asegall@sharplabs.com
64	Kiran Misra	Sharp	misrak@sharplabs.com
65	Takeshi Chujoh	Sharp	chujoh.takeshi@sharp.co.jp
66	Tomohiro Ikai	Sharp	ikai.tomohiro@sharp.co.jp
67	Dongjae Won	SJU	bigdj2002@sju.ac.kr
68	Jaemin Ha	SJU	hajama0123@sju.ac.kr
69	Masaru Ikeda	Sony	masaru.ikeda@sony.com
70	Teruhiko Suzuki	Sony	teruhiko.s@sony.com
71	Eric Chai	Ubilinx	eric.chai@ublnx.com
72	Weimin Zeng	Ubilinx	weimin.zeng@ublnx.com
73	Jason Jung	WILUS Inc	jason.jung@wilusgroup.com

2 Test conditions and evaluation criteria

The proposals should be encoded according to two test conditions, which correspond to two anchors.

Anchor-1 is VTM-3.0 according to the CTC [1].

Anchor-2 is VTM-3.0 with ALF switched off (other conditions are the same as in Anchor 2).

The proposals in the CE shall be implemented based on VTM-3.0. The results for two test conditions shall be provided: Test Conditions 1 use configuration of Anchor 1, Test Conditions 2 use configuration of Anchor 2. The results for LDP configuration are mandatory for this CE.

Proposals will be compared with respect to subjective, objective quality, and complexity.

3 Tool description

3.1 Introduction

The main goal with this Core Experiment is to enhance the subjective quality of the Test Model and to ensure that CUs can be filtered in parallel.

Following aspects will be studied:

- Long deblocking filters and decisions
- Deblocking 4x4 block boundaries

Different technologies will be compared at a subjective viewing at the next meeting.

4 Tests specification

Test#	Description	Document#
11.1.1	Very strong deblocking filtering with conditional activation signalling	JVET-L0523
11.1.2	Extended deblocking filter for luma and chroma	JVET-L0393
11.1.3	Long deblocking filters (only luma)	JVET-L0102
11.1.4	CE11: Test results of CE11.1.5 long-tap deblocking filter	JVET-L0062
11.1.5	Longer tap Luma deblocking filter	JVET-L0224
11.1.6	Combination of JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140. JVET-L0337, JVET-L0572, JVET-L0072	JVET-L0403 JVET-L0327 JVET-L0405 JVET-L0140 JVET-L0337 JVET-L0572 JVET-L0072
11.1.7	Combination of JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140. JVET-L0337, JVET-L0572, JVET-L0072 with 6 CTU line buffers	JVET-L0403 JVET-L0327 JVET-L0405 JVET-L0140 JVET-L0337 JVET-L0572 JVET-L0072
11.1.8	Combination of JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140. JVET-L0337, JVET-L0572, JVET-L0072 with 4 CTU line buffers	JVET-L0403 JVET-L0327 JVET-L0405 JVET-L0140 JVET-L0337 JVET-L0572 JVET-L0072

5 CE 11.1: long-tap deblocking filter

Proposals in this sub-CE investigate longer deblocking filters. Some proposals in this sub-CE only experiment with luma deblocking, while several proposals test changes in both luma and chroma deblocking.

Common aspect of most proposals is applying longer deblocking filters for blocks of larger sizes. By larger block sizes >=32 samples in the direction orthogonal to the block boundary is usually understood. Some proposals differentiate between blocks of size 16 and >=32 with longer filters applied to larger blocks. Another aspect different among the proposals is applying the same or different length of deblocking filters to the sides of the block boundary. The decisions for choosing strong deblocking filtering (in addition to large block sizes) are studied.

Test	Proponent(s)	Cross-checker(s)
CE11.1.1	Christian Helmrich christian.helmrich@hhi.fraunhofer.de JVET-L0523	Kenneth Andersson kenneth.r.andersson@ericsson.com
CE11.1.2	Kei Kawamura	TBD

	ki-kawamura@kddi.com JVET-L0393	
CE11.1.3	Chia-Ming Tsai chia-ming.tsai@mediatek.com JVET-L0102	TBD
CE11.1.4	Woong IL Choi woongil.choi@samsung.com JVET-L0062	TBD
CE11.1.5	Anand Meher Kotra Anand.meher.kotra@huawei.com JVET-L0224	Kei Kawamura ki-kawamura@kddi.com
CE11.1.6	Kenneth Andersson kenneth.r.andersson@ericsson.com Masaru Ikeda Masaru.ikeda@sony.com Dmytro Rusanovskyy dmytror@qti.qualcomm.com Kiran Misra misrak@sharplabs.com JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140, JVET-L0337, JVET-L0572, JVET-L0072	Christian Helmrich christian.helmrich@hhi.fraunhofer.de
CE11.1.7 (based on CE11.1.6 with 6 CTU line buffers)	Kenneth Andersson kenneth.r.andersson@ericsson.com Masaru Ikeda Masaru.ikeda@sony.com Dmytro Rusanovskyy dmytror@qti.qualcomm.com Kiran Misra misrak@sharplabs.com JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140, JVET-L0337, JVET-L0572, JVET-L0072	TBD
CE11.1.8 (based on CE11.1.6 with 4 CTU line buffers)	Kenneth Andersson kenneth.r.andersson@ericsson.com Masaru Ikeda Masaru.ikeda@sony.com Dmytro Rusanovskyy dmytror@qti.qualcomm.com Kiran Misra misrak@sharplabs.com JVET-L0403, JVET-L0327, JVET-	TBD

	L0405, JVET-L0140, JVET-L0337, JVET-L0572, JVET-L0072	

5.1 Description of technologies in CE11.1

CE11.1.1 Very strong deblocking filtering with conditional activation signaling JVET-L0523

The following is proposed to be tested:

1. selective signaling of an in-loop filter control parameter (FCP) per CTU to *allow* the application of very strong deblocking filtering; this FCP shall only be signaled in a CTU if that CTU is partitioned into fewer than *A* sub-units (TUs) and if residual coefficient coding is used in at least *B* of the TUs.
2. use of a very strong deblocking design, with a filter kernel size of 16 taps, for large transform units. This "superstrong" deblocking is *disallowed* in CTUs for which no FCP is signaled according to 1.

Details including the encoder decision are provided in JVET-L0523.

CE11.1.2 JVET-L0393

[1] Technology to be investigated

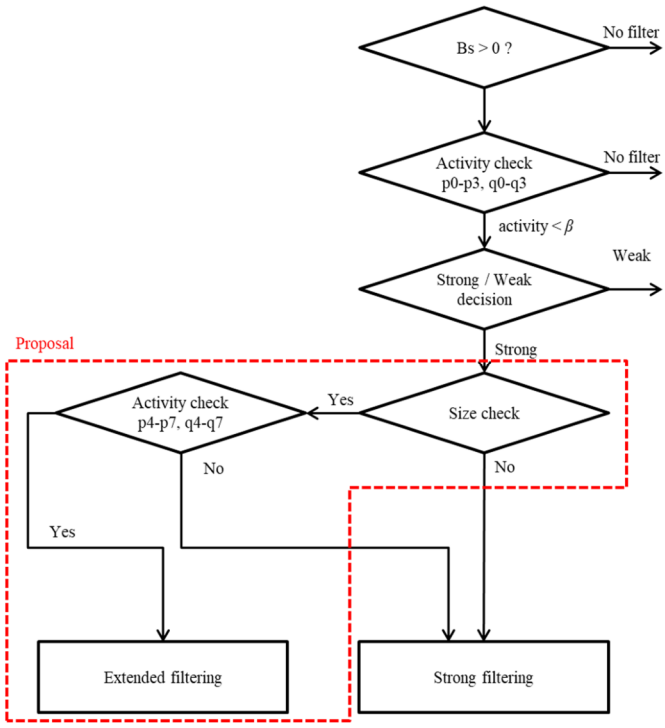


Figure 11.1.2.1 Decision process for luma.

Figure 11.1.2.1 shows proposed filter decision process for luma. The processes which are bounded by red dashed line are added to the current process. It is presented that details of added processes in below.

- Size check

When the following condition is true, the proposed decision process is proceeded to “Yes”;

- Edge boundary is located between two blocks of which cross edge sizes are Th or longer than Th,

Th is defined as follows.

- If (Bs == 2), then Th = 16
- else if (Bs == 1), then Th = 32

It is noted that a block size is replaced by an edge size. Direction of edge size depends on that of edge boundary like EDGE_VER or EDGE_HOR.

- Activity check p4-p7, q4-q7

  At first, the following condition (1) and (2) are checked.

$$|p7_0 - p0_0| + |q7_0 - q0_0| < \frac{\beta}{8} \quad (1)$$

$$|p7_3 - p0_3| + |q7_3 - q0_3| < \frac{\beta}{8} \quad (2)$$

  If both (1) and (2) are true, then following conditions (3) and (4) are checked respectively at block P and block Q. If (3) is true, extended filter is applied to block P. In the same manner, if (4) is true, extended filter is applied to block Q.

$$|\boldsymbol{p7_0 - p4_0}| + |\boldsymbol{p7_3 - p4_3}| < \frac{\boldsymbol{\beta}}{\boldsymbol{8}} \quad (\boldsymbol{3})$$

$$|\boldsymbol{q7_0 - q4_0}| + |\boldsymbol{q7_3 - q4_3}| < \frac{\boldsymbol{\beta}}{\boldsymbol{8}} \quad (\boldsymbol{4})$$

- Extended filtering

In extended filtering process, following filter kernels are used.

p6 = 3,5,2,1,1,1,1,1,1,0,0,0,0,0,0,0
p5 = 1,2,5,2,1,1,1,1,1,1,0,0,0,0,0,0
p4 = 0,1,3,3,3,1,1,1,1,1,1,0,0,0,0,0
p3 = 0,1,2,2,2,2,2,1,1,1,1,1,0,0,0,0
p2 = 0,0,1,2,2,2,2,2,2,1,1,1,0,0,0,0
p1 = 0,0,1,1,2,2,2,2,2,2,1,1,0,0,0,0
p0 = 0,0,0,1,2,2,2,2,2,2,2,1,0,0,0,0

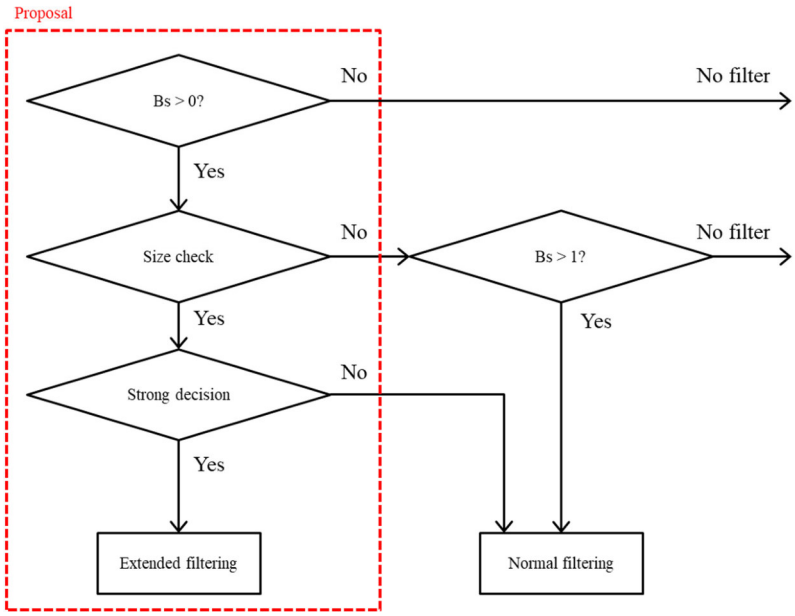


Figure 11.1.2.2 Decision process for chroma.

Figure 11.1.2.2 shows proposed filter decision process for chroma. The processes which are bounded by red dashed line are added to the current process. It is presented that details of added processes in below.

- Size check

  When the following condition is true, the proposed decision process is proceeded to “Yes”;

  - Edge boundary is located between two blocks of which cross edge sizes are Th or longer than Th,

  Th is defined as follows.

  - If (Bs == 2), then Th = 8
  - else if (Bs == 1), then Th = 32

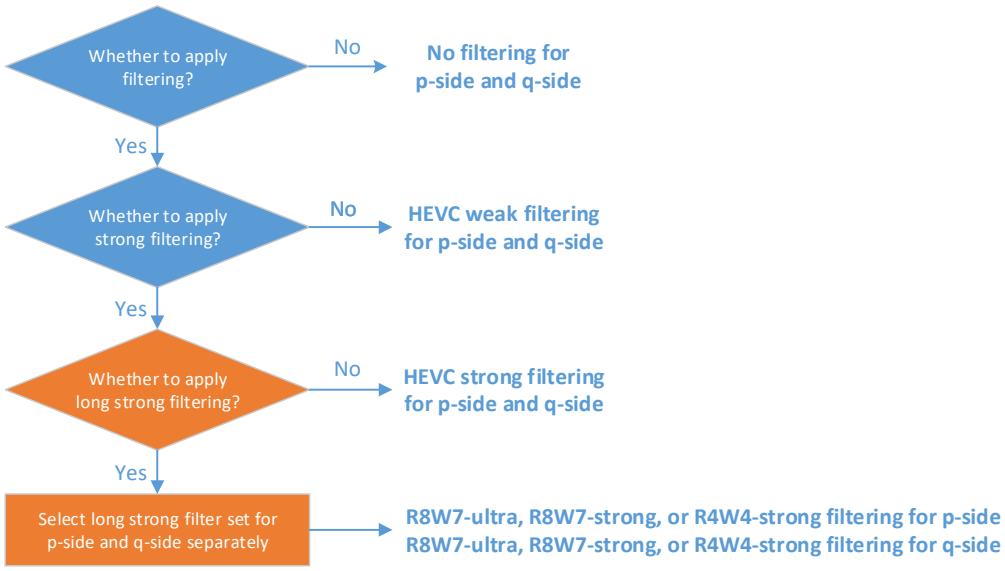
  It is noted that a block size is replaced by an edge size. Direction of edge size depends on that of edge boundary like EDGE_VER or EDGE_HOR.

- Strong decision and Extended filtering

  Those are exact same procedure as strong filtering for luma.

CE11.1.3 JVET-L0102

HEVC deblocking filters are extended with longer strong filters for luma. The flowchart of the proposed filtering decisions if shown below.



When all the following conditions are true, the proposed long strong filtering is applied. Otherwise, the HEVC strong filtering is applied for p-side and q-side

- The slice type is inter.
- The CB length at p-side and the CB length at q-side are either larger than or equal to 16.
- Both of the two inter-side difference values representing difference between p-side and q-side are larger than a threshold, *inter_side_diff_th*. The two inter-side difference values are defined as
  - $InterSideDiff_i = \left|\left(p_0^i + p_1^i\right) - \left(q_0^i + q_1^i\right) + 1\right| >> 1$, where $i = 0$ or 3.
- Both of the two intra-side difference values representing variation within p-side plus variation within q-side are smaller than a threshold, *intra_side_diff_th*. The two intra-side difference values are defined as

$$IntraSideDiff_i = \sum_{s=4}^{7} \left|p_s^i - p_3^i\right| + \sum_{s=4}^{7} \left|q_s^i - q_3^i\right|, \text{ where } i = 0 \text{ or } 3.$$

- Any one of adjacent block edges also satisfies all the above four conditions.

When the long strong filtering decision output is yes, the long strong filter set selection is activated to select one of the three long strong filter sets for p-side and one of the three long strong filter sets for q-side. The selection based on the p-side and q-side length of the current block edge is described as follows, where len_p and $leng_q$ denote the CB length at p-side and the CB length at q-side, respectively.

1) If "$len_p \geq 32$ and $len_q \geq 32$", R8W7-ultra or R8W7-strong filtering is selected as follows:
   1.1) If any one of adjacent block edges of the current block edge is also satisfied "$len_p \geq 32$ and $len_q \geq 32$", R8W7-ultra filtering is applied to both p-side and q-side of the current block edge.
   1.2) Otherwise (i.e., "$len_p \geq 32$ and $len_q \geq 32$" is only true for the current block edge), R8W7-strong filtering is applied to both p-side and q-side of the current block edge.
2) If "$len_p > 16$ and $len_q = 16$", R8W7-strong filtering is applied to p-side of the current block edge, and R4W4-strong filtering is applied to q-side of the current block edge.
3) If "$len_p = 16$ and $len_q > 16$", R4W4-strong filtering is applied to p-side of the current block edge, and R8W7-strong filtering is applied to q-side of the current block edge.
4) If "$len_p = 16$ and $len_q = 16$", R4W4-strong filtering is applied to both p-side and q-side of the current block edge.

The filtered values are clipped to stay in the bounds of $-2t_c$ to $2t_c$ of the original values, as in HEVC strong filtering.

If R8W7-ultra filtering is applied to p-side, filtered sample values from p'^i_6 to p'^i_0 are as follows:

$$p'^{i}_{6} = (10\ p^{i}_{7} + 8\,p^{i}_{6} + 2\,p^{i}_{5} + 2\,p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + q^{i}_{0} + q^{i}_{1} + 16) >> 5$$
$$p'^{i}_{5} = (8\,p^{i}_{7} + 4\,p^{i}_{6} + 6\,p^{i}_{5} + 2\,p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + q^{i}_{1} + q^{i}_{2} + 16) >> 5$$
$$p'^{i}_{4} = (6\,p^{i}_{7} + 3\,p^{i}_{6} + 3\,p^{i}_{5} + 6\,p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + q^{i}_{2} + q^{i}_{3} + 16) >> 5$$
$$p'^{i}_{3} = (4\,p^{i}_{7} + 3\,p^{i}_{6} + 3\,p^{i}_{5} + 3\,p^{i}_{4} + 5\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + 2q^{i}_{2} + q^{i}_{3} + q^{i}_{4} + 16) >> 5$$
$$p'^{i}_{2} = (3\,p^{i}_{7} + 2\,p^{i}_{6} + 2\,p^{i}_{5} + 3\,p^{i}_{4} + 3\,p^{i}_{3} + 5\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + 2q^{i}_{2} + 2q^{i}_{3} + q^{i}_{4} + q^{i}_{5} + 16) >> 5$$
$$p'^{i}_{1} = (2\,p^{i}_{7} + 2\,p^{i}_{6} + 2\,p^{i}_{5} + 2\,p^{i}_{4} + 2\,p^{i}_{3} + 3\,p^{i}_{2} + 5\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + 2q^{i}_{2} + 2q^{i}_{3} + 2q^{i}_{4} + q^{i}_{5} + q^{i}_{6} + 16) >> 5$$
$$p'^{i}_{0} = (p^{i}_{7} + 2\,p^{i}_{6} + 2\,p^{i}_{5} + 2\,p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 5\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + 2q^{i}_{2} + 2q^{i}_{3} + 2q^{i}_{4} + 2q^{i}_{5} + q^{i}_{6} + q^{i}_{7} + 16) >> 5$$

If R8W7-ultra filtering is applied to q-side, filtered sample values from q'^{i}_{6} to q'^{i}_{0} are as follows:

$$q'^{i}_{6} = (10\,q^{i}_{7} + 8q^{i}_{6} + 2q^{i}_{5} + 2q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + p^{i}_{0} + p^{i}_{1} + 16) >> 5$$
$$q'^{i}_{5} = (8q^{i}_{7} + 4q^{i}_{6} + 6q^{i}_{5} + 2q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + p^{i}_{1} + p^{i}_{2} + 16) >> 5$$
$$q'^{i}_{4} = (6q^{i}_{7} + 3q^{i}_{6} + 3q^{i}_{5} + 6q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + 16) >> 5$$
$$q'^{i}_{3} = (4q^{i}_{7} + 3q^{i}_{6} + 3q^{i}_{5} + 3q^{i}_{4} + 5q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + 2\,p^{i}_{2} + p^{i}_{3} + p^{i}_{4} + 16) >> 5$$
$$q'^{i}_{2} = (3q^{i}_{7} + 2q^{i}_{6} + 2q^{i}_{5} + 3q^{i}_{4} + 3q^{i}_{3} + 5q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + 2\,p^{i}_{2} + 2\,p^{i}_{3} + p^{i}_{4} + p^{i}_{5} + 16) >> 5$$
$$q'^{i}_{1} = (2q^{i}_{7} + 2q^{i}_{6} + 2q^{i}_{5} + 2q^{i}_{4} + 2q^{i}_{3} + 3q^{i}_{2} + 5q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + 2\,p^{i}_{2} + 2\,p^{i}_{3} + 2\,p^{i}_{4} + p^{i}_{5} + p^{i}_{6} + 16) >> 5$$

$$q'^{i}_{0} = (q^{i}_{7} + 2q^{i}_{6} + 2q^{i}_{5} + 2q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 5q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + 2\,p^{i}_{2} + 2\,p^{i}_{3} + 2\,p^{i}_{4} + 2\,p^{i}_{5} + p^{i}_{6} + p^{i}_{7} + 16) >> 5$$

If R8W7-strong filtering is applied to p-side, filtered sample values from p'^{i}_{6} to p'^{i}_{0} are as follows:

$$p'^{i}_{6} = (3\,p^{i}_{7} + 4\,p^{i}_{6} + 3\,p^{i}_{5} + 2\,p^{i}_{4} + p^{i}_{3} + p^{i}_{2} + p^{i}_{1} + p^{i}_{0} + 8) >> 4$$
$$p'^{i}_{5} = (p^{i}_{7} + 3\,p^{i}_{6} + 4\,p^{i}_{5} + 2\,p^{i}_{4} + p^{i}_{3} + p^{i}_{2} + p^{i}_{1} + p^{i}_{0} + q^{i}_{0} + 8) >> 4$$
$$p'^{i}_{4} = (p^{i}_{7} + 2\,p^{i}_{6} + 2\,p^{i}_{5} + 3\,p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + p^{i}_{1} + p^{i}_{0} + q^{i}_{0} + q^{i}_{1} + 8) >> 4$$
$$p'^{i}_{3} = (p^{i}_{6} + p^{i}_{5} + 2\,p^{i}_{4} + 3\,p^{i}_{3} + 3\,p^{i}_{2} + 2\,p^{i}_{1} + p^{i}_{0} + q^{i}_{0} + q^{i}_{1} + q^{i}_{2} + 8) >> 4$$
$$p'^{i}_{2} = (p^{i}_{5} + p^{i}_{4} + 2\,p^{i}_{3} + 3\,p^{i}_{2} + 3\,p^{i}_{1} + 2\,p^{i}_{0} + q^{i}_{0} + q^{i}_{1} + q^{i}_{2} + q^{i}_{3} + 8) >> 4$$
$$p'^{i}_{1} = (p^{i}_{5} + p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + q^{i}_{2} + q^{i}_{3} + 8) >> 4$$
$$p'^{i}_{0} = (p^{i}_{4} + 2\,p^{i}_{3} + 2\,p^{i}_{2} + 2\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + q^{i}_{2} + q^{i}_{3} + q^{i}_{4} + 8) >> 4$$

If R8W7-strong filtering is applied to q-side, filtered sample values from q'^{i}_{6} to q'^{i}_{0} are as follows:

$$q'^{i}_{6} = (3q^{i}_{7} + 4q^{i}_{6} + 3q^{i}_{5} + 2q^{i}_{4} + q^{i}_{3} + q^{i}_{2} + q^{i}_{1} + q^{i}_{0} + 8) >> 4$$
$$q'^{i}_{5} = (q^{i}_{7} + 3q^{i}_{6} + 4q^{i}_{5} + 2q^{i}_{4} + q^{i}_{3} + q^{i}_{2} + q^{i}_{1} + q^{i}_{0} + p^{i}_{0} + 8) >> 4$$
$$q'^{i}_{4} = (q^{i}_{7} + 2q^{i}_{6} + 2q^{i}_{5} + 3q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + q^{i}_{1} + q^{i}_{0} + p^{i}_{0} + p^{i}_{1} + 8) >> 4$$
$$q'^{i}_{3} = (q^{i}_{6} + q^{i}_{5} + 2q^{i}_{4} + 3q^{i}_{3} + 3q^{i}_{2} + 2q^{i}_{1} + q^{i}_{0} + p^{i}_{0} + p^{i}_{1} + p^{i}_{2} + 8) >> 4$$
$$q'^{i}_{2} = (q^{i}_{5} + q^{i}_{4} + 2q^{i}_{3} + 3q^{i}_{2} + 3q^{i}_{1} + 2q^{i}_{0} + p^{i}_{0} + p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + 8) >> 4$$
$$q'^{i}_{1} = (q^{i}_{5} + q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + 8) >> 4$$
$$q'^{i}_{0} = (q^{i}_{4} + 2q^{i}_{3} + 2q^{i}_{2} + 2q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + p^{i}_{4} + 8) >> 4$$

If R4W4-strong filtering is applied to p-side, filtered sample values from p'^{i}_{3} to p'^{i}_{0} are as follows:

$$p'^{i}_{3} = (5\,p^{i}_{3} + 4\,p^{i}_{2} + 3\,p^{i}_{1} + p^{i}_{0} + q^{i}_{0} + q^{i}_{1} + q^{i}_{2} + 8) >> 4$$
$$p'^{i}_{2} = (3\,p^{i}_{3} + 4\,p^{i}_{2} + 3\,p^{i}_{1} + 2\,p^{i}_{0} + q^{i}_{0} + q^{i}_{1} + q^{i}_{2} + q^{i}_{3} + 8) >> 4$$
$$p'^{i}_{1} = (2\,p^{i}_{3} + 3\,p^{i}_{2} + 3\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + q^{i}_{2} + q^{i}_{3} + 8) >> 4$$
$$p'^{i}_{0} = (2\,p^{i}_{3} + 2\,p^{i}_{2} + 3\,p^{i}_{1} + 2\,p^{i}_{0} + 2q^{i}_{0} + 2q^{i}_{1} + 2q^{i}_{2} + q^{i}_{3} + 8) >> 4$$

If R4W4-strong filtering is selected at q-side, filtered sample values from q'^{i}_{3} to q'^{i}_{0} are as follows:

$$q'^{i}_{3} = (5q^{i}_{3} + 4q^{i}_{2} + 3q^{i}_{1} + q^{i}_{0} + p^{i}_{0} + p^{i}_{1} + p^{i}_{2} + 8) >> 4$$
$$q'^{i}_{2} = (3q^{i}_{3} + 4q^{i}_{2} + 3q^{i}_{1} + 2q^{i}_{0} + p^{i}_{0} + p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + 8) >> 4$$
$$q'^{i}_{1} = (2q^{i}_{3} + 3q^{i}_{2} + 3q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + 8) >> 4$$
$$q'^{i}_{0} = (2q^{i}_{3} + 2q^{i}_{2} + 3q^{i}_{1} + 2q^{i}_{0} + 2\,p^{i}_{0} + 2\,p^{i}_{1} + 2\,p^{i}_{2} + p^{i}_{3} + 8) >> 4$$

Within this CE, the performance of the deblocking method described in JVET-K0232, the filter coefficients, and the samples used in long strong filtering are to be studied.

CE11.1.4 JVET-L0062

HEVC deblocking filters are extended with longer deblocking filters for luma. 7 samples on a side of the boundary is filtered when length orthogonal to block boundary is >= 16, strong/weak filter determines use strong filter, sum of absolute pixel difference for pixel p8 and p0 and q8 and q0 compared to a threshold. Only 3 samples are filtered for the upper block when filtering horizontal boundaries. Decision on 4 sample segments of an 8x8 grid.

Following filters are used for left block in vertical filtering:

p6	{3, 5, 2, 1, 1, 1, 1, 1, 1}	p7 ~ q0
p5	{1, 2, 5, 2, 1, 1, 1, 1, 1, 1}	p7 ~ q1
p4	{1, 3, 3, 3, 1, 1, 1, 1, 1, 1}	p6 ~ q2
p3	{1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1}	p6 ~ q3
p2	{1, 2, 2, 2, 2, 2, 2, 1, 1, 1}	p5 ~ q3
p1	{1, 1, 2, 2, 2, 2, 2, 2, 1, 1}	p5 ~ q3
p0	{1, 2, 2, 2, 2, 2, 2, 2, 1}	p4 ~ q3

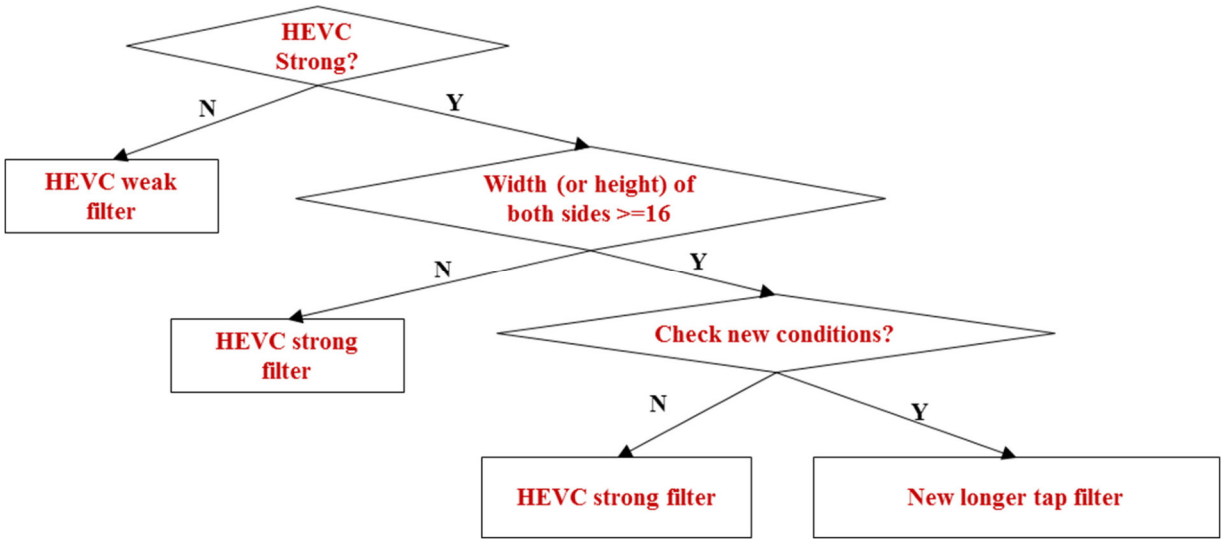
For the upper block in horizontal filtering only p0 to p2.

p2	{3, 5, 2, 2, 1, 1, 1, 1}	p3 ~ q3
p1	{2, 2, 4, 2, 2, 2, 1, 1}	p3 ~ q3
p0	{2, 2, 2, 2, 2, 2, 2, 2}	p3 ~ q3

CE11.1.5 JVET-L0224

New luma longer tap filters for large blocks are proposed.

The flowchart for the filtering decisions is as follows:



New luma longer tap filters:

Filter condition: HEVCstrongweak &&

$$(|p_0 - p_3 - p_4 + p_7| + |q_0 - q_3 - q_4 + q_7|) < \frac{3 * \beta}{32} \;\&\&$$
$$(|p_0 - p_2 - p_3 + p_5| + |q_0 - q_2 - q_3 + q_5|) < \frac{3 * \beta}{32}$$

Proposed filter coefficients for longer tap filter:

$p_6' = (6*p_7+3*p_6+p_5+p_4+p_3+p_2+ p_1+p_0+q_0+ 8) >> 4$
$p_5' = (5*p_7+p_6+3*p_5+p_4+p_3+p_2+p_1+p_0+q_0+q_1+ 8) >> 4$
$p_4' = (4*p_7+p_6+p_5+3*p_4+p_3+p_2+p_1+p_0+q_0+q_1+q_2+ 8) >> 4$
$p_3' = (3*p_7+p_6+p_5+p_4+3*p_3+p_2+p_1+p_0+q_0+q_1+q_2+q_3+ 8) >> 4$
$p_2' = (2*p_7+p_6+p_5+p_4+p_3+3*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (p_7+p_6+p_5+p_4+p_3+p_2+3*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (p_7+p_6+p_5+p_4+p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$
$q_6' = (6*q_7+3*q_6+q_5+q_4+q_3+q_2+ q_1+q_0+p_0+ 8) >> 4$
$q_5' = (5*q_7+q_6+3*q_5+q_4+q_3+q_2+q_1+q_0+p_0+p_1+ 8) >> 4$
$q_4' = (4*q_7+q_6+q_5+3*q_4+q_3+q_2+q_1+q_0+p_0+p_1+p_2+ 8) >> 4$
$q_3' = (3*q_7+q_6+q_5+q_4+3*q_3+q_2+q_1+q_0+p_0+p_1+p_2+p_3+ 8) >> 4$
$q_2' = (2*q_7+q_6+q_5+q_4+q_3+3*q_2+q_1+q_0+p_0+p_1+p_2+p_3+p_4+8) >> 4$
$q_1' = (q_7+q_6+q_5+q_4+q_3+q_2+3*q_1+q_0+p_0+p_1+p_2+p_3+p_4+p_5+8) >> 4$
$q_0' = (q_7+q_6+q_5+q_4+q_3+q_2+q_1+2*q_0+p_0+p_1+p_2+p_3+p_4+p_5+p_6+8) >> 4$

Proposed longer tap filter condition for horizontal boundaries overlapping with CTU boundaries:

HEVCstrongweak &&

$$(|p_3 - p_2 - p_1 + p_0| + |q_0 - q_3 - q_4 + q_7|) < \frac{3 * \beta}{32} \;\&\&$$
$$(|p_3 - p_2 - p_1 + p_0| + |q_0 - q_2 - q_3 + q_5|) < \frac{3 * \beta}{32}$$

Proposed longer tap filter coefficients for horizontal boundaries overlapping with CTU boundaries:

$p_2' = (6*p_3+3*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (5*p_3+p_2+3*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (5*p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$
$q_6' = (6*q_7+3*q_6+q_5+q_4+q_3+q_2+ q_1+q_0+p_0+ 8) >> 4$
$q_5' = (5*q_7+q_6+3*q_5+q_4+q_3+q_2+q_1+q_0+p_0+p_1+ 8) >> 4$
$q_4' = (4*q_7+q_6+q_5+3*q_4+q_3+q_2+q_1+q_0+p_0+p_1+p_2+ 8) >> 4$
$q_3' = (3*q_7+q_6+q_5+q_4+3*q_3+q_2+q_1+q_0+p_0+p_1+p_2+p_3+ 8) >> 4$
$q_2' = (2*q_7+q_6+q_5+q_4+q_3+3*q_2+q_1+q_0+p_0+p_1+p_2+2*p_3+8) >> 4$
$q_1' = (q_7+q_6+q_5+q_4+q_3+q_2+3*q_1+q_0+p_0+p_1+p_2+3*p_3+8) >> 4$
$q_0' = (q_7+q_6+q_5+q_4+q_3+q_2+q_1+2*q_0+p_0+p_1+p_2+4*p_3+8) >> 4$

CE11.1.6 Combination of JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140, JVET-L0337, JVET-L0572, JVET-L0072

Test CE11.1.13 modifications include: long filter deblocking of luma, long filter deblocking of chroma, and position dependent clipping.

Long deblocking filter for luma

The proposal uses a bilinear filter when samples at either one side of a boundary belong to a large block. A sample belonging to a large block is defined as when the width >= 32 for a vertical edge, and when height>=32 for a horizontal edge.

The bilinear filter is listed below.

Block boundary samples p_i and q_i for i=0 to S-1 are then replaced by linear interpolation as follows:

— $p_i' = (f_i * Middle_{s,t} + (64 - f_i) * P_s + 32) \gg 6), clipped\ to\ p_i \pm tcPD_i$

— $q_i' = (g_i * Middle_{s,t} + (64 - g_i) * Q_s + 32) \gg 6), clipped\ to\ q_i \pm tcPD_i$

where $tcPD_i$ term is a position dependent clipping described in Section 2.5 and f_i, $Middle_{s,t}$, P_s and Q_s are given below:

7, 7 (p side: 7, q side: 7)	$f_i = 59 - i * 9,\ can\ also\ be\ described\ as\ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $g_i = 59 - i * 9,\ can\ also\ be\ described\ as\ \boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $Middle_{7,7} = (2 * (p_o + q_o) + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1, \quad Q_7 = (q_6 + q_7 + 1) \gg 1$
7, 3 (p side: 7 q side: 3)	$f_i = 59 - i * 9,\ can\ also\ be\ described\ as\ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $g_i = 53 - i * 21,\ can\ also\ be\ described\ as\ \boldsymbol{g} = \{53,32,11\}$ $Middle_{7,3} = (2 * (p_o + q_o) + q_0 + 2 * (q_1 + q_2) + p_1 + q_1 + p_2 + p_3 + p_4 + p_5 + p_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1, \quad Q_3 = (q_2 + q_3 + 1) \gg 1$
3, 7 (p side: 3 q side: 7)	$g_i = 59 - i * 9,\ can\ also\ be\ described\ as\ \boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $f_i = 53 - i * 21,\ can\ also\ be\ described\ as\ \boldsymbol{f} = \{53,32,11\}$ $Middle_{3,7} = (2 * (q_o + p_o) + p_0 + 2 * (p_1 + p_2) + q_1 + p_1 + q_2 + q_3 + q_4 + q_5 + q_6 + 8) \gg 4$ $Q_7 = (q_6 + q_7 + 1) \gg 1, \quad P_3 = (p_2 + p_3 + 1) \gg 1$

Deblocking control for luma

The deblocking decision process is described in this sub-section.

Long luma filters are used only if all of the *Condition1*, *Condition2* and *Condition3* are TRUE.

The *Condition1* is the "large block condition". This condition detects whether the samples at P-side and Q-side belong to large blocks, which are represented by the variable bSidePisLargeBlk and bSideQisLargeBlk respectively. The bSidePisLargeBlk and bSideQisLargeBlk are defined as follows.

bSidePisLargeBlk = ((edge type is vertical and p_0 belongs to CU with width >= 32) | | (edge type is horizontal and p_0 belongs to CU with height >= 32))? TRUE: FALSE

bSideQisLargeBlk = ((edge type is vertical and q_0 belongs to CU with width >= 32) | | (edge type is horizontal and q_0 belongs to CU with height >= 32))? TRUE: FALSE

Based on bSidePisLargeBlk and bSideQisLargeBlk, the condition 1 is defined as follows.

Condition1 = (bSidePisLargeBlk || bSidePisLargeBlk) ? TRUE: FALSE

Next, if *Condition1* is true**,** the *Condition2* will be further checked. First, the following variables are derived:

dp0, dp3, dq0, dq3 are first derived as in HEVC

if (p side is greater than or equal to 32)

$dp0 = (dp0 + Abs(p_{5,0} - 2 * p_{4,0} + p_{3,0}) + 1) >> 1$

$dp3 = (dp3 + Abs(p_{5,3} - 2 * p_{4,3} + p_{3,3}) + 1) >> 1$

if (q side is greater than or equal to 32)

$dq0 = (dq0 + Abs(q_{5,0} - 2 * q_{4,0} + q_{3,0}) + 1) >> 1$

$dq3 = (dq3 + Abs(q_{5,3} - 2 * q_{4,3} + q_{3,3}) + 1) >> 1$

dpq0, dpq3, dp, dq, d are then derived as in HEVC.

Then the condition 2 is defined as follows.

Condition2 = (d < β) ? TRUE: FALSE

Finally, if both the *Condition1* and *Condition2* are valid, the proposed deblocking method will check the *Condition3* (the large block strong filter condition), which is defined as follows.

In the *Condition3* StrongFilterCondition, the following variables are derived:

dpq is derived as in HEVC.

$sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

if (p side is greater than or equal to 32)

$sp_3 = (sp_3 + Abs(p_7 - p_3) + 1) >> 1$

$sq_3 = Abs(q_0 - q_3)$, derived as in HEVC

if (q side is greater than or equal to 32)

$sq_3 = (sq_3 + Abs(q_7 - q_3) + 1) >> 1$

As in HEVC derive, StrongFilterCondition = (dpq is less than (β >> 2), $sp_3 + sq_3$ is less than (β >> 3), and $Abs(p_0 - q_0)$ is less than (5 * t_C + 1) >> 1) ? TRUE : FALSE

Long deblocking filter for chroma

The following strong deblocking filter for chroma is defined:

$p_2' = (3*p_3+2*p_2+p_1+p_0+q_0+4) >> 3$

$p_1' = (2*p_3+p_2+2*p_1+p_0+q_0+q_1+4) >> 3$

$p_0' = (p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+4) >> 3$

The proposed chroma filter performs deblocking on a 4x4 chroma sample grid.

Deblocking control for chroma

The chroma strong filters are used on both sides of the block boundary. Here, the chroma filter is selected when both sides of the chroma edge are greater than or equal to 8 (chroma position), and the following decision with three conditions are satisfied: The first one is for decision of boundary strength as well as large block. The proposed filter can be applied when the block width or height which orthogonally crosses the block edge is equal to or larger than 8 in chroma sample domain. The second and third one are basically the same as for HEVC luma decison, which are on/off decision and strong filter decision, respectively.

In the first decision, boundary strength (bS) is modified for chorma filtering as shown in Table 1. The condition in Table 1 are checked sequentially. If a condition is satisfied then the remaining conditions with lower priorities are skipped.

The modified boundary strength

Priority	Conditions	Y	U	V
5	At least one of the adjacent blocks is intra	2	2	2
4	At least one of the adjacent blocks has non-zero transform coefficients	1	1	1
3	Absolute difference between the motion vectors that belong to the adjacent blocks is greater than or equal to one integer luma sample	1	N/A	N/A
2	Motion prediction in the adjacent blocks refers to vectors is different	1	N/A	N/A
1	Otherwise	0	0	0

Chroma deblocking is performing when bS is equal to 2, or bS is equal to 1 when a large block boundary is detected.

The second and third condition is basically the same as HEVC luma strong filter decision as follows.

In the second condition:

d is then derived as in HEVC luma deblocking.

The second condition will be TRUE when d is less than β.

The test may include subsampling chroma gradient computation with threshold scaling as described in JVET-L0572

In the third condition StrongFilterCondition is derived as follows:

dpq is derived as in HEVC.

$sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

$sq_3 = Abs(q_0 - q_3)$, derived as in HEVC

As in HEVC derive, StrongFilterCondition = (dpq is less than (β >> 2), $sp_3 + sq_3$ is less than (β >> 3), and $Abs(p_0 - q_0)$ is less than (5 * t_C + 1) >> 1)

Position dependent clipping

The proposal also introduces a position dependent clipping tcPD which is applied to the output samples of the luma filtering process involving strong and long filters that are modifying 7 and 3 samples at the boundary. Assuming quantization error distribution, it is proposed to increase clipping value for samples which are expected to have higher quantization noise, thus expected to have higher deviation of the reconstructed sample value from the true sample value.

For each P or Q boundary, depending on the result of decision making process described in Section 2.2, position dependent threshold table is selected from Tc7 and Tc 3 tables that are provided to decoder as a side information:

tcPD = (SP == 3) ? Tc3 : Tc7;
tcQD = (SQ == 3) ? Tc3 : Tc7;

Following this, filtered p'_i and q'_i sample values are clipped according to tcP and tcQ clipping values:

$p''_i = clip3(p'_i + tcP_i, p'_i - tcP_i, p'_i);$
$q''_j = clip3(q'_j + tcQ_j, q'_j - tcQ_j, q'_j);$

where p'_i and q'_i are filtered sample values, p''_i and q''_j are output sample value after the clipping and tcP_i tcP_i are clipping thresholds that are derived from the VVC tc parameter and *tcPD* and *tcQD*. Term clip3 is a clipping function as it is specified in VVC.

CE11.1.7 Combination of JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140, JVET-L0337, JVET-L0572, JVET-L0072 with 6 CTU Line Buffers

The test makes use of the filtering and control processes of CE11.1.13. However, in this test the decoder would not access sample values beyond 6 and 3 CTU line buffers for luma and chroma, respectively. Filtering and control processes making use of samples at position beyond the CTU line buffers would be appropriately modified e.g. using padding / availability criteria / asymmetric filtering

CE11.1.8 Combination of JVET-L0403, JVET-L0327, JVET-L0405, JVET-L0140, JVET-L0337, JVET-L0572, JVET-L0072 with 4 CTU Line Buffers

The test makes use of the filtering and control processes of CE11.1.13. However, in this test the decoder would not access sample values beyond 4 and 2 CTU line buffers for luma and chroma, respectively. Filtering and control processes making use of samples at position beyond the CTU line buffers would be appropriately modified e.g. using padding / availability criteria / asymmetric filtering

6 CE 11.2: Deblocking at 4x4 block boundaries

The proposals in this CE study application of deblocking filtering to the 4x4 boundaries. Common feature of proposals studied in this CE is filtering on the block boundaries coinciding with the 4x4 sample grid and making sure it does not introduce dependency of deblocking for an edge on a parallel edge preceding in a filtering order. Test CE11.2.2 investigates whether filtering any edges adjacent to the blocks of size 4 can be avoided.

Test	Proponent(s)	Cross-checker(s)
CE11.2.1	Kenneth Andersson kenneth.r.andersson@ericsson.com JVET-L0073 Anand Meher Kotra Anand.meher.kotra@huawei.com JVET-L0225 Chia-Ming Tsai chia-ming.tsai@mediatek.com JVET-L0103	TBD
CE11.2.2	Hyeongmun Jang hm.jang@lge.com JVET-L0170	Kenneth Andersson kenneth.r.andersson@ericsson.com

6.1 Description of technologies in CE11.2

CE11.2.1 Combination of JVET-L0073, JVET-L0225, and JVET-L0170

In current VVC software large rectangular blocks of size Nx8 and Nx4 are not deblocked where N can be up to 64. This test applies deblocking on 4x4 grid to allow for deblocking of Nx4 blocks aligned with the current 8x8 grid but also for Nx8 blocks that not are aligned with the current grid. The number of samples to read and modify is limited to allow for parallel friendly processing. If at least one side of the block boundary has a length orthogonal to the block boundary equal to 4 only use the weak filter. Furthermore, only one sample is filtered on each side of the block boundary in this case.

CE11.2.2 Parallel deblocking filter JVET-L0170

Deblocking filter can be processed in parallel because it is not applied to 4x4 TU and PU boundaries. Within this CE, the performance of the parallel deblocking method described in JVET-D0044 is to be studied in terms of subjective quality and computational complexity.

For the luma component, deblocking filtering process is not applied on block boundary of 4xN block for vertical edge and Nx4 block for horizontal edge

For the chroma component, deblocking filtering process is not applied on block boundary of 2xN block for vertical edge and Nx2 block for horizontal edge.

7 Expected Gains

- Subjective picture quality improvement while preferably keeping rate-distortion performance.

- Coding efficiency improvement.

8 Evaluation Criteria

1. Measure impact on Bitrate/PSNR. The BD-rate is calculated for CTC and for CTC with ALF switched off relative to Anchor-1 and Anchor-2 respectively. The results for LDP configurations are mandatory for both CTC and CTC with ALF switched off.

2. Visual impact on output pictures of representative sequences. The subjective test is expected to be held to determine the visual impact. The subjective test will use 10 second versions of the sequences. The proponents shall provide the bitstreams and decoders for the sequences that are chosen for evaluation in the subjective test.

3. Complexity of the proposals evaluated in the CE shall be compared. In particular, the following aspects of the proposals shall be summarized:

- decoding/encoding time vs the anchor
- number of samples from the block boundary that can be modified by deblocking (may be per block size)
- number of samples from the block boundary that are needed for deblocking decisions (may be per block size)
- number of filtering operations (such as additions, multiplications, comparisons)
- number of line buffers
- parallelization capabilities, such as size of a unit of a picture that can be processed in parallel to other units.
- whether proposals increase the worst case complexity and if so, by how much.
- data should be provided for both luma and/or chroma if they are modified.

9 Subjective test(s)

There are two possibilities for performing a subjective test.

a) A subjective test will be conducted by the test chair before the Marrakesh meeting.

The following test sequences and conditions will be used in the subjective tests. The sequence length is 10 seconds. This means that the versions of some sequences encoded for the subjective test are longer than those used in the CTC [1].

The subjective test sequences shall be provided by December 7, 2018 (uploaded to the JVET ftp folder /ce/JVET-L1031_Deblocking), coding configuration with ALF switched off shall be used.

The anchors for the subjective viewing will be provided by A.M. Kotra.

Sequences for long-tap deblocking filters (CE11.1)

The following sequences and QP will be used for subjective evaluation in CE11.1:

Num	Name	Resolution	fps	Frames	Mode	QP
01	FoodMarket	3840x2160p	60fps	600	RA	39, 34
02	Campfire	3840x2160p	30fps	300	RA	39, 34
03	Kimono	1920x1080p	24fps	240	LDB	34, 39
04	KristenAndSara	1280x720p	60fps	600	LDP	34, 39
05	Red Kayak*	1920x1080p	30fps	300	RA	34, 39

* First 300 frames of the RedKayak sequence shall be used.

Sequences for 4x4 filtering (CE11.2)

The following sequences and QP will be used for subjective evaluation in CE11.2.

Num	Name	Resolution	fps	Frames	Mode	QP
01	Campfire	3840x2160p	30fps	300	RA	30, 34
02	Johnny	1280x720p	60fps	600	LDP	30, 34
03	Kimono	1920x1080p	24fps	240	LDB	30, 34
04	KristenAndSara	1280x720p	60fps	600	LDP	30, 34

b) Expert viewing at the Macao meeting may be performed.

The sequences listed in this section shall be provided at the Marrakesh meeting by the proponents for their corresponding sub-CEs (CE11.1 and CE11.2). The sequences coded with ALF switched on and ALF switched off.

9.1 Subjective test description:

The test material at resolution lower than 4K will be expanded to full screen by the player and evaluated together with the 4K test material.

10 Test conditions and data to be provided

The test conditions are according to CTC on VTM with ALF turned on and VTM with ALF turned off. In addition, the proponents need to provide results for additional QP and additional test sequences chosen by the CE (see above). The sequences length for the subjective viewing is 10 seconds. It was agreed to report information regarding what block sizes are, to which long-tap deblocking is applied. The complexity of the proposals according to Section 8 should be reported by the proponents.

11 Time-line and Responsibilities

2018-November-02:	Final CE description uploaded
2018-November-23:	Cross-checking begins, proponents provide software
2018-December-7:	Final version of CE software, full test results must be provided at this time.
	Sequences for the subjective test are uploaded to the ftp.

12 References

[1] “JVET common test conditions and software reference configurations”, JVET-L1010.
[2] “Algorithm description for Versatile Video Coding and Test Model 2 (VTM3)”, JVET-L1002.

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

Document: JVET-M0471

Title: CE11.1.6, CE11.1.7 and CE11.1.8: Joint proposals for long deblocking from Sony, Qualcomm, Sharp, Ericsson

Status: Input document to JVET

Purpose: Proposal

Author(s) or Contact(s):
M. Ikeda, T. Suzuki (Sony) Tel: +1-360-834-8054
D. Rusanovskyy, M.Karczewicz (Qualcomm) Email: masaru.ikeda@sony.com
W. Zhu, K. Misra, P. Cowan, A. Segall (Sharp) dmytror@qti.qualcomm.com
K. Andersson, J. Enhorn, Z. Zhang, R. Sjöberg (Ericsson) misrak@sharplabs.com
kenneth.r.andersson@ericsson.com

Source: Sony Corp, Qualcomm, Sharp Labs of America, Ericsson

Abstract

This contribution describes a parallel friendly long filter design for deblocking of large blocks common for CE11.1.6, CE11.1.7 and CE11.1.8. It further describes two variants of CTU line buffer reduction in CE11.1.7 and CE11.1.8 respectively. The CE tests are based on technologies that scored highly in subjective assessment carried out for CE11 before Macao meeting. It is asserted that all three tests give better subjective quality than the VTM-3.0 anchor.

The BD-Rate number for CE11.1.6 is Y/U/V reportedly shows 0.0%/-0.5%/-0.5% at AI, 0.0%/-0.8%/-0.8% at RA, 0.1%/-1.3% /-1.6% at LDB and 0.0%/-1.5%/-1.9% at LDP over VTM-3.0 with ALF switched on, respectively. When ALF is switched off, the BD-Rate results for Y/U/V reportedly shows 0.0%/-0.7%/-0.6 % at AI, 0.0%/-0.9%/ -0.9% at RA, 0.0%/-1.1%/-1.7% at LDB and 0.0%/-1.8%/-1.9% at LDP, respectively.

The BD-Rate number for CE11.1.7 is Y/U/V reportedly shows 0.0%/-0.5%/-0.5% at AI, 0.0%/-0.8%/-0.8% at RA, 0.1%/-1.4% /-1.5% at LDB and 0.0%/-1.5%/-1.9% at LDP over VTM-3.0 with ALF switched on, respectively. When ALF is switched off, the BD-Rate results for Y/U/V reportedly shows 0.0%/-0.7%/-0.6 % at AI, 0.0%/-0.9%/ -0.9% at RA, 0.0%/-1.3%/-1.6% at LDB and -0.1%/-1.9%/-2.0% at LDP, respectively.

The BD-Rate number for CE11.1.8 is Y/U/V reportedly shows 0.0%/-0.4%/-0.4% at AI, 0.0%/-0.7%/-0.7% at RA, 0.1%/-1.2% /-1.4% at LDB and 0.0%/-1.2%/-1.6% at LDP over VTM-3.0 with ALF switched on, respectively. When ALF is switched off, the BD-Rate results for Y/U/V reportedly shows 0.0%/-0.6%/-0.5 % at AI, 0.0%/-0.8%/ -0.8% at RA, 0.1%/-1.1%/-1.5% at LDB and 0.0%/-1.6%/-1.7% at LDP, respectively.

1 Introduction

Tests CE11.1.6, CE11.1.7 and CE11.1.8 [1] are based on a combination of high-scoring technologies [4][5][6][7] tested in CE11 subjective assessment [8]. The modifications common to all: long filter deblocking of luma, long filter deblocking of chroma, position dependent clipping and sub-block deblocking adjustment. CE11.1.7 and CE11.1.8 also includes modifications to address CTU line buffer restriction. Details of the design is described in section 2. The experimental results are shown in section 3 and the conclusion is provided at the end.

2 Proposed Scheme

We first describe CE11.1.6 in sections 2.1-2-6 which also is the basis for CE11.1.7 and CE11.1.8. CE11.1.7 and CE11.1.8 apply additional modifications related to CTU line buffer reduction and those are described in subsections 2.7 and 2.8 respectively.

2.1 Stronger deblocking filter for luma

The proposal uses a bilinear filter when samples at either one side of a boundary belong to a large block. A sample belonging to a large block is defined as when the width >= 32 for a vertical edge, and when height>=32 for a horizontal edge.

The bilinear filter is listed below.

Block boundary samples p_i for i=0 to Sp-1 and q_i for j=0 to Sq-1 are then replaced by linear interpolation as follows:

— $p_i' = (f_i * Middle_{s,t} + (64 - f_i) * P_s + 32) \gg 6), clipped\ to\ p_i \pm tcPD_i$

— $q_j' = (g_j * Middle_{s,t} + (64 - g_j) * Q_s + 32) \gg 6), clipped\ to\ q_j \pm tcPD_j$

where $tcPD_i$ and $tcPD_j$ term is a position dependent clipping described in Section 2.5 and g_j, f_i, $Middle_{s,t}$, P_s and Q_s are given below:

Sp, Sq 7, 7 (p side: 7, q side: 7)	$f_i = 59 - i * 9$, *can also be described as* $\boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $g_j = 59 - j * 9$, *can also be described as* $\boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $Middle_{7,7} = (2 * (p_o + q_o) + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1, \quad Q_7 = (q_6 + q_7 + 1) \gg 1$
7, 3 (p side: 7 q side: 3)	$f_i = 59 - i * 9$, *can also be described as* $\boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $g_j = 53 - j * 21$, *can also be described as* $\boldsymbol{g} = \{53,32,11\}$ $Middle_{7,3} = (2 * (p_o + q_o) + q_0 + 2 * (q_1 + q_2) + p_1 + q_1 + p_2 + p_3 + p_4 + p_5 + p_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1, \quad Q_3 = (q_2 + q_3 + 1) \gg 1$
3, 7 (p side: 3 q side: 7)	$g_j = 59 - j * 9$, *can also be described as* $\boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $f_i = 53 - i * 21$, *can also be described as* $\boldsymbol{f} = \{53,32,11\}$ $Middle_{3,7} = (2 * (q_o + p_o) + p_0 + 2 * (p_1 + p_2) + q_1 + p_1 + q_2 + q_3 + q_4 + q_5 + q_6 + 8) \gg 4$ $Q_7 = (q_6 + q_7 + 1) \gg 1, \quad P_3 = (p_2 + p_3 + 1) \gg 1$
7, 5 (p side: 7 q side: 5)	$g_j = 58 - j * 13$, *can also be described as* $\boldsymbol{g} = \{58,45,32,19,6\}$ $f_i = 59 - i * 9$, *can also be described as* $\boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $Middle7,5 = (2 * (p_o + q_o + p_1 + q_1) + q_2 + p_2 + q_3 + p_3 + q_4 + p_4 + q_5 + p_5 + 8) \gg 4$ $Q_5 = (q_4 + q_5 + 1) \gg 1, \quad P_7 = (p_6 + p_7 + 1) \gg 1$
5, 7 (p side: 5	$g_j = 59 - j * 9$, *can also be described as* $\boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $f_i = 58 - i * 13$, *can also be described as* $\boldsymbol{f} = \{58,45,32,19,6\}$

q side: 7)	$Middle5{,}7 = (2 * (q_o + p_o + p_1 + q_1) + q_2 + p_2 + q_3 + p_3 + q_4 + p_4 + q_5 + p_5 + 8) \gg 4$ $Q_7 = (q_6 + q_7 + 1) \gg 1, \quad P_5 = (p_4 + p_5 + 1) \gg 1$
5, 5 (p side: 5 q side: 5)	$g_j = 58 - j * 13$, *can also be described as* $\boldsymbol{g} = \{58,45,32,19,6\}$ $f_i = 58 - i * 13$, *can also be described as* $\boldsymbol{f} = \{58,45,32,19,6\}$ $Middle5{,}5 = (2 * (q_o + p_o + p_1 + q_1 + q_2 + p_2) + q_3 + p_3 + q_4 + p_4 + 8) \gg 4$ $Q_5 = (q_4 + q_5 + 1) \gg 1, \quad P_5 = (p_4 + p_5 + 1) \gg 1$
5, 3 (p side: 5 q side: 3)	$g_j = 53 - j * 21$, *can also be described as* $\boldsymbol{g} = \{53,32,11\}$ $f_i = 58 - i * 13$, *can also be described as* $\boldsymbol{f} = \{58,45,32,19,6\}$ $Middle5{,}3 = (q_o + p_o + p_1 + q_1 + q_2 + p_2 + q_3 + p_3 + 4) \gg 3$ $Q_3 = (q_2 + q_3 + 1) \gg 1, \quad P_5 = (p_4 + p_5 + 1) \gg 1$
3, 5 (p side: 3 q side: 5)	$g_j = 58 - j * 13$, *can also be described as* $\boldsymbol{g} = \{58,45,32,19,6\}$ $f_i = 53 - i * 21$, *can also be described as* $\boldsymbol{f} = \{53,32,11\}$ $Middle3{,}5 = (q_o + p_o + p_1 + q_1 + q_2 + p_2 + q_3 + p_3 + 4) \gg 3$ $Q_5 = (q_4 + q_5 + 1) \gg 1, \quad P_3 = (p_2 + p_3 + 1) \gg 1$

2.2 Deblocking control for luma

The deblocking decision process is described in this sub-section.

Wider-stronger luma filter is filters are used only if all of the **Condition1**, **Condition2** and **Condition 3** are TRUE.

The condition 1 is the "large block condition". This condition detects whether the samples at P-side and Q-side belong to large blocks, which are represented by the variable bSidePisLargeBlk and bSideQisLargeBlk respectively. The bSidePisLargeBlk and bSideQisLargeBlk are defined as follows.

bSidePisLargeBlk = ((edge type is vertical and p_0 belongs to CU with width >= 32) | | (edge type is horizontal and p_0 belongs to CU with height >= 32))? TRUE: FALSE

bSideQisLargeBlk = ((edge type is vertical and q_0 belongs to CU with width >= 32) | | (edge type is horizontal and q_0 belongs to CU with height >= 32))? TRUE: FALSE

Based on bSidePisLargeBlk and bSideQisLargeBlk, the condition 1 is defined as follows.

Condition1 = (bSidePisLargeBlk || bSidePisLargeBlk) ? TRUE: FALSE

Next, if **Condition 1** is true**,** the condition 2 will be further checked. First, the following variables are derived:

dp0, dp3, dq0, dq3 are first derived as in HEVC

if (p side is greater than or equal to 32)

dp0 = (dp0 + Abs($p_{5,0} - 2 * p_{4,0} + p_{3,0}$) + 1) >> 1

dp3 = (dp3 + Abs($p_{5,3} - 2 * p_{4,3} + p_{3,3}$) + 1) >> 1

if (q side is greater than or equal to 32)

dq0 = (dq0 + Abs($q_{5,0} - 2 * q_{4,0} + q_{3,0}$) + 1) >> 1

dq3 = (dq3 + Abs($q_{5,3} - 2 * q_{4,3} + q_{3,3}$) + 1) >> 1

dpq0, dpq3, dp, dq, d are then derived as in HEVC.

Then the condition 2 is defined as follows.

Condition2 = (d < β) ? TRUE: FALSE

If Condition1 and Condition2 are valid it is checked if any of the blocks uses sub-blocks:

```
If(bSidePisLargeBlk)
  If(mode block P == SUBBLOCKMODE)
     Sp =5
  else
     Sp =7
else
  Sp = 3
If(bSideQisLargeBlk)
    If(mode block Q == SUBBLOCKMODE)
       Sq =5
    else
       Sq =7
else
    Sq = 3
```

Finally, if both the **Condition 1** and **Condition 2** are valid, the proposed deblocking method will check the condition 3 (the large block Strong filter condition), which is defined as follows.

In the Condition3 StrongFilterCondition, the following variables are derived:

dpq is derived as in HEVC.

$sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

if (p side is greater than or equal to 32)

  if(Sp==5)

    $sp_3 = (sp_3 + Abs(p_5 - p_3) + 1) >> 1$

  else

    $sp_3 = (sp_3 + Abs(p_7 - p_3) + 1) >> 1$

$sq_3 = Abs(q_0 - q_3)$, derived as in HEVC

if (q side is greater than or equal to 32)

  If(Sq==5)

    $sq_3 = (sq_3 + Abs(q_5 - q_3) + 1) >> 1$

  else

    $sq_3 = (sq_3 + Abs(q_7 - q_3) + 1) >> 1$

As in HEVC derive, StrongFilterCondition = (dpq is less than ($\beta >> 2$), $sp_3 + sq_3$ is less than ($3*\beta >> 5$), and $Abs(p_0 - q_0)$ is less than ($5 * t_C + 1) >> 1$) ? TRUE : FALSE

2.3 Strong deblocking filter for chroma

The following strong deblocking filter for chroma is defined:

$p_2' = (3*p_3+2*p_2+p_1+p_0+q_0+4) >> 3$

$p_1' = (2*p_3+p_2+2*p_1+p_0+q_0+q_1+4) >> 3$

$p_0' = (p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+4) >> 3$

The proposed chroma filter performs deblocking on a 4x4 chroma sample grid.

2.4 Deblocking control for chroma

The chroma strong filters are used on both sides of the block boundary. Here, the chroma filter is selected when both sides of the chroma edge are greater than or equal to 8 (chroma position), and the following decision with three conditions are satisfied: The first one is for decision of boundary strength as well as large block. The proposed filter can be applied when the block width or height which orthogonally crosses the block edge is equal to or larger than 8 in chroma sample domain. The second and third one are basically the same as for HEVC luma decison, which are on/off decision and strong filter decision, respectively.

In the first decision, boundary strength (bS) is modified for chorma filtering as shown in Table 1. The condition in Table 1 are checked sequentially. If a condition is satisfied then the remaining conditions with lower priorities are skipped.

Table 1. The modified boundary strength

Priority	Conditions	Y	U	V
5	At least one of the adjacent blocks is intra	2	2	2
4	At least one of the adjacent blocks has non-zero transform coefficients	1	1	1
3	Absolute difference between the motion vectors that belong to the adjacent blocks is greater than or equal to one integer luma sample	1	N/A	N/A
2	Motion prediction in the adjacent blocks refers to vectors is different	1	N/A	N/A
1	Otherwise	0	0	0

Chroma deblocking is performing when bS is equal to 2, or bS is equal to 1 when a large block boundary is detected.

The second and third condition is basically the same as HEVC luma strong filter decision as follows.

In the second condition :

d is then derived as in HEVC luma deblocking.

The second condition will be TRUE when d is less than β.

In the third condition StrongFilterCondition is derived as follows:

dpq is derived as in HEVC.

$sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

$sq_3 = Abs(q_0 - q_3)$, derived as in HEVC

As in HEVC derive, StrongFilterCondition = (dpq is less than (β >> 2), $sp_3 + sq_3$ is less than (β >> 3), and $Abs(p_0 - q_0)$ is less than ($5 * t_C + 1$) >> 1)

2.5 Position dependent clipping

The proposal also introduces a position dependent clipping tcPD which is applied to the output samples of the luma filtering process involving strong and long filters that are modifying 7, 5 and 3 samples at the boundary. Assuming quantization error distribution, it is proposed to increase clipping value for samples which are expected to have higher quantization noise, thus expected to have higher deviation of the reconstructed sample value from the true sample value.

For each P or Q boundary filtered with proposed asymmetrical filter, depending on the result of decision making process described in Section 2.2, position dependent threshold table is selected from Tc7 and Tc3 tables that are provided to decoder as a side information:

```
Tc7 = { 6, 5, 4, 3, 2, 1, 1};
Tc3 = { 6, 4, 2 };
tcPD = (SP == 3) ? Tc3 : Tc7;
tcQD = (SQ == 3) ? Tc3 : Tc7;
```

For the P or Q boundaries being filtered with a short symmetrical filter, position dependent threshold of lower magnitude is applied:

```
Tc3 = { 3, 2, 1 };
```

Following defining the threshold, filtered p'_i and q'_i sample values are clipped according to tcP and tcQ clipping values:

$$p''_i = clip3(p'_i + tcP_i, p'_i - tcP_i, p'_i);$$
$$q''_j = clip3(q'_j + tcQ_j, q'_j - tcQ_j, q'_j);$$

where p'_i and q'_i are filtered sample values, p''_i and q''_j are output sample value after the clipping and tcP_i tcP_i are clipping thresholds that are derived from the VVC tc parameter and *tcPD* and *tcQD*. Term clip3 is a clipping function as it is specified in VVC.

2.6 Sub-block deblocking adjustment

To enable parallel friendly deblocking using both long filters and sub-block deblocking the long filters is restricted to modify at most 5 samples on a side that uses sub-block deblocking (AFFINE or ATMVP) as shown in the luma control for long filters. Additionally the sub-block deblocking is adjusted such that that sub-block boundaries on an 8x8 grid that are close to a CU or an implicit TU boundary is restricted to modify at most two samples on each side.

Following applies to sub-block boundaries that not are aligned with the CU boundary.

```
If(mode block Q == SUBBLOCKMODE && edge!=0){
  if (!(implicitTU && (edge == (64 / 4))))
    if (edge == 2 || edge == (orthogonalLength - 2) || edge == (56 / 4) || edge == (72 / 4))
      Sp = Sq = 2;
    else
      Sp = Sq = 3;
  else
    Sp = Sq = bSideQisLargeBlk ? 5:3
}
```

Where edge equal to 0 corresponds to CU boundary, edge equal to 2 or equal to orthogonalLength-2 corresponds to sub-block boundary 8 samples from a CU boundary etc. Where implicit TU is true if implicit split of TU is used.

2.7 CE11.1.7 Restriction to 6 CTU/3CTU line buffers for luma/chroma

Filtering of horizontal boundary is limiting Sp=5 for luma, Sp=3 for chroma when the horizontal boundary is aligned with the CTU boundary.

In chroma, the reference sample for p side is limited up to three samples away from the boundary in case of the CTU boundary to reduce CTU line buffers. Accordingly, the StrongFilterCondition and chroma filter are modified by padding as follows.

The sp3 derivation is modified by padding of p_2 sample instead of p_3:

$$sp_3 = Abs(p_2 - p_0)$$

The chroma filter for p_2 is modified by padding of p_2 sample instead of p_3:

$$p_2' = (5*p_2+p_1+p_0+q_0+4) >> 3$$

2.8 CE11.1.8 Restriction to 4CTU/2CTU line buffers for luma/chroma

Filtering of horisontal boundary is limiting Sp= 3 for luma, Sp=1 and Sq=1 for chroma, when the horisontal boundary is aligned with the CTU boundary.

3 Experimental Results

The following two conditions are evaluated: 1) VTM-3.0 with ALF switched on 2) VTM-3.0 with ALF switched off. LDP results are provided in addition to CTC .[2]

3.1 CE11.1.6

Table 1 Coding Performance

	All Intra Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.08%	-2.17%	-1.50%	96%	98%	-0.02%	-1.32%	-1.01%		99%
Class A2	-0.01%	-0.21%	-0.19%	93%	102%	-0.01%	-0.18%	-0.20%		99%
Class B	-0.01%	-0.47%	-0.40%	87%	103%	-0.01%	-0.44%	-0.37%		101%
Class C	-0.01%	-0.21%	-0.28%	97%	103%	0.00%	-0.22%	-0.29%		100%
Class E	-0.08%	-0.80%	-0.96%	90%	100%	-0.07%	-0.66%	-0.75%		97%
Overall	-0.03%	-0.71%	-0.61%	92%	101%	-0.02%	-0.53%	-0.49%		99%
Class D	0.00%	-0.13%	-0.18%	95%	103%	0.00%	-0.14%	-0.18%		103%
Class F	-0.01%	-0.18%	-0.29%	93%	103%	-0.01%	-0.18%	-0.30%		101%

	Random Access Main 10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.20%	-2.20%	-2.15%	84%	102%	-0.06%	-1.79%	-1.88%		100%
Class A2	-0.01%	-0.44%	-0.40%	85%	102%	-0.02%	-0.58%	-0.45%		103%
Class B	0.06%	-0.77%	-0.84%	85%	103%	0.04%	-0.74%	-0.84%		103%
Class C	0.00%	-0.35%	-0.36%	84%	103%	0.01%	-0.32%	-0.29%		101%
Class E										
Overall	-0.02%	-0.88%	-0.89%	84%	103%	0.00%	-0.80%	-0.82%		102%
Class D	0.04%	-0.21%	-0.33%	85%	101%	-0.02%	-0.28%	-0.32%		102%

Class F	0.00%	-0.22%	-0.29%	90%	101%	0.00%	-0.16%	-0.17%		102%

	Low delay B Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.11%	-1.65%	-2.11%	93%	103%	0.14%	-1.89%	-2.08%		102%
Class C	-0.02%	-0.23%	-0.77%	90%	104%	0.02%	-0.25%	-0.33%		101%
Class E	-0.02%	-1.41%	-2.42%	81%	101%	0.26%	-1.80%	-2.40%		101%
Overall	0.04%	-1.12%	-1.74%	89%	103%	0.13%	-1.32%	-1.58%		102%
Class D	0.02%	-0.67%	-0.91%	95%	102%	-0.01%	-0.48%	-0.68%		103%
Class F	-0.04%	0.68%	-0.36%	85%	105%	0.01%	0.55%	0.54%		101%

	Low delay P Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	-0.04%	-2.30%	-2.63%	80%	103%	0.00%	-1.86%	-1.92%		101%
Class C	0.01%	-0.38%	-0.71%	78%	101%	-0.03%	-0.25%	-0.84%		99%
Class E	-0.06%	-2.72%	-2.42%	88%	102%	-0.08%	-2.71%	-3.21%		106%
Overall	-0.02%	-1.77%	-1.94%	81%	102%	-0.03%	-1.54%	-1.88%		102%
Class D	-0.05%	-0.72%	-0.58%	80%	105%	-0.02%	-0.56%	-0.66%		102%
Class F	0.01%	-0.01%	-0.22%	84%	105%	0.02%	0.06%	-0.36%		103%

3.2 CE11.1.7

	All Intra Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.08%	-2.18%	-1.51%	105%	98%	-0.02%	-1.34%	-1.02%	98%	98%
Class A2	-0.01%	-0.22%	-0.20%	104%	98%	-0.01%	-0.19%	-0.20%	99%	99%
Class B	-0.01%	-0.47%	-0.41%	108%	98%	-0.01%	-0.45%	-0.38%	101%	99%
Class C	-0.01%	-0.21%	-0.28%	110%	100%	0.00%	-0.23%	-0.30%	101%	99%
Class E	-0.08%	-0.81%	-0.97%	111%	100%	-0.07%	-0.67%	-0.76%	104%	100%
Overall	-0.03%	-0.71%	-0.62%	108%	99%	-0.02%	-0.54%	-0.50%	101%	99%
Class D	0.00%	-0.13%	-0.17%	111%	100%	0.00%	-0.14%	-0.17%	103%	99%
Class F	-0.01%	-0.19%	-0.29%	107%	100%	-0.01%	-0.19%	-0.31%	101%	101%

	Random Access Main 10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.20%	-2.22%	-2.16%	102%	102%	-0.07%	-1.88%	-1.90%	99%	101%
Class A2	-0.02%	-0.37%	-0.45%	100%	102%	-0.03%	-0.48%	-0.40%	100%	102%
Class B	0.04%	-0.71%	-0.79%	100%	102%	0.04%	-0.82%	-0.91%	103%	102%
Class C	-0.02%	-0.38%	-0.27%	100%	102%	0.00%	-0.27%	-0.31%	101%	103%
Class E										
Overall	-0.04%	-0.86%	-0.86%	100%	102%	-0.01%	-0.82%	-0.85%	101%	102%
Class D	0.00%	-0.26%	-0.29%	101%	102%	0.00%	-0.21%	-0.36%	101%	102%

Class F	0.00%	-0.21%	-0.30%	101%	103%	-0.02%	-0.26%	-0.26%	102%	103%

	Low delay B Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.10%	-1.69%	-2.05%	107%	104%	0.12%	-2.03%	-1.69%	103%	102%
Class C	-0.02%	-0.25%	-0.58%	100%	105%	0.03%	0.01%	-0.48%	101%	104%
Class E	0.03%	-2.12%	-2.20%	107%	106%	0.21%	-2.14%	-2.47%	100%	105%
Overall	0.04%	-1.32%	-1.60%	105%	105%	0.12%	-1.38%	-1.48%	102%	103%
Class D	-0.01%	-0.89%	-1.02%	104%	102%	-0.01%	-0.41%	-1.25%	101%	103%
Class F	0.07%	0.29%	0.00%	103%	103%	0.14%	0.59%	0.34%	101%	103%

	Low delay P Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	-0.02%	-2.27%	-2.69%	95%	103%	-0.01%	-1.74%	-1.85%	102%	102%
Class C	-0.04%	-0.48%	-0.93%	92%	105%	0.02%	-0.21%	-0.68%	101%	101%
Class E	-0.16%	-2.98%	-2.31%	99%	103%	-0.08%	-2.79%	-3.45%	100%	102%
Overall	-0.06%	-1.85%	-2.01%	95%	104%	-0.02%	-1.49%	-1.86%	101%	102%
Class D	-0.10%	-0.68%	-0.71%	99%	103%	-0.02%	-0.85%	-0.44%	101%	101%
Class F	0.06%	0.18%	-0.11%	100%	101%	0.03%	-0.27%	0.04%	101%	101%

3.3 CE11.1.8

	All Intra Main10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.07%	-1.80%	-1.22%	106%	107%	-0.02%	-1.08%	-0.80%	102%	102%
Class A2	-0.01%	-0.16%	-0.15%	105%	105%	0.00%	-0.14%	-0.15%	99%	101%
Class B	-0.01%	-0.38%	-0.33%	102%	105%	-0.01%	-0.36%	-0.30%	99%	101%
Class C	0.00%	-0.17%	-0.23%	102%	103%	0.00%	-0.19%	-0.25%	102%	105%
Class E	-0.07%	-0.64%	-0.78%	105%	105%	-0.07%	-0.49%	-0.58%	104%	105%
Overall	-0.03%	-0.58%	-0.50%	104%	105%	-0.02%	-0.43%	-0.39%	101%	103%
Class D	0.00%	-0.12%	-0.16%	100%	102%	0.00%	-0.12%	-0.16%	101%	104%
Class F	-0.01%	-0.14%	-0.23%	104%	103%	-0.01%	-0.14%	-0.24%	101%	104%

	Random Access Main 10									
	Over VTM-3.0 (ALF OFF)					Over VTM-3.0 (ALF ON)				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.17%	-1.99%	-1.85%	102%	103%	-0.05%	-1.56%	-1.52%	100%	100%
Class A2	-0.01%	-0.33%	-0.31%	103%	103%	-0.02%	-0.50%	-0.35%	97%	98%
Class B	0.04%	-0.61%	-0.70%	104%	103%	0.04%	-0.69%	-0.73%	102%	101%
Class C	-0.01%	-0.38%	-0.35%	102%	103%	0.01%	-0.27%	-0.25%	101%	104%
Class E										
Overall	-0.03%	-0.77%	-0.76%	103%	103%	0.00%	-0.71%	-0.68%	100%	101%

Class D	0.01%	-0.26%	-0.26%	100%	103%	0.01%	-0.28%	-0.27%	100%	103%
Class F	0.00%	-0.17%	-0.24%	102%	103%	0.00%	-0.18%	-0.17%	100%	103%

	Low delay B Main10									
	Over VTM-3.0 (ALF OFF)					**Over VTM-3.0 (ALF ON)**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.09%	-1.50%	-1.86%	101%	103%	0.12%	-1.55%	-1.50%	99%	105%
Class C	-0.04%	-0.20%	-0.52%	105%	103%	0.03%	-0.20%	-0.52%	97%	98%
Class E	0.11%	-1.76%	-2.16%	105%	102%	0.09%	-1.75%	-2.51%	100%	99%
Overall	0.05%	-1.13%	-1.49%	104%	103%	0.08%	-1.15%	-1.42%	99%	101%
Class D	0.00%	-0.69%	-0.90%	101%	103%	0.02%	-0.34%	-0.76%	103%	104%
Class F	0.12%	0.47%	-0.10%	104%	102%	0.11%	0.56%	0.35%	99%	101%

	Low delay P Main10									
	Over VTM-3.0 (ALF OFF)					**Over VTM-3.0 (ALF ON)**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	-0.03%	-2.06%	-2.21%	99%	103%	0.02%	-1.44%	-1.84%	100%	102%
Class C	0.02%	-0.50%	-0.69%	103%	103%	0.01%	-0.25%	-0.53%	100%	98%
Class E	-0.05%	-2.46%	-1.99%	106%	103%	-0.15%	-2.12%	-2.70%	102%	100%
Overall	-0.02%	-1.64%	-1.65%	102%	103%	-0.02%	-1.21%	-1.62%	101%	100%
Class D	-0.07%	-0.92%	-0.74%	101%	104%	-0.05%	-0.41%	-0.43%	102%	106%
Class F	-0.01%	0.63%	0.23%	104%	102%	0.15%	0.02%	0.20%	100%	100%

4 Conclusion

Subject to the preferred CTU line buffer configuration we recommend CE11.1.6 (8 CTU line buffers), CE11.1.7 (6 CTU line buffers) or CE11.1.8 (4 CTU line buffers) to be considered for adoption in next version of VTM and VVC Working Draft.

5 Reference

[1] A. Norkin, A. M. Kotra “Description of Core Experiment 11 (CE11): Deblocking”, JVET-L1031, October 2018, Macao, China.

[2] F. Bossen, J. Boyce, X. Li, V. Seregin, K. Sühring “JVET common test conditions and software reference configurations for SDR video”, JVET-L1010, October 2018, Macao, China.

[3] VTM 3.0, https://vcgit.hhi.fraunhofer.de/jvet/VVCSoftware_vtm.git

[4] D. Rusanovskyy, M.Karczewicz, “CE11: Test on long deblocking filtering from JVET-J0021/K0334 (CE11.1.4),” JVET-L0403, 12th Meeting: Macao, CN, 3–12 Oct. 2018

[5] M. Ikeda, T. Suzuki, “CE11: Long-tap deblocking filter for luma and chroma (CE11.1.6),” , JVET-L0327, 12th Meeting: Macao, CN, 3–12 Oct. 2018

[6] W. Zhu, K. Misra, P. Cowan, A. Segall, “CE11: Deblocking modifications for Large CUs both luma and chroma (Test 11.1.7a and CE11.1.7b),” JVET-L0405, 12th Meeting: Macao, CN, 3–12 Oct. 2018

[7] K. Andersson, Z. Zhang, R. Sjöberg, “CE11: Long deblocking filters for luma (CE11.1.1) and for both luma and chroma (CE11.1.9),” JVET-L0072, 12th Meeting: Macao, CN, 3–12 Oct. 2018

[8] V. Baroncini, A. Norkin, A. M. Kotra, “Subjective assessment of CE11 proposals,” JVET-L0611, 12th Meeting: Macao, CN, 3–12 Oct. 2018

6 Patent rights declaration(s)

Sony Corp may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).

Qualcomm may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).

Sharp Labs of America may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).

Ericsson may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).

Joint Video Experts Team (JVET)
of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11
12th Meeting: Macao, CN, 3–12 Oct. 2018

Document: JVET-L0031-v3

Title: **CE11: Summary Report on Deblocking**

Status: Input document to JVET

Purpose: Report

Author(s) or Contact(s): Andrey Norkin
Anand Meher Kotra

Tel:
Email: anorkin@netflix.com
anand.meher.kotra@huawei.com

Source: CE11 coordinators

Abstract

This contribution provides a summary report of Core Experiment 11 on deblocking filtering. Three categories of proposals are covered by this CE, split into three sub-tests. These sub-tests are 1) long-tap deblocking filters, 2) general aspects of deblocking filters, and 3) deblocking at 4x4 block boundaries.

The corresponding compression performance of each coding tool evaluated in CE11 is summarized in this contribution. In addition, answers to questions mentioned in [3], such as complexity of the proposals and cross-checking results are also provided.

The software basis for this CE was VTM-2.0.1. Configurations and test conditions in JVET-K1010 [1] for SDR sequences are used. Results for additional configuration with ALF turned off are also reported. For the subjective viewing, additional encodes with different QPs have been used.

1 Test conditions and evaluation criteria

The proposals have been encoded according to two test conditions, which correspond to two anchors.

Anchor-1 is VTM-2.0.1 according to the CTC [1].

Anchor-2 is VTM-2.0.1 with ALF switched off (other conditions are the same as in Anchor 2).

2 Tests specification

Test#	Description	Document#
11.1.1	CE11: Long deblocking filters for luma (CE11.1.1)	JVET-L0072
11.1.2	Extended Deblocking Filter (only luma)	JVET-L0380
11.1.3	Long deblocking filters (only luma)	JVET-L0102
11.1.4	Test on long deblocking filtering from JVET-J0021/K0334	JVET-L0403
11.1.5	CE11: Test results of CE11.1.5 long-tap deblocking filter	JVET-L0062
11.1.6	Long-tap deblocking filter for luma and chroma	JVET-L0327
11.1.7	Deblocking modifications for Large CUs both luma and chroma	JVET-L0405
11.1.8	Longer tap Luma deblocking filter	JVET-L0224
11.1.9	CE11: Long deblocking filters for both luma and chroma (CE11.1.9)	JVET-L0072

11.1.10	Combination of CE11.1.6 and CE11.1.7	JVET-L0140
11.1.11	Combination of CE11.1.1 and CE11.1.7	JVET-L0337
11.2.1	Higher precision modification for VVC deblocking filter	JVET-L0192
11.2.2	Luma-adaptive deblocking filter (QP offset change based on luma level)	JVET-L0414
11.3.1	Non-recursive deblocking of luma on 4x4 grid (CE11.3.1)	JVET-L0073
11.3.2	Deblocking of sub-block boundaries for luma (CE11.3.2)	JVET-L0074
11.3.3	CE2-related: Deblocking for 4 x N and N x 4 block boundaries	JVET-L0225
11.3.4	Parallel deblocking for 4 x N and N x 4 block boundaries	JVET-L0103
11.3.5	Parallel deblocking filter	JVET-L0170

3 CE 11.1: long-tap deblocking filter

Proposals in this sub-CE investigate longer deblocking filters. Some proposals in this sub-CE only experiment with luma deblocking, while several proposals test changes in both luma and chroma deblocking.

Common aspect of most proposals is applying longer deblocking filters for blocks of larger sizes. By larger block sizes >=32 samples in the direction orthogonal to the block boundary is usually understood. Some proposals differentiate between blocks of size 16 and >=32 with longer filters applied to larger blocks. Another aspect different among the proposals is applying the same or different length of deblocking filters to the sides of the block boundary. The decisions for choosing strong deblocking filtering (in addition to large block sizes) are studied.

Test	**Proponent(s)**	**Cross-checker(s)**
CE11.1.1	Kenneth Andersson kenneth.r.andersson@ericsson.com JVET-L0072	Masaru Ikeda Masaru.Ikeda@sony.com; D.Rusanovskyy dmytror@qti.qualcomm.com
CE11.1.2	Kei Kawamura ki-kawamura@kddi.com JVET-L0380	Woong Il Choi woongil.choi@samsung.com
CE11.1.3	Chia-Ming Tsai chia-ming.tsai@mediatek.com JVET-L0102	Hyeongmun Jang hm.jang@lge.com
CE11.1.4	Dmytro Rusanovskyy dmytror@qti.qualcomm.com JVET-L0403	Kenneth Andersson kenneth.r.andersson@ericsson.com
CE11.1.5	Woong IL Choi woongil.choi@samsung.com JVET-L0062	Kei Kawamura ki-kawamura@kddi.com
CE11.1.6	Masaru Ikeda masaru.ikeda@sony.com JVET-L0327	Kenneth Andersson kenneth.r.andersson@ericsson.com
CE11.1.7	Kiran Misra misrak@sharplabs.com JVET-L0405	Masaru Ikeda Masaru.Ikeda@sony.com Adam Wieckowski adam.wieckowski@hhi.fraunhofer.de

CE11.1.8	Anand Meher Kotra Anand.meher.kotra@huawei.com JVET-L0224	Patrice Onno patrice.onno@crf.canon.fr Woong Il Choi woongil.choi@samsung.com
CE11.1.9	Kenneth Andersson kenneth.r.andersson@ericsson.com JVET-L0072	Christian Helmrich christian.helmrich@hhi.fraunhofer.de
CE11.1.10	Masaru Ikeda masaru.ikeda@sony.com Kiran Misra misrak@sharplabs.com JVET-L0140	Anand Meher Kotra anand.meher.kotra@huawei.com
CE11.1.11	Kenneth Andersson kenneth.r.andersson@ericsson.com misrak@sharplabs.com JVET-L0337	Jie Zhao jie.zhao@lge.com

3.1 Complexity of technologies in CE11.1

3.1.1 Scope of proposals in CE11.1

Tests	Luma modified (Y/N)	Chroma modified (Y/N)
CE11.1.1	Y	N
CE11.1.2	Y	N
CE11.1.3	Y	N
CE11.1.4 S1	Y	N
CE11.1.4 S2	Y	N
CE11.1.5	Y	N
CE11.1.6	Y	Y
CE11.1.7 S1	Y	Y
CE11.1.7 S2	Y	Y
CE11.1.8	Y	N
CE11.1.9	Y	Y
CE11.1.10	Y	Y
CE11.1.11	Y	Y

3.1.2 Luma deblocking complexity (*CE11.1*)

Tests	Samples from block bound. modified	Samples from block bound. deblocking decision	Max num. oper for filtering per line (add/mult/compar/shift)	Max number of oper. for decision for 8-sample boundary (add/mult/compar/shift)	Num. line buffers
VTM2.0.1	3+3	4+4	56 (28/2/12/14)	94 per 8 line segment	4
CE11.1.1	7+7, 5+5	16	120 (46,24,28,22)	30 (15,0,13,2) per line for 32x32 blocks* 8 = + 240 op	8
CE11.1.2	7+7	7+7	246 (138, 12, 28, 68)	20(10,0,6,4) per 4 line segment* 2 segments = + 20 op	8
CE11.1.3	7+7, 7+4, 4+7, 4+4	16	266 (168, 34, 28, 36)	25 (12,0,11,2) per line	8

CE11.1.4 S1	4+4/4+3	6+6/6+4	123 (56/39/0/28)	32(20/0/2/10) per line*4 = + 120op	6
CE11.1.4 S2	4+4/4+3	6+6/6+4	123 (56/39/0/28)	32(20/0/2/10) per line* 4 = + 128 op	6
CE11.1.5	7+7, 3+7	16	190(124/12/0/54)	5(3/0/1/1) per line* 4 = + 20 op	VTM
CE11.1.6	7+7	16	146 (104,0,28,14)	48(25,0,14,9) per line* 4 =+ 192 op	8
CE11.1.7 S1	7+7, 7+3, 3+7	16, 12	97* (55, 0, 28, 14)	Worst case is same as HEVC. 20 (11, 0, 5, 4) * 4 = + 80 op	8
CE11.1.7 S2	7+7, 7+3, 3+7	16, 12	97* (55, 0, 28 14)	Worst case is same as HEVC. 20 (11, 0, 5, 4) per line*4 = + 80 op	8
CE11.1.8	7+7, 3 + 7 for hor. edges between CTUs	16	142(96/4/0/42)	20(14, 2,2,2) per line * 4 lines = + 80 op	VTM
CE11.1.9	7+7, 5+5	16	120 (46,24,28,22)	30 (15,0,13,2) per line for 32x32 blocks* 8 = + 240 op	8
CE11.1.10	7+7, 7+3, 3+7	16, 12	97* (55, 0, 28, 14)	Worst case is same as HEVC. 20 (11, 0, 5, 4) per line*4 = + 80 op	8
CE11.1.11	7+7, 7+3, 3+7	16, 12	120 (46,24,28,22)	Worst case is same as HEVC. 20 (11, 0, 5, 4) per line*4 = + 80 op	8

* Numbers obtained with taking into account certain possible optimizations

3.1.3 Chroma deblocking complexity (*CE11.1*)

Tests	**Samples from block bound. modified**	**Samples from block bound. for deblocking decision**	**Max num. oper for filtering per line (add/mult/compar /shift)**	**Max number of oper. for decision for 8-sample boundary (add/mult/compar/shift)**	**Num. line buffers**
VTM2.0.1	1+1	None	10 (6/0/2/2)	None	2
CE11.1.1	VTM	VTM	VTM	VTM	VTM
CE11.1.2	VTM	VTM	VTM	VTM	VTM
CE11.1.3	VTM	VTM	VTM	VTM	VTM
CE11.1.4 S1	VTM	VTM	VTM	VTM	VTM
CE11.1.4 S2	VTM	VTM	VTM	VTM	VTM
CE11.1.5	VTM	VTM	VTM	VTM	VTM
CE11.1.6	3+3	8	58(32,0,12,14)	17 (17, 0, 12, 5)/2 per line	4
CE11.1.7 S1	3+3	8	41 (21, 0, 12, 8)	17 (17, 0, 12, 5)/2 per line	4
CE11.1.7 S2	3+3	8	41 (21, 0, 12, 8)	17 (17, 0, 12, 5)/2 per line	4
CE11.1.8	VTM	VTM	VTM	VTM	VTM
CE11.1.9	3+3, 2+2, 2+1,1+2, 1+1	8	52	24 (9,0,13,2) per line	4
CE11.1.10	3+3	8	41 (21, 0, 12, 8)	17 (17, 0, 12, 5)/2 per line	4
CE11.1.11	3+3	8	41 (21, 0, 12, 8)	17 (17, 0, 12, 5)/2 per line	4

3.1.4 Parallel processing capabilities (*CE11.1*)

Tests	Min unit size in luma samples that can be processed separately
CE11.1.1	8x8 grid (boundary segment size 16 samples, min block sizes P16x16/Q16x16 for 5+5, P32x32/Q32x32 for 7+7)
CE11.1.2	8x8
CE11.1.3	8x8 grid
CE11.1.4 S1	4x4 grid, 8x8 block asymmetric filtering: filter support 4 for P8/Q8, filter support 6 for longer P/Q
CE11.1.4 S2	4x4 grid, 8x8 block asymmetric filtering: filter support 4 for P8/Q8, filter support 6 for longer P/Q
CE11.1.5	8x8 grid
CE11.1.6	8x8 grid
CE11.1.7 S1	8x8 grid
CE11.1.7 S2	8x8 grid
CE11.1.8	8 x 8 grid (longer tap filter only applied when both blocks sharing the edge have a size >=16 samples)
CE11.1.9	8x8 grid (Luma: boundary segment size 16 samples, min block sizes P16x16/Q16x16 for 5+5, P32x32/Q32x32 for 7+7; Chroma: boundary segment size 1 sample, min block sizes P2x4/Q2x4 for 1+1, P2x8/Q2x8 for 2+2, P2x16/Q2x16 for 3+3)
CE11.1.10	8x8 grid
CE11.1.11	8x8 grid

3.2 Objective results of technologies in CE 11.1

3.2.1 VTM 2.0.1 CTC (ALF on)

	AI					RA				
Test	Y	U	V	EncT (%)	DecT (%)	Y	U	V	EncT (%)	DecT (%)
CE11.1.1	0.0%	0.0%	0.0%	106%*	99%	0.0%	0.0%	-0.1%	99%	100%
CE11.1.2	0.2%	0.0%	0.0%	100%	101%	0.1%	0.0%	0.0%	100%	101%
CE11.1.3	0.0%	0.0%	0.0%	100%	101%	0.0%	0.0%	0.0%	100%	103%
CE11.1.4 S1	0.2%	0.0%	0.0%	100%	100%	-0.2%	-0.1%	-0.3%	99%	100%
CE11.1.4 S2	0.0%	0.0%	0.0%	99%	100%	-0.1%	-0.1%	-0.2%	99%	100%
CE11.1.5	0.0%	0.0%	0.0%	100%	100%	0.0%	0.0%	-0.1%	100%	100%
CE11.1.6	0.0%	-0.7%	-0.6%	100%	102%	-0.1%	-1.2%	-1.1%	100%	103%
CE11.1.7 S1	0.0%	1.0%	1.3%	102%	102%	0.2%	1.0%	1.1%	100%	102%
CE11.1.7 S2	0.0%	0.4%	0.4%	102%	102%	0.2%	0.3%	0.3%	100%	102%
CE11.1.8	0.0%	0.0%	0.0%	100%	102%	0.0%	0.0%	0.0%	99%	101%
CE11.1.9	0.0%	-0.6%	-0.6%	103%*	103%	0.0%	-1.8%	-2.1%	98%*	104%
CE11.1.10	0.0%	-0.7%	-0.6%	100%	100%	0.1%	-1.1%	-1.2%	99%	103%
CE11.1.11	0.0%	-0.4%	-0.4%	103%	102%	0.3%	-0.4%	-0.4%	96%	104%

	LD-B					LD-P				
Test	Y	U	V	EncT (%)	DecT (%)	Y	U	V	EncT (%)	DecT (%)

CE11.1.1	0.0%	0.0%	-0.1%	101%	100%	0.0%	0.1%	0.2%	102%*	101%
CE11.1.2	0.1%	-0.1%	-0.1%	100%	101%	0.1%	0.0%	0.2%	100%	101%
CE11.1.3	0.0%	-0.1%	-0.1%	100%	102%	0.0%	0.1%	0.1%	100%	102%
CE11.1.4 S1	-0.2%	-0.3%	-0.4%	99%	101%	-0.2%	-0.4%	-0.1%	100%	101%
CE11.1.4 S2	-0.1%	-0.2%	-0.4%	99%	101%	-0.1%	-0.3%	-0.1%	100%	102%
CE11.1.5	0.1%	0.0%	-0.2%	100%	100%	0.1%	0.0%	0.2%	100%	101%
CE11.1.6	-0.1%	-1.5%	-1.7%	96%	101%	-0.1%	-1.7%	-1.7%	100%	104%
CE11.1.7 S1	0.5%	0.5%	0.4%	100%	102%	0.3%	0.8%	0.8%	100%	102%
CE11.1.7 S2	0.4%	0.3%	0.0%	100%	102%	0.3%	0.3%	0.4%	102%	103%
CE11.1.8	0.1%	-0.1%	-0.3%	100%	100%	0.0%	0.1%	0.1%	100%	100%
CE11.1.9	-0.1%	-2.0%	-2.1%	101%	105%	-0.1%	-2.2%	-2.0%	100%	104%
CE11.1.10	0.3%	-1.5%	-1.4%	96%	103%	0.2%	-1.5%	-1.5%	100%	101%
CE11.1.11	0.4%	-0.2%	-0.1%	99%	101%	0.3%	0.1%	0.0%	100%	98%

3.2.2 VTM 2.0.1 ALF off

	AI					RA				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.1.1	0.0%	0.0%	0.0%	99%	99%	-0.1%	0.0%	0.0%	100%	101%
CE11.1.2	0.2%	0.0%	0.0%	99%	100%	0.1%	-0.1%	-0.1%	100%	102%
CE11.1.3	0.0%	0.0%	0.0%	100%	102%	0.0%	0.0%	0.0%	100%	103%
CE11.1.4 S1	0.1%	0.0%	0.0%	100%	101%	-0.3%	-0.2%	-0.3%	100%	101%
CE11.1.4 S2	0.0%	0.0%	0.0%	100%	99%	-0.1%	-0.1%	-0.2%	100%	101%
CE11.1.5	0.0%	0.0%	0.0%	100%	101%	-0.1%	0.0%	0.0%	101%	100%
CE11.1.6	0.0%	-0.8%	-0.7%	100%	103%	-0.1%	-1.4%	-1.2%	90%	103%
CE11.1.7 S1	0.0%	2.2%	2.7%	100%	106%	0.1%	1.4%	1.5%	101%	103%
CE11.1.7 S2	0.0%	0.7%	0.8%	100%	106%	0.1%	0.5%	0.5%	102%	103%
CE11.1.8	0.0%	0.0%	0.0%	99%	103%	-0.1%	0.0%	0.0%	99%	101%
CE11.1.9	0.0%	-0.6%	-0.6%	99%	103%	-0.1%	-2.2%	-2.4%	99%	106%
CE11.1.10	0.0%	-0.8%	-0.7%	114%	103%	0.0%	-1.4%	-1.3%	104%	102%
CE11.1.11	0.0%	-0.6%	-0.5%	100%	102%	0.2%	-0.5%	-0.4%	102%	103%

	LD-B					LD-P				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.1.1	0.1%	0.0%	0.0%	100%	101%	-0.1%	0.1%	-0.1%	101%	101%
CE11.1.2	0.1%	-0.1%	-0.1%	100%	102%	0.0%	0.0%	-0.1%	100%	101%
CE11.1.3	0.0%	0.0%	0.0%	100%	102%	0.0%	0.0%	-0.1%	100%	102%
CE11.1.4 S1	-0.2%	-0.2%	-0.1%	100%	102%	-0.4%	-0.4%	-0.4%	100%	101%
CE11.1.4 S2	0.0%	-0.1%	-0.2%	100%	101%	-0.1%	-0.1%	-0.1%	100%	101%
CE11.1.5	0.1%	0.0%	0.0%	100%	99%	0.0%	0.1%	-0.1%	100%	103%
CE11.1.6	0.0%	-1.6%	-2.0%	99%	101%	-0.1%	-1.9%	-2.1%	98%	103%
CE11.1.7 S1	0.4%	0.8%	1.0%	102%	105%	0.2%	1.1%	1.2%	100%	103%
CE11.1.7 S2	0.4%	0.3%	0.2%	102%	104%	0.1%	0.4%	0.2%	101%	103%
CE11.1.8	0.1%	0.0%	0.0%	100%	101%	-0.1%	0.0%	0.1%	100%	101%
CE11.1.9	0.0%	-1.8%	-2.0%	100%	107%	-0.2%	-2.3%	-2.6%	100%	105%
CE11.1.10	0.3%	-1.6%	-1.6%	94%	101%	0.0%	-1.8%	-1.8%	99%	101%
CE11.1.11	0.5%	-0.1%	-0.1%	100%	102%	0.2%	0.0%	-0.1%	99%	102%

3.3 Cross-checks for CE11.1

Test	Cross-checker	PSNR&	RunT	SW	Significant	Studied	Other comments

		BR Match (Y/N)	matched? (Y/N)	studied? (Y/N)	inconsistencies between description and SW? (Y/N)	visual quality? (Y/N)	
CE11.1.1	M. Ikeda	Y	Y	Y	N	N	
	D.Rusanovskyy	Y	Y	Y	N	N	
CE11.1.2	W. I. Choi	Y	Y	Y	N	N	
CE11.1.3	H. Jang	Y	Y	Y	N	N	
CE11.1.4 S1	K. Andersson	Y except 1 point with ALF=0	Y	Y	N	N	Not parallel friendly due to 4x4 grid. Can be solved by asymmetric filtering as in CE11.3.1.
CE11.1.4 S2	K. Andersson	Y for completed so far.. In progress		Y	N	N	
CE11.1.5	K. Kawamura	Y	Y	Y	N	N	
CE11.1.6	K.Andersson	Y	Y	Y	N	N	
CE11.1.7 S1	M. Ikeda	Y	Y	Y	N	N	
CE11.1.7 S2	A. Wieckowski	Y	Y	Y	N	N	
	P. Onno	Y	Y	Y	N	N	
CE11.1.8	W. I. Choi	Y	Y	Y	N	N	
CE11.1.9	C. Helmrich	Y	Y	Y	N	Y	See a comment below
CE11.1.10	A. M. Kotra	Y	Y	Y	N	N	
CE11.1.11	J. Zhao	Y	Y	Y	N	N	

Other comments from cross-checkers

CE11.1.9 (C. Helmrich)**:** BD-WPSNR gains are a bit higher than the corresponding PSNR gains, indicating the visual relevance of CE 11.1.9. Subjectively, cross-checker looked at the RA sequences with the highest BD-(W)PSNR gains, i.e., BasketballDrive, Tango2, and FoodMarket4. The cross-checker sees only very slight visual improvements on the latter two (e. g., the ruby/burgundy colored dress of the Tango2 lady on the far right).
On BasketballDrive (and BQTerrace, which the cross-checker looked at briefly) the cross-checker sees no improvement at all. Regarding Campfire, the cross-checker confirms that the improvement over CE11.1.1 is quite obvious. The cross-checker reports that with CE11.1.1 and 11.1.9, VTM 2.x now looks much better on Campfire than VTM 1.x.

3.4 Description of technologies in CE11.1

CE11.1.1 JVET-L0072

HEVC deblocking filters are extended with longer deblocking filters for luma. One of 7 and 5 sample filtering on respective side of the block boundary can be applied for a 16-sample segment of the block boundary if current and neighboring block size >= 32 and >=16 respectively and corresponding conditions for 8 lines of the boundary are fulfilled as shown below.

Size	Conditions for each of the 8 lines of a 16 sample segment (line 0, 3, 4, 7, 8, 11, 12, 15)

7	$HEVC_{strong\_weak}$&& $\left(\lvert p_0 - 2*p_2 + p_4\rvert + \lvert q_0 - 2*q_2 + q_4\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - p_2 - p_3 + p_5\rvert + \lvert q_0 - q_2 - q_3 + q_5\rvert < \frac{3*\beta}{32}\right)$&& $\left(\lvert p_0 - 2*p_3 + p_6\rvert + \lvert q_0 - 2*q_3 + q_6\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - p_3 - p_4 + p_7\rvert + \lvert q_0 - q_3 - q_4 + q_7\rvert < \frac{3*\beta}{32}\right)$
5	$HEVC_{strong\_weak}$&& $\left(\lvert p_0 - p_4\rvert + \lvert q_0 - q_4\rvert < \frac{\beta}{8}\right)$ && $\left(\lvert p_0 - p_5\rvert + \lvert q_0 - q_5\rvert < \frac{\beta}{8}\right)$
0-3	HEVC filter and decisions

Block boundary samples p_i and q_i for i=0 to S-1 are then replaced by linear interpolation as follows:

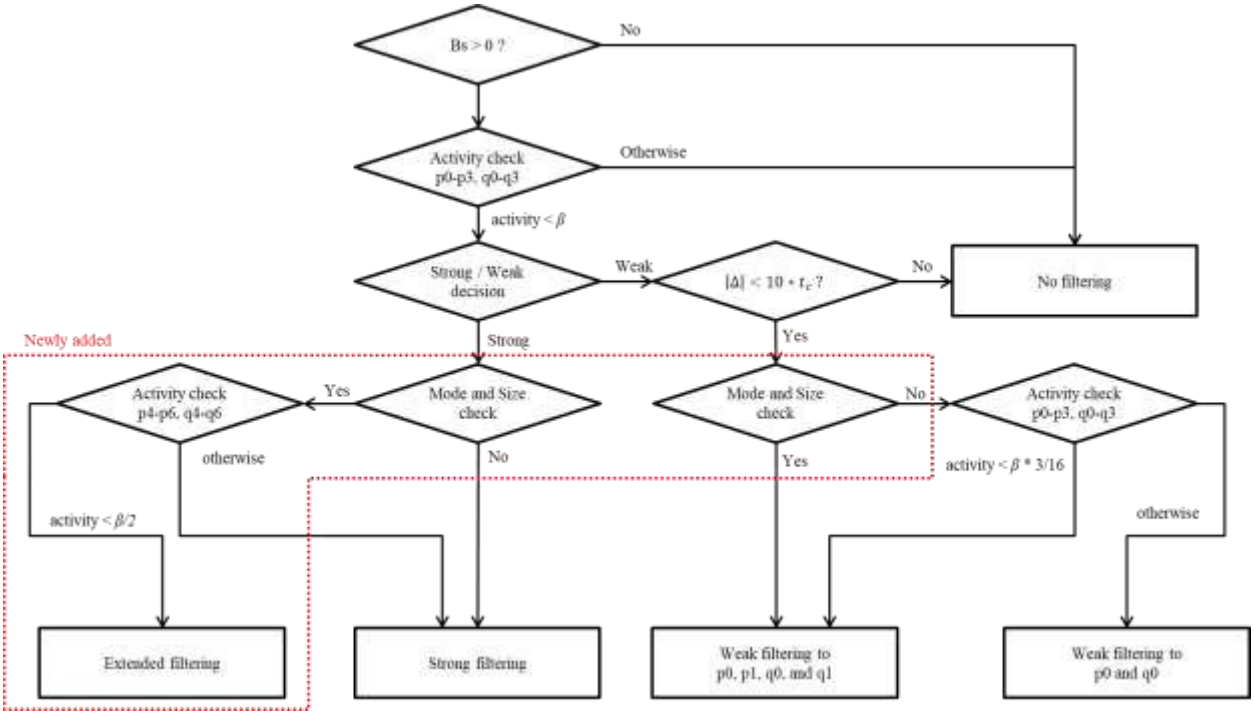
- $p_i' = (f_i * Middle_s + (64 - f_i) * P_s) + 32) \gg 6, clipped\ \ to\ \ p_i \pm \text{tc}$
- $q_i' = (f_i * Middle_s + (64 - f_i) * Q_s + 32) \gg 6), clipped\ \ to\ \ q_i \pm \text{tc}$

where f, Middle, P and Q are given below:

S	Filter kernels
7	$f_i = 59 - i * 9,$ $can\ \ also\ \ be\ \ described\ \ as\ \ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $Middle_7 = (2 * (p_o + q_o) + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1,$ $Q_7 = (q_6 + q_7 + 1) \gg 1$
5	$f_i = 58 - i * 13,$ $can\ \ also\ \ be\ \ described\ \ as\ \ \boldsymbol{f} = \{58,45,32,19,6\}$ $Middle_5 = (2 * (p_o + q_o + p_1 + q_1 + p_2 + q_2) + p_3 + q_3 + p_4 + q_4 + 8) \gg 4$ $P_5 = (p_4 + p_5 + 1) \gg 1,$ $Q_5 = (q_4 + q_5 + 1) \gg 1$

CE11.1.2 JVET-L0380

HEVC deblocking filters are extended with longer deblocking filters for luma. New blocks are shown with red dashed line.



The proposed method consisted of two parts; edge boundary detection of large intra block and extension of the filtered samples on edge filtering process.

When the following conditions are true, the proposed decision process is continued:

- Block boundary is between two blocks of size of >=16 in the direction orthogonal to the boundary.
- The filter type is strong.
- At least one of the blocks forming the boundary is intra coded.

It is noted that a block size is replaced by an edge size. Direction of edge size depends on that of edge boundary like EDGE_VER or EDGE_HOR.

When the above criteria is not true, the default short deblocking filter may be applied.

If the above criteria is true, at block P and block Q, following conditions are checked respectively. If (11.1.2-1) is true, extended filter is applied to block P. In the same manner, if (11.1.2-2) is true, extended filter is applied to block Q.

$$|p6_0 - 2 * p5_0 + p4_0| + |p6_3 - 2 * p5_3 + p4_3| < \frac{\beta}{2} \qquad (11.1.2-1)$$

$$|q6_0 - 2 * q4_0 + q4_0| + |q6_3 - 2 * q5_3 + q4_3| < \frac{\beta}{2} \qquad (11.1.2-2)$$

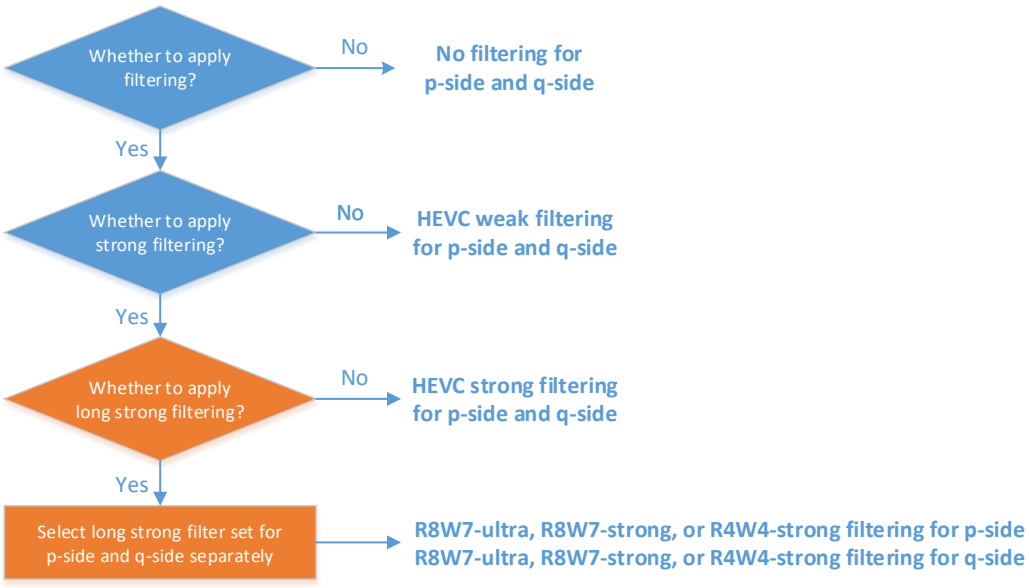
Following filter kernels are used in the extended filtering process.

p6 = 3,5,2,1,1,1,1,1,1,0,0,0,0,0,0,0
p5 = 1,2,5,2,1,1,1,1,1,1,0,0,0,0,0,0
p4 = 0,1,3,3,3,1,1,1,1,1,1,0,0,0,0,0
p3 = 0,1,2,2,2,2,2,1,1,1,1,1,0,0,0,0
p2 = 0,0,1,2,2,2,2,2,2,1,1,1,0,0,0,0
p1 = 0,0,1,1,2,2,2,2,2,2,1,1,0,0,0,0
p0 = 0,0,0,1,2,2,2,2,2,2,2,1,0,0,0,0

There are no changes for chroma deblocking filter process.

CE11.1.3 JVET-L0102

HEVC deblocking filters are extended with longer strong filters for luma. The flowchart of the proposed filtering decisions if shown below.



When all the following conditions are true, the proposed long strong filtering is applied. Otherwise, the HEVC strong filtering is applied for p-side and q-side

- The slice type is inter.
- The CB length at p-side and the CB length at q-side are either larger than or equal to 16.
- Both of the two inter-side difference values representing difference between p-side and q-side are larger than a threshold, *inter_side_diff_th*. The two inter-side difference values are defined as
  - $InterSideDiff_i = \left| \left(p_0^i + p_1^i \right) - \left(q_0^i + q_1^i \right) + 1 \right| >> 1$, where $i = 0$ or 3.
- Both of the two intra-side difference values representing variation within p-side plus variation within q-side are smaller than a threshold, *intra_side_diff_th*. The two intra-side difference values are defined as

  $$IntraSideDiff_i = \sum_{s=4}^{7} \left| p_s^i - p_3^i \right| + \sum_{s=4}^{7} \left| q_s^i - q_3^i \right|$$, where $i = 0$ or 3.
- Any one of adjacent block edges also satisfies all the above four conditions.

When the long strong filtering decision output is yes, the long strong filter set selection is activated to select one of the three long strong filter sets for p-side and one of the three long strong filter sets for q-side. The selection based on the p-side and q-side length of the current block edge is described as follows, where len_p and $leng_q$ denote the CB length at p-side and the CB length at q-side, respectively.

1) If "$len_p \geq 32$ and $len_q \geq 32$", R8W7-ultra or R8W7-strong filtering is selected as follows:
   1.1) If any one of adjacent block edges of the current block edge is also satisfied "$len_p \geq 32$ and $len_q \geq 32$", R8W7-ultra filtering is applied to both p-side and q-side of the current block edge.
   1.2) Otherwise (i.e., "$len_p \geq 32$ and $len_q \geq 32$" is only true for the current block edge), R8W7-strong filtering is applied to both p-side and q-side of the current block edge.
2) If "$len_p > 16$ and $len_q = 16$", R8W7-strong filtering is applied to p-side of the current block edge, and R4W4-strong filtering is applied to q-side of the current block edge.
3) If "$len_p = 16$ and $len_q > 16$", R4W4-strong filtering is applied to p-side of the current block edge, and R8W7-strong filtering is applied to q-side of the current block edge.
4) If "$len_p = 16$ and $len_q = 16$", R4W4-strong filtering is applied to both p-side and q-side of the current block edge.

The filtered values are clipped to stay in the bounds of $-2t_c$ to $2t_c$ of the original values, as in HEVC strong filtering.

If R8W7-ultra filtering is applied to p-side, filtered sample values from p'^i_6 to p'^i_0 are as follows:

$$p'^i_6 = \left(10p^i_7 + 8p^i_6 + 2p^i_5 + 2p^i_4 + 2p^i_3 + 2p^i_2 + 2p^i_1 + 2p^i_0 + q^i_0 + q^i_1 + 16\right) >> 5$$
$$p'^i_5 = \left(8p^i_7 + 4p^i_6 + 6p^i_5 + 2p^i_4 + 2p^i_3 + 2p^i_2 + 2p^i_1 + 2p^i_0 + 2q^i_0 + q^i_1 + q^i_2 + 16\right) >> 5$$
$$p'^i_4 = \left(6p^i_7 + 3p^i_6 + 3p^i_5 + 6p^i_4 + 2p^i_3 + 2p^i_2 + 2p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + q^i_2 + q^i_3 + 16\right) >> 5$$
$$p'^i_3 = \left(4p^i_7 + 3p^i_6 + 3p^i_5 + 3p^i_4 + 5p^i_3 + 2p^i_2 + 2p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + 2q^i_2 + q^i_3 + q^i_4 + 16\right) >> 5$$
$$p'^i_2 = \left(3p^i_7 + 2p^i_6 + 2p^i_5 + 3p^i_4 + 3p^i_3 + 5p^i_2 + 2p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + 2q^i_2 + 2q^i_3 + q^i_4 + q^i_5 + 16\right) >> 5$$
$$p'^i_1 = \left(2p^i_7 + 2p^i_6 + 2p^i_5 + 2p^i_4 + 2p^i_3 + 3p^i_2 + 5p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + 2q^i_2 + 2q^i_3 + 2q^i_4 + q^i_5 + q^i_6 + 16\right) >> 5$$
$$p'^i_0 = \left(p^i_7 + 2p^i_6 + 2p^i_5 + 2p^i_4 + 2p^i_3 + 2p^i_2 + 2p^i_1 + 5p^i_0 + 2q^i_0 + 2q^i_1 + 2q^i_2 + 2q^i_3 + 2q^i_4 + 2q^i_5 + q^i_6 + q^i_7 + 16\right) >> 5$$

If R8W7-ultra filtering is applied to q-side, filtered sample values from q'^i_6 to q'^i_0 are as follows:

$$q'^i_6 = \left(10q^i_7 + 8q^i_6 + 2q^i_5 + 2q^i_4 + 2q^i_3 + 2q^i_2 + 2q^i_1 + 2q^i_0 + p^i_0 + p^i_1 + 16\right) >> 5$$
$$q'^i_5 = \left(8q^i_7 + 4q^i_6 + 6q^i_5 + 2q^i_4 + 2q^i_3 + 2q^i_2 + 2q^i_1 + 2q^i_0 + 2p^i_0 + p^i_1 + p^i_2 + 16\right) >> 5$$
$$q'^i_4 = \left(6q^i_7 + 3q^i_6 + 3q^i_5 + 6q^i_4 + 2q^i_3 + 2q^i_2 + 2q^i_1 + 2q^i_0 + 2p^i_0 + 2p^i_1 + p^i_2 + p^i_3 + 16\right) >> 5$$
$$q'^i_3 = \left(4q^i_7 + 3q^i_6 + 3q^i_5 + 3q^i_4 + 5q^i_3 + 2q^i_2 + 2q^i_1 + 2q^i_0 + 2p^i_0 + 2p^i_1 + 2p^i_2 + p^i_3 + p^i_4 + 16\right) >> 5$$
$$q'^i_2 = \left(3q^i_7 + 2q^i_6 + 2q^i_5 + 3q^i_4 + 3q^i_3 + 5q^i_2 + 2q^i_1 + 2q^i_0 + 2p^i_0 + 2p^i_1 + 2p^i_2 + 2p^i_3 + p^i_4 + p^i_5 + 16\right) >> 5$$
$$q'^i_1 = \left(2q^i_7 + 2q^i_6 + 2q^i_5 + 2q^i_4 + 2q^i_3 + 3q^i_2 + 5q^i_1 + 2q^i_0 + 2p^i_0 + 2p^i_1 + 2p^i_2 + 2p^i_3 + 2p^i_4 + p^i_5 + p^i_6 + 16\right) >> 5$$

$$q'^i_0 = \left(q^i_7 + 2q^i_6 + 2q^i_5 + 2q^i_4 + 2q^i_3 + 2q^i_2 + 2q^i_1 + 5q^i_0 + 2p^i_0 + 2p^i_1 + 2p^i_2 + 2p^i_3 + 2p^i_4 + 2p^i_5 + p^i_6 + p^i_7 + 16\right) >> 5$$

If R8W7-strong filtering is applied to p-side, filtered sample values from p'^i_6 to p'^i_0 are as follows:

$$p'^i_6 = \left(3p^i_7 + 4p^i_6 + 3p^i_5 + 2p^i_4 + p^i_3 + p^i_2 + p^i_1 + p^i_0 + 8\right) >> 4$$
$$p'^i_5 = \left(p^i_7 + 3p^i_6 + 4p^i_5 + 2p^i_4 + p^i_3 + p^i_2 + p^i_1 + p^i_0 + q^i_0 + 8\right) >> 4$$
$$p'^i_4 = \left(p^i_7 + 2p^i_6 + 2p^i_5 + 3p^i_4 + 2p^i_3 + 2p^i_2 + p^i_1 + p^i_0 + q^i_0 + q^i_1 + 8\right) >> 4$$
$$p'^i_3 = \left(p^i_6 + p^i_5 + 2p^i_4 + 3p^i_3 + 3p^i_2 + 2p^i_1 + p^i_0 + q^i_0 + q^i_1 + q^i_2 + 8\right) >> 4$$
$$p'^i_2 = \left(p^i_5 + p^i_4 + 2p^i_3 + 3p^i_2 + 3p^i_1 + 2p^i_0 + q^i_0 + q^i_1 + q^i_2 + q^i_3 + 8\right) >> 4$$
$$p'^i_1 = \left(p^i_5 + p^i_4 + 2p^i_3 + 2p^i_2 + 2p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + q^i_2 + q^i_3 + 8\right) >> 4$$
$$p'^i_0 = \left(p^i_4 + 2p^i_3 + 2p^i_2 + 2p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + q^i_2 + q^i_3 + q^i_4 + 8\right) >> 4$$

If R8W7-strong filtering is applied to q-side, filtered sample values from q'^i_6 to q'^i_0 are as follows:

$$q'^i_6 = \left(3q^i_7 + 4q^i_6 + 3q^i_5 + 2q^i_4 + q^i_3 + q^i_2 + q^i_1 + q^i_0 + 8\right) >> 4$$
$$q'^i_5 = \left(q^i_7 + 3q^i_6 + 4q^i_5 + 2q^i_4 + q^i_3 + q^i_2 + q^i_1 + q^i_0 + p^i_0 + 8\right) >> 4$$
$$q'^i_4 = \left(q^i_7 + 2q^i_6 + 2q^i_5 + 3q^i_4 + 2q^i_3 + 2q^i_2 + q^i_1 + q^i_0 + p^i_0 + p^i_1 + 8\right) >> 4$$
$$q'^i_3 = \left(q^i_6 + q^i_5 + 2q^i_4 + 3q^i_3 + 3q^i_2 + 2q^i_1 + q^i_0 + p^i_0 + p^i_1 + p^i_2 + 8\right) >> 4$$
$$q'^i_2 = \left(q^i_5 + q^i_4 + 2q^i_3 + 3q^i_2 + 3q^i_1 + 2q^i_0 + p^i_0 + p^i_1 + p^i_2 + p^i_3 + 8\right) >> 4$$
$$q'^i_1 = \left(q^i_5 + q^i_4 + 2q^i_3 + 2q^i_2 + 2q^i_1 + 2q^i_0 + 2p^i_0 + 2p^i_1 + p^i_2 + p^i_3 + 8\right) >> 4$$
$$q'^i_0 = \left(q^i_4 + 2q^i_3 + 2q^i_2 + 2q^i_1 + 2q^i_0 + 2p^i_0 + 2p^i_1 + p^i_2 + p^i_3 + p^i_4 + 8\right) >> 4$$

If R4W4-strong filtering is applied to p-side, filtered sample values from p'^i_3 to p'^i_0 are as follows:

$$p'^i_3 = \left(5p^i_3 + 4p^i_2 + 3p^i_1 + p^i_0 + q^i_0 + q^i_1 + q^i_2 + 8\right) >> 4$$
$$p'^i_2 = \left(3p^i_3 + 4p^i_2 + 3p^i_1 + 2p^i_0 + q^i_0 + q^i_1 + q^i_2 + q^i_3 + 8\right) >> 4$$
$$p'^i_1 = \left(2p^i_3 + 3p^i_2 + 3p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + q^i_2 + q^i_3 + 8\right) >> 4$$
$$p'^i_0 = \left(2p^i_3 + 2p^i_2 + 3p^i_1 + 2p^i_0 + 2q^i_0 + 2q^i_1 + 2q^i_2 + q^i_3 + 8\right) >> 4$$

If R4W4-strong filtering is selected at q-side, filtered sample values from q'^i_3 to q'^i_0 are as follows:

$$q'^i_3 = \left(5q^i_3 + 4q^i_2 + 3q^i_1 + q^i_0 + p^i_0 + p^i_1 + p^i_2 + 8\right) >> 4$$
$$q'^i_2 = \left(3q^i_3 + 4q^i_2 + 3q^i_1 + 2q^i_0 + p^i_0 + p^i_1 + p^i_2 + p^i_3 + 8\right) >> 4$$

$$q'^{i}_{1} = \left(2q^{i}_{3} + 3q^{i}_{2} + 3q^{i}_{1} + 2q^{i}_{0} + 2p^{i}_{0} + 2p^{i}_{1} + p^{i}_{2} + p^{i}_{3} + 8\right) >> 4$$
$$q'^{i}_{0} = \left(2q^{i}_{3} + 2q^{i}_{2} + 3q^{i}_{1} + 2q^{i}_{0} + 2p^{i}_{0} + 2p^{i}_{1} + 2p^{i}_{2} + p^{i}_{3} + 8\right) >> 4$$

Within this CE, the performance of the deblocking method described in JVET-K0232, the filter coefficients, and the samples used in long strong filtering are to be studied.

CE11.1.4 JVET-L0403

HEVC deblocking filters are extended with longer deblocking filters for luma. The deblocking process tested in CE11.1.4 is similar to that in JEM with a long-tap filter (instead of strong HEVC filter) being conditionally applied to the boundaries of the blocks with size (orthogonal to the filtered boundary) being larger or equal to 16. A long deblocking filter is applied to large blocks boundary samples if smoothness criteria are met on both sides of the boundary:

1. HEVC strong filter is selected
2. Boundary separates blocks of size larger than 8, with either side being large than 16
3. Long filter smoothness criteria return TRUE

The following pseudo code provides an example of implementation:

```
minBLkSize = min(lengthP, lengthQ);
maxBLkSize = max(lengthP, lengthQ);
if ((maxBLkSize > LARGE_BLOCK_SIZE_DEBLOCK ) && (minBLkSize >= 8))

   if ( isSmoothAsymArea() )

   {
       if (minBLkSize >= LARGE_BLOCK_SIZE_DEBLOCK)
       {
          ApplyLongSymmetricFiltering()
       }
       else if ((MODE_INTRA != cuP.predMode) && (MODE_INTRA != cuQ.predMode))
       {
          ApplyAsymmetricFiltering();
       }
   }
   else
   {
       ApplyHEVCFiltering();
   }
```

Long filter smoothness criterion (`isSmoothAsymArea()`) is computed as an absolute value of second derivatives computed for P and Q blocks independently and this estimate is compared against the thresholds `thresholdP and thresholdQ`. Decision making is conducted for group of 4 lines, with only 2 out of these 4 lines are analyzed.

The thresholds `thresholdP` and `thresholdQ` are derived from the block size parameters, coded QP value and the type of the slice. Derivation of these threshold values can be tabulated.

The following pseudo code provides an example of implementation of the function `isSmoothAsymArea()`:

```
function isSmoothAsymArea()
{
       countQ = min(LARGE_BLOCK_SIZE_CLASSIFIER, (lengthQ >> 1)) - 2);
       countP = min(LARGE_BLOCK_SIZE_CLASSIFIER, (lengthP >> 1)) - 2);
       thresholdP = fun (countP, perCountTh, m_deblockSmoothThQP[iQP]);
       thresholdQ = fun (countP, perCountTh, m_deblockSmoothThQP[iQP]);
       varP = sum(abs(LaplacianOverP))
       varQ = sum(abs(LaplacianOverQ))
       if (varP < thresholdP && varQ < thresholdQ)
       {
         return true;
```

```
        }
        else
          return false;
}
```

When long filter is selected, the following long filters are used to modify pixels q_0, q_1, q_2, q_3 and p_0, p_1, p_2, p_3, with filter $\boldsymbol{h_i}$ is centered at pixel p_i, where $i = 0,1,2,3$:

$$\boldsymbol{h_0} = [0, 2, 9, 12, 1, 3, 13, 16, 5, 3, 0],$$
$$\boldsymbol{h_1} = [-1, 6, 7, 8, -2, 13, 15, 10, 8, 0, 0],$$
$$\boldsymbol{h_2} = [-1, 4, 12, -3, 6, 14, 14, 16, 2, 0, 0],$$
$$\boldsymbol{h_3} = [-3, 16, -3, 1, 5, 16, 18, 14, 0, 0, 0]$$

It should be noted, that filters $\boldsymbol{h_i}$ defined above are shown in 11-tap representation for simplicity of writing, effective filter support area is spanning to 6 boundary samples.

A side of the boundary for which long filter is not applicable either due to length of the filter or not meeting smoothness criteria, is being filtered with HEVC filter.

Clipping is used to limit the amount of filtering. A boundary strength of 3 is used to determine tc for both luma and chroma. Increase tc offset if QP > 42 and decrease if QP<= 35, used for both luma and chroma. Tc is then scaled by 3 for first two samples near the boundary and scaled by 2 for the other samples.

Long filter can be turned on for respective 4 sample segment. Deblocking applied for 4x4 grid for luma.

CE11.1.5 JVET-L0062

HEVC deblocking filters are extended with longer deblocking filters for luma. 7 samples on a side of the boundary is filtered when length orthogonal to block boundary is >= 16, strong/weak filter determines use strong filter, sum of absolute pixel difference for pixel p8 and p0 and q8 and q0 compared to a threshold. Only 3 samples are filtered for the upper block when filtering horizontal boundaries. Decision on 4 sample segments of an 8x8 grid.

Following filters are used for left block in vertical filtering:

p6	{3, 5, 2, 1, 1, 1, 1, 1, 1}	p7 ~ q0
p5	{1, 2, 5, 2, 1, 1, 1, 1, 1, 1}	p7 ~ q1
p4	{1, 3, 3, 3, 1, 1, 1, 1, 1, 1}	p6 ~ q2
p3	{1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1}	p6 ~ q3
p2	{1, 2, 2, 2, 2, 2, 2, 1, 1, 1}	p5 ~ q3
p1	{1, 1, 2, 2, 2, 2, 2, 2, 1, 1}	p5 ~ q3
p0	{1, 2, 2, 2, 2, 2, 2, 2, 1}	p4 ~ q3

For the upper block in horizontal filtering only p0 to p2.

p2	{3, 5, 2, 2, 1, 1, 1, 1}	p3 ~ q3
p1	{2, 2, 4, 2, 2, 2, 1, 1}	p3 ~ q3
p0	{2, 2, 2, 2, 2, 2, 2, 2}	p3 ~ q3

CE11.1.6 JVET-L0327

HEVC deblocking filters are extended with longer deblocking filters for luma and chroma. 7 samples on each side of the boundary is filtered when the minimal length orthogonal to block boundary is >= 32, sum of absolute pixel difference and also Laplacians, and comparison with absolute value of pixel difference across boundary in comparison with tc. Decision is taken on two lines of 4 sample segments on a 8x8 grid.

The decision which has two additional conditions is also modified, accordingly. The first one is for large block decision since the proposed filter can be applied when the block width or height which orthogonally crosses the block edge is equal to or larger than 32. The second one is for proposed wide strong filter. The decision is extended based on HEVC strong filter decision (xUseStrongFiltering) as follows.

dp0 = xCalcDP(R[0]); dp1 = xCalcDP(R[-2]); dp2 = xCalcDP(R[-4]);

dq0 = xCalcDQ(R[0]); dq1 = xCalcDQ(R[2]); dq2 = xCalcDQ(R[4]);

dd0 = 2 * (dp0+dq0); dd2 = 2 * (dp1+dq1); dd4 = 2 * (dp2+dq2);

bdd = (dd0 < (beta>>2) && dd2 < (beta>>2) && dd4 < (beta>>2));

dd_strong0 = abs(R[-8] – R[-1]) + abs(R[0] – R[7]);

dd_strong1 = abs(R[-8] – R[-3]) + abs(R[2] – R[7]);

dd_strong2 = abs(R[-8] – R[-5]) + abs(R[4] – R[7]);

dd_strong_flag =

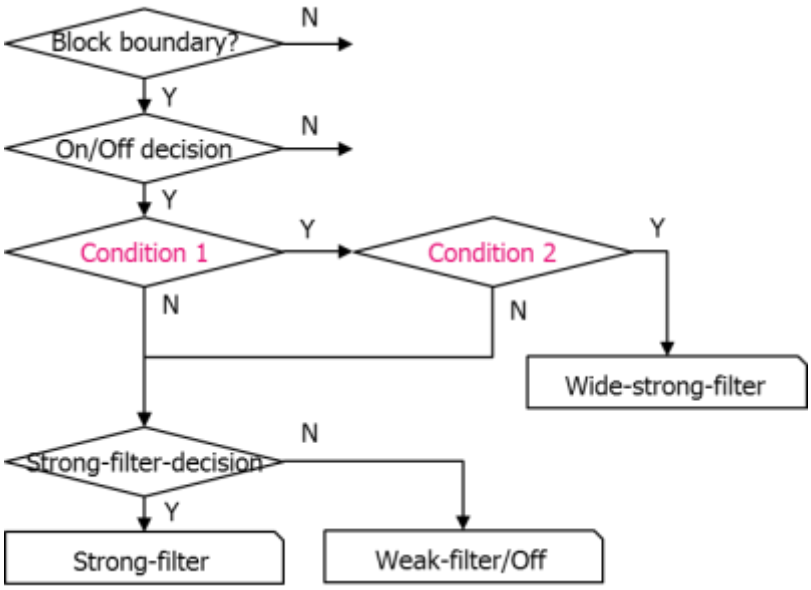
((dd_strong0<(beta>>3)) && (dd_strong0<(beta>>3)) && (dd_strong0<(beta>>3)));

If (dd_strong_flag && bdd && (abs(R[-1]-R[0]) < ((tc*5+1)>>1))) is TRUE, and then the following filter is applied to P side in the same with Q side.

$p_6' = (7*p_7+2*p_6+p_5+p_4+p_3+p_2+ p_1+p_0+q_0+ 8) >> 4$
$p_5' = (6*p_7+p_6+2*p_5+p_4+p_3+p_2+p_1+p_0+q_0+q_1+ 8) >> 4$
$p_4' = (5*p_7+p_6+p_5+2*p_4+p_3+p_2+p_1+p_0+q_0+q_1+q_2+ 8) >> 4$
$p_3' = (4*p_7+p_6+p_5+p_4+2*p_3+p_2+p_1+p_0+q_0+q_1+q_2+q_3+ 8) >> 4$
$p_2' = (3*p_7+p_6+p_5+p_4+p_3+2*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (2*p_7+p_6+p_5+p_4+p_3+p_2+2*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (p_7+p_6+p_5+p_4+p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$

In this conribution, long-tap filter is also proposed for chroma. The decision which has three additional conditions is also modified, accordingly. The first one is for decision of boundary strength as well as large block. The proposed filter can be applied when the block width or height which orthogonally crosses the block edge is equal to or larger than 8 in chroma sample domain. The second and third one are basically the same as for HEVC luma decison, which are on/off decision and strong filter decision, respectively.

In the first decision, boundary strength (bS) is modified for chroma filtering as follows.



Conditions	Y	U	V
At least one of the adjacent blocks is intra	2	2	2
At least one of the adjacent blocks has non-zero transform coefficients	1	1	1
Absolute difference between the motion vectors that belong to the adjacent blocks is greater than or equal to one integer luma sample	1	N/A	N/A
Motion prediction in the adjacent blocks refers to vectors is different	1	N/A	N/A
Otherwise	0	0	0

The following two decisions are the same as HEVC luma strong filter decision:

The second decision:

d < beta

$d = Abs(p_{2,0}-2*p_{1,0}+p_{0,0}) + Abs(q_{2,0}-2*q_{1,0}+q_{0,0}) + Abs(p_{2,1}-2*p_{1,1}+p_{0,1}) + Abs(q_{2,1}-2*q_{1,1}+q_{0,1})$;

The third decision:

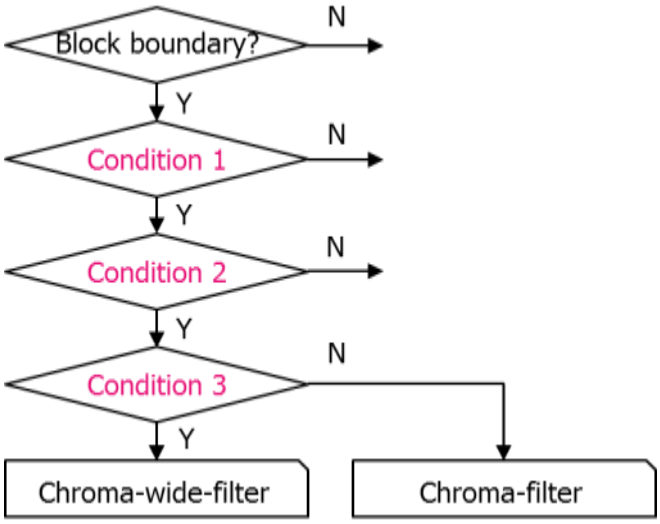
xUseStrongFiltering( ); // same as luma's decision

If those criteria are satisfied, and then the following filter is applied to P side in the same with Q side.

$p_2' = (3*p_3+2*p_2+p_1+p_0+q_0+4) >> 3$

$p_1' = (2*p_3+p_2+2*p_1+p_0+q_0+q_1+4) >> 3$

$p_0' = (p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+4) >> 3$

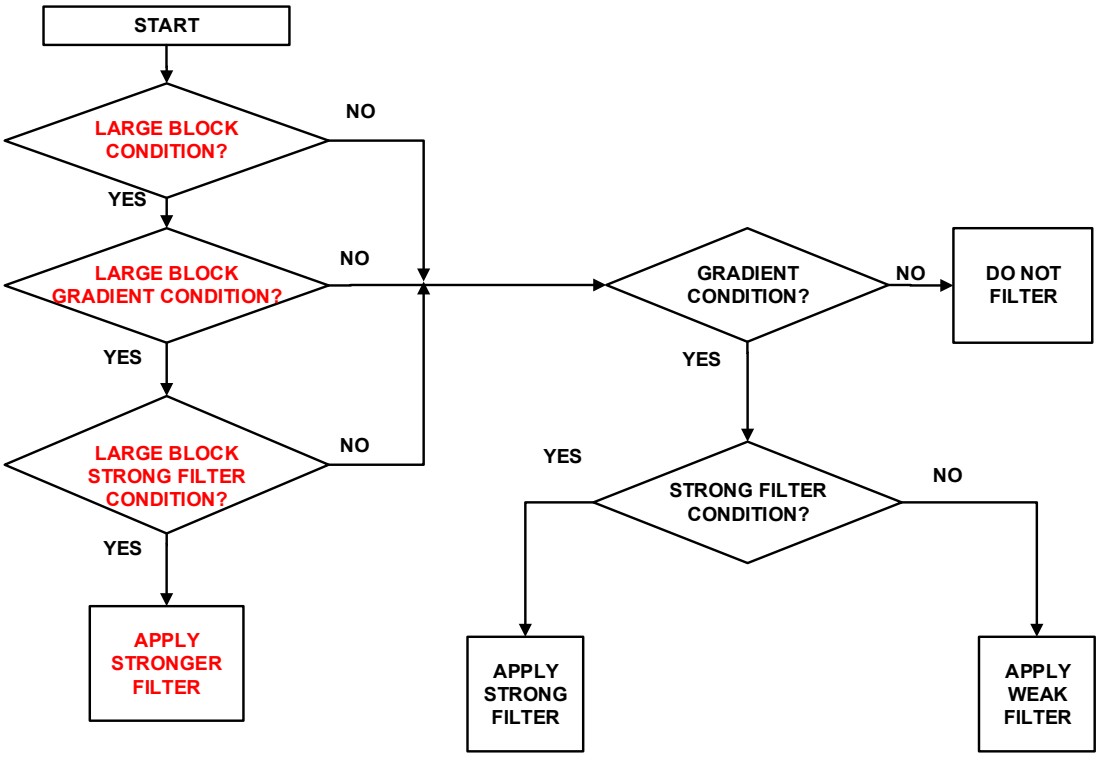


CE11.1.7 JVET-L0405

The proposal incorporates stronger filters for both luma and chroma, compared to JEM deblocking. Additionally, the proposed modification includes a control process that considers the block sizes on both sides of the boundary being deblocked. The stronger filters are used for luma samples that correspond to larger block sizes, while the JEM deblocking filters are still used for luma samples corresponding to the smaller block sizes. For chroma, the filter selection uses a different approach, and a stronger filter is applied when chroma samples on either sides of the deblocking boundary belong to a large block.

Luma filtering.

The flowchart for a stronger filter decision for luma is provided in the picture below:



Luma is deblocked on 8x8 luma sample grid.
Luma stronger filters (same as JVET-J0038):

$p_6' = (7*p_7+2*p_6+p_5+p_4+p_3+p_2+ p_1+p_0+q_0+ 8) >> 4$
$p_5' = (6*p_7+p_6+2*p_5+p_4+p_3+p_2+p_1+p_0+q_0+q_1+ 8) >> 4$
$p_4' = (5*p_7+p_6+p_5+2*p_4+p_3+p_2+p_1+p_0+q_0+q_1+q_2+ 8) >> 4$
$p_3' = (4*p_7+p_6+p_5+p_4+2*p_3+p_2+p_1+p_0+q_0+q_1+q_2+q_3+ 8) >> 4$
$p_2' = (3*p_7+p_6+p_5+p_4+p_3+2*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (2*p_7+p_6+p_5+p_4+p_3+p_2+2*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (p_7+p_6+p_5+p_4+p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$
$q_6' = (7*q_7+2*q_6+q_5+q_4+q_3+q_2+ q_1+q_0+p_0+ 8) >> 4$
$q_5' = (6*q_7+q_6+2*q_5+q_4+q_3+q_2+q_1+q_0+p_0+p_1+ 8) >> 4$
$q_4' = (5*q_7+q_6+q_5+2*q_4+q_3+q_2+q_1+q_0+p_0+p_1+p_2+ 8) >> 4$
$q_3' = (4*q_7+q_6+q_5+q_4+2*q_3+q_2+q_1+q_0+p_0+p_1+p_2+p_3+ 8) >> 4$
$q_2' = (3*q_7+q_6+q_5+q_4+q_3+2*q_2+q_1+q_0+p_0+p_1+p_2+p_3+p_4+8) >> 4$
$q_1' = (2*q_7+q_6+q_5+q_4+q_3+q_2+2*q_1+q_0+p_0+p_1+p_2+p_3+p_4+p_5+8) >> 4$
$q_0' = (q_7+q_6+q_5+q_4+q_3+q_2+q_1+2*q_0+p_0+p_1+p_2+p_3+p_4+p_5+p_6+8) >> 4$

Luma stronger filters are used for large blocks where large block corresponds to width >= 32 for a vertical edge, and height>=32 for a horizontal edge. The control process is further based on gradients computed for two lines of 4 sample segments; comparison of absolute pixel value difference with tc; and comparison of other absolute pixel value differences with β. More gradients are computed for large block side.

1. The variables dpq0, dpq3, dp, dq, and d are derived as follows:

   dp0, dp3, dq0, dq3 are first derived as in HEVC

   if (p side is greater than or equal to 32)

$dp0 = (dp0 + Abs(p_{5,0} - 2 * p_{4,0} + p_{3,0}) + 1) >> 1$

$dp3 = (dp3 + Abs(p_{5,3} - 2 * p_{4,3} + p_{3,3}) + 1) >> 1$

if (q side is greater than or equal to 32)

$dq0 = (dq0 + Abs(q_{5,0} - 2 * q_{4,0} + q_{3,0}) + 1) >> 1$

$dq3 = (dq3 + Abs(q_{5,3} - 2 * q_{4,3} + q_{3,3}) + 1) >> 1$

dpq0, dpq3, dp, dq, d are then derived as in HEVC

2. As in HEVC, When d is less than β, the following ordered steps apply:
    a. dpq is derived as in HEVC.
    b. $sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

       if (p side is greater than or equal to 32)

       $sp_3 = (sp_3 + Abs(p_7 - p_3) + 1) >> 1$
    c. $sq_3 = Abs(q_0 - q_3)$, derived as in HEVC
       if (q side is greater than or equal to 32)

       $sq_3 = (sq_3 + Abs(q_7 - q_3) + 1) >> 1$
    d. As in HEVC derive, StrongFilterCondition = (dpq is less than (β >> 2), $sp_3 + sq_3$ is less than (β >> 3), and $Abs(p_0 - q_0)$ is less than ($5 * t_C + 1$) >> 1) ? TRUE : FALSE
    e. When StrongFilterCondition is TRUE, use luma stronger filter on the side with length perpendicular to boundary edge greater than or equal to 32.

Chroma filtering.

Chroma deblocks on a 4x4 chroma sample grid.

Chroma stronger filters (i.e. HEVC luma strong filter):

$p_0' = (p_2 + 2 * p_1 + 2 * p_0 + 2 * q_0 + q_1 + 4) >> 3$

$p_1' = (p_2 + p_1 + p_0 + q_0 + 2) >> 2$

$p_2' = (2 * p_3 + 3 * p_2 + p_1 + p_0 + q_0 + 4) >> 3$

$q_0' = (p_1 + 2 * p_0 + 2 * q_0 + 2 * q_1 + q_2 + 4) >> 3$

$q_1' = (p_0 + q_0 + q_1 + q_2 + 2) >> 2$

$q_2' = (p_0 + q_0 + q_1 + 3 * q_2 + 2 * q_3 + 4) >> 3$

Two sub-tests for chroma are evaluated.

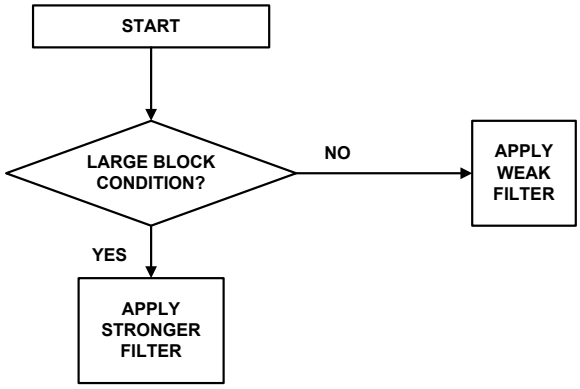
Sub-test 1. Chroma stronger filters are used on both sides of the block boundary. Specifically, the strong filter is used when any of the following conditions is true:

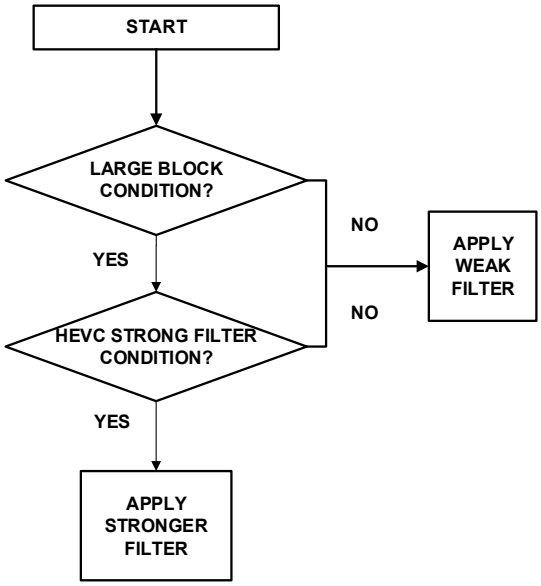
1. The edge type is vertical an p_0 belongs to CU with width >= 16 (chroma samples)
2. The edge type is horizontal and p_0 belongs to CU with height >= 16 (chroma samples)
3. The edge type is vertical and q_0 belongs to CU with width >= 16 (chroma samples)
4. The edge type is horizontal and q_0 belongs to CU with height >= 16 (chroma samples)

Sub-test 2. In another test (b), is same as above, except chroma stronger filters are used on both sides of the block boundary. Specifically, the strong filter is used when HEVC luma strong filter condition when computed for chroma is TRUE and any of the following conditions is true:

1. The edge type is vertical an p_0 belongs to CU with width >= 16 (chroma samples) and q_0 belongs to CU with width >= 16 (chroma samples)

2. The edge type is horizontal and p_0 belongs to CU with height >= 16 (chroma samples) and q_0 belongs to CU with height >= 16 (chroma samples)





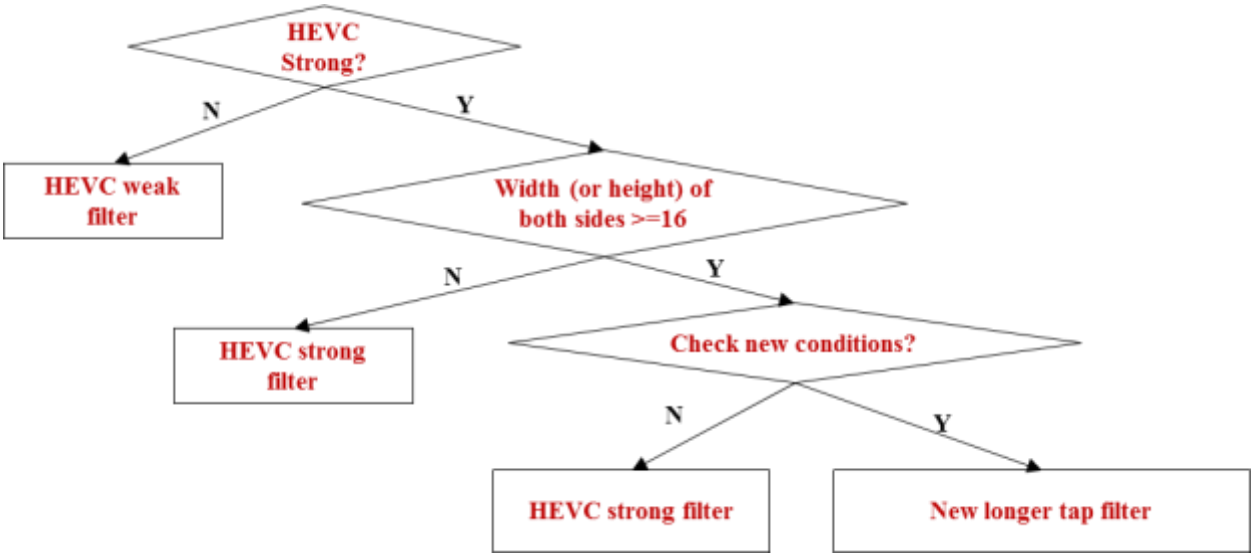
Sub-test 1.

Sub-test 2.

CE11.1.8 JVET-L0224

New luma longer tap filters for large blocks are proposed.

The flowchart for the filtering decisions is as follows:



New luma longer tap filters:

Filter condition: HEVCstrongweak &&

$$(|p_0 - p_3 - p_4 + p_7| + |q_0 - q_3 - q_4 + q_7|) < \frac{3 * \beta}{32} \;\&\&$$
$$(|p_0 - p_2 - p_3 + p_5| + |q_0 - q_2 - q_3 + q_5|) < \frac{3 * \beta}{32}$$

Proposed filter coefficients for longer tap filter:

$p_6' = (6*p_7+3*p_6+p_5+p_4+p_3+p_2+ p_1+p_0+q_0+ 8) >> 4$
$p_5' = (5*p_7+p_6+3*p_5+p_4+p_3+p_2+p_1+p_0+q_0+q_1+ 8) >> 4$
$p_4' = (4*p_7+p_6+p_5+3*p_4+p_3+p_2+p_1+p_0+q_0+q_1+q_2+ 8) >> 4$
$p_3' = (3*p_7+p_6+p_5+p_4+3*p_3+p_2+p_1+p_0+q_0+q_1+q_2+q_3+ 8) >> 4$
$p_2' = (2*p_7+p_6+p_5+p_4+p_3+3*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (p_7+p_6+p_5+p_4+p_3+p_2+3*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (p_7+p_6+p_5+p_4+p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$
$q_6' = (6*q_7+3*q_6+q_5+q_4+q_3+q_2+ q_1+q_0+p_0+ 8) >> 4$
$q_5' = (5*q_7+q_6+3*q_5+q_4+q_3+q_2+q_1+q_0+p_0+p_1+ 8) >> 4$
$q_4' = (4*q_7+q_6+q_5+3*q_4+q_3+q_2+q_1+q_0+p_0+p_1+p_2+ 8) >> 4$
$q_3' = (3*q_7+q_6+q_5+q_4+3*q_3+q_2+q_1+q_0+p_0+p_1+p_2+p_3+ 8) >> 4$
$q_2' = (2*q_7+q_6+q_5+q_4+q_3+3*q_2+q_1+q_0+p_0+p_1+p_2+p_3+p_4+8) >> 4$
$q_1' = (q_7+q_6+q_5+q_4+q_3+q_2+3*q_1+q_0+p_0+p_1+p_2+p_3+p_4+p_5+8) >> 4$
$q_0' = (q_7+q_6+q_5+q_4+q_3+q_2+q_1+2*q_0+p_0+p_1+p_2+p_3+p_4+p_5+p_6+8) >> 4$

Proposed longer tap filter condition for horizontal boundaries overlapping with CTU boundaries:

HEVCstrongweak &&

$$(|p_3 - p_2 - p_1 + p_0| + |q_0 - q_3 - q_4 + q_7|) < \frac{3 * \beta}{32} \;\&\&$$
$$(|p_3 - p_2 - p_1 + p_0| + |q_0 - q_2 - q_3 + q_5|) < \frac{3 * \beta}{32}$$

Proposed longer tap filter coefficients for horizontal boundaries overlapping with CTU boundaries:

$p_2' = (6*p_3+3*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (5*p_3+p_2+3*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (5*p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$
$q_6' = (6*q_7+3*q_6+q_5+q_4+q_3+q_2+ q_1+q_0+p_0+ 8) >> 4$
$q_5' = (5*q_7+q_6+3*q_5+q_4+q_3+q_2+q_1+q_0+p_0+p_1+ 8) >> 4$
$q_4' = (4*q_7+q_6+q_5+3*q_4+q_3+q_2+q_1+q_0+p_0+p_1+p_2+ 8) >> 4$
$q_3' = (3*q_7+q_6+q_5+q_4+3*q_3+q_2+q_1+q_0+p_0+p_1+p_2+p_3+ 8) >> 4$
$q_2' = (2*q_7+q_6+q_5+q_4+q_3+3*q_2+q_1+q_0+p_0+p_1+p_2+2*p_3+8) >> 4$
$q_1' = (q_7+q_6+q_5+q_4+q_3+q_2+3*q_1+q_0+p_0+p_1+p_2+3*p_3+8) >> 4$
$q_0' = (q_7+q_6+q_5+q_4+q_3+q_2+q_1+2*q_0+p_0+p_1+p_2+4*p_3+8) >> 4$

CE11.1.9 JVET-L0072

This test is done on top of long-tap luma filter in test CE11.1.1:

Use HEVC luma filter and decisions with following modifications for chroma deblocking filtering.

Determine the number of samples to modify in block P and Q as follows:

lengthP = edge==EDGE_HOR ? (CUP_height_chroma >> 2) : (CUP_width_chroma >>2)

lengthQ = edge==EDGE_HOR ? (CUQ_height_chroma >> 2) : (CUQ_width_chroma >>2)

Only filter boundary if both lengthQ and lengthP is larger than 0.

The HEVC strong filter (reading 4 samples and modifying 3 samples on respective side) is only applied if both sides have a maximum deblocking filter length larger than 2.

Strong/weak filter and number of samples to filter is determined for each line. Inter blocks are only filtered if Laplacian gradients near the boundary on respective side is below the structure threshold "beta" but for intra blocks that check is by passed.

"beta" parameter is divided by 8 compared to what is used for luma.

CE11.1.10 Combination of CE11.1.6 and CE11.1.7 JVET-L0140

In this test, the luma part comes from CE11.1.7 and the chroma part from CE11.1.6. The stronger filters are used for luma samples that correspond to larger block sizes, while the JEM deblocking filters are still used for luma samples corresponding to the smaller block sizes. Luma deblocks on 8x8 luma sample grid.

Luma Stronger Filter :

$p_6' = (7*p_7+2*p_6+p_5+p_4+p_3+p_2+ p_1+p_0+q_0+ 8) >> 4$
$p_5' = (6*p_7+p_6+2*p_5+p_4+p_3+p_2+p_1+p_0+q_0+q_1+ 8) >> 4$
$p_4' = (5*p_7+p_6+p_5+2*p_4+p_3+p_2+p_1+p_0+q_0+q_1+q_2+ 8) >> 4$
$p_3' = (4*p_7+p_6+p_5+p_4+2*p_3+p_2+p_1+p_0+q_0+q_1+q_2+q_3+ 8) >> 4$
$p_2' = (3*p_7+p_6+p_5+p_4+p_3+2*p_2+p_1+p_0+q_0+q_1+q_2+q_3+q_4+8) >> 4$
$p_1' = (2*p_7+p_6+p_5+p_4+p_3+p_2+2*p_1+p_0+q_0+q_1+q_2+q_3+q_4+q_5+8) >> 4$
$p_0' = (p_7+p_6+p_5+p_4+p_3+p_2+p_1+2*p_0+q_0+q_1+q_2+q_3+q_4+q_5+q_6+8) >> 4$

Luma stronger filters are used for large blocks where large block corresponds to width >= 32 for a vertical edge, and height>=32 for a horizontal edge. The adjacent block must be greater than or equal to 16. The control process is further based on gradients computed for two lines of 4 sample segments; comparison of absolute pixel value difference with tc; and comparison of other absolute pixel value differences with β. More gradients are computed for large block side.

3. The variables dpq0, dpq3, dp, dq, and d are derived as follows:

   dp0, dp3, dq0, dq3 are first derived as in HEVC

   if (p side is greater than or equal to 32)

   $dp0 = (dp0 + Abs(p_{5,0} - 2 * p_{4,0} + p_{3,0}) + 1) >> 1$

   $dp3 = (dp3 + Abs(p_{5,3} - 2 * p_{4,3} + p_{3,3}) + 1) >> 1$

   if (q side is greater than or equal to 32)

   $dq0 = (dq0 + Abs(q_{5,0} - 2 * q_{4,0} + q_{3,0}) + 1) >> 1$

   $dq3 = (dq3 + Abs(q_{5,3} - 2 * q_{4,3} + q_{3,3}) + 1) >> 1$

   dpq0, dpq3, dp, dq, d are then derived as in HEVC

4. As in HEVC, When d is less than β, the following ordered steps apply:

f. dpq is derived as in HEVC.

g. $sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

if (p side is greater than or equal to 32)

$sp_3 = (sp_3 + Abs(p_7 - p_3) + 1) >> 1$

h. $sq_3 = Abs(q_0 - q_3)$, derived as in HEVC
if (q side is greater than or equal to 32)

$sq_3 = (sq_3 + Abs(q_7 - q_3) + 1) >> 1$

i. As in HEVC derive, StrongFilterCondition = (dpq is less than ($\beta >> 2$), $sp_3 + sq_3$ is less than ($\beta >> 3$), and $Abs(p_0 - q_0)$ is less than ($5 * t_C + 1) >> 1$) ? TRUE : FALSE

j. When StrongFilterCondition is TRUE, use luma stronger filter on the side with length perpendicular to boundary edge greater than or equal to 32.

Chroma deblocks on a 4x4 chroma sample grid

Use of HEVC luma strong filter is also proposed for chroma when both sides of the chroma edge are greater than or equal to 8 and the following decision with three conditions are satisfied. The first one is for decision of boundary strength as well as large block. The proposed filter can be applied when the block width or height which orthogonally crosses the block edge is equal to or larger than 8 in chroma sample domain. The second and third one are basically the same as for HEVC luma decison, which are on/off decision and strong filter decision, respectively.

In the first decision, boundary strength (bS) is modified for chroma filtering as follows.

Conditions	Y	U	V
At least one of the adjacent blocks is intra	2	2	2
At least one of the adjacent blocks has non-zero transform coefficients	1	1	1
Absolute difference between the motion vectors that belong to the adjacent blocks is greater than or equal to one integer luma sample	1	N/A	N/A
Motion prediction in the adjacent blocks refers to vectors is different	1	N/A	N/A
Otherwise	0	0	0

The second and third decision is basically the same as HEVC luma strong filter decision as follows.

The second decision:

d < beta

$d = Abs(p_{2,0}-2*p_{1,0}+ p_{0,0}) + Abs(q_{2,0}-2*q_{1,0}+q_{0,0}) + Abs(p_{2,1}-2*p_{1,1}+p_{0,1}) + Abs(q_{2,1}-2*q_{1,1}+q_{0,1})$;

The third decision:

xUseStrongFiltering( ); // same as luma's decision

If those decisions are satisfied, and then the following filter is applied to P side in the same with Q side.

$p_0' = (p_2 + 2 * p_1 + 2 * p_0 + 2 * q_0 + q_1 + 4) >> 3$
$p_1' = (p_2 + p_1 + p_0 + q_0 + 2) >> 2$
$p_2' = (2 * p_3 + 3 * p_2 + p_1 + p_0 + q_0 + 4) >> 3$

CE11.1.11 Combination of CE11.1.1 and CE11.1.7 JVET-L0337

The filtering method in CE11.1.1 (JVET-K0307) is modified to enable use of long asymmetric filters. For long asymmetric filters the number of samples deblocked on the larger block side is greater than that on the smaller block side. The deblocking decision processes described in CE11.1.1 (JVET-K0307) and CE11.1.7 (JVET-K0315) are combined to select from this extended filter set. The extended filter set may be used for strong deblocking. The luma filtering process is mostly adopted from CE11.1.1 whereas the filtering decisions from CE11.1.7 are used.

Luma stronger filters are used when either side has a large block and modified strong filter condition is met. Luma large block corresponds to width >= 32 for a vertical edge, and height>=32 for a horizontal edge.

Luma Stronger Filter :

Block boundary samples p_i and q_i for i=0 to S-1 are then replaced by linear interpolation as follows:

- $p_i' = (f_i * Middle_{s,t} + (64 - f_i) * P_s + 32) \gg 6), clipped\ \ to\ \ p_i \pm$ tc
- $q_i' = (g_i * Middle_{s,t} + (64 - g_i) * Q_s + 32) \gg 6), clipped\ \ to\ \ q_i \pm$ tc

where f_i, Middle_s,t, P_s and Q_s are given below:

7, 7 (p side: 7, q side: 7)	$f_i = 59 - i * 9,\ \ can\ also\ be\ described\ as\ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $g_i = 59 - i * 9,\ \ can\ also\ be\ described\ as\ \boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $Middle_{7,7} = (2 * (p_o + q_o) + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1, \quad Q_7 = (q_6 + q_7 + 1) \gg 1$
7, 3 (p side: 7 q side: 3)	$f_i = 59 - i * 9,\ \ can\ also\ be\ described\ as\ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $g_i = 53 - i * 21,\ \ can\ also\ be\ described\ as\ \boldsymbol{g} = \{53,32,11\}$ $Middle_{7,3} = (2 * (p_o + q_o) + q_0 + 2 * (q_1 + q_2) + p_1 + q_1 + p_2 + p_3 + p_4 + p_5 + p_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1, \quad Q_3 = (q_2 + q_3 + 1) \gg 1$
3, 7 (p side: 3 q side: 7)	$g_i = 59 - i * 9,\ \ can\ also\ be\ described\ as\ \boldsymbol{g} = \{59,50,41,32,23,14,5\}$ $f_i = 53 - i * 21,\ \ can\ also\ be\ described\ as\ \boldsymbol{f} = \{53,32,11\}$ $Middle_{3.7} = (2 * (q_o + p_o) + p_0 + 2 * (p_1 + p_2) + q_1 + p_1 + q_2 + q_3 + q_4 + q_5 + q_6 + 8) \gg 4$ $Q_7 = (q_6 + q_7 + 1) \gg 1, \quad P_3 = (p_2 + p_3 + 1) \gg 1$

The control process is further based on gradients computed for two lines of 4 sample segments; comparison of absolute pixel value difference with tc; and comparison of other absolute pixel value differences with β. More gradients are computed for large block side.

1. The variables dpq0, dpq3, dp, dq, and d are derived as follows:

dp0, dp3, dq0, dq3 are first derived as in HEVC

if (p side is greater than or equal to 32)

$dp0 = (dp0 + Abs(p_{5,0} - 2 * p_{4,0} + p_{3,0}) + 1) >> 1$

$dp3 = (dp3 + Abs(p_{5,3} - 2 * p_{4,3} + p_{3,3}) + 1) >> 1$

if (q side is greater than or equal to 32)

$dq0 = (dq0 + Abs(q_{5,0} - 2 * q_{4,0} + q_{3,0}) + 1) >> 1$

$dq3 = (dq3 + Abs(q_{5,3} - 2 * q_{4,3} + q_{3,3}) + 1) >> 1$

dpq0, dpq3, dp, dq, d are then derived as in HEVC

2. As in HEVC, When d is less than β, the following ordered steps apply:
   a. dpq is derived as in HEVC.
   b. $sp_3 = Abs(p_3 - p_0)$, derived as in HEVC

      if (p side is greater than or equal to 32)

      $sp_3 = (sp_3 + Abs(p_7 - p_3) + 1) >> 1$
   c. $sq_3 = Abs(q_0 - q_3)$, derived as in HEVC
      if (q side is greater than or equal to 32)

      $sq_3 = (sq_3 + Abs(q_7 - q_3) + 1) >> 1$
   d. As in HEVC derive, StrongFilterCondition = (dpq is less than (β >> 2), $sp_3 + sq_3$ is less than (β >> 3), and $Abs(p_0 - q_0)$ is less than (5 * t_C + 1) >> 1) ? TRUE : FALSE
   e. When StrongFilterCondition is TRUE, use luma stronger filter.

Use of HEVC luma strong filter is also proposed for chroma (as chroma stronger filter) when both sides of the chroma edge are greater than or equal to 16 (other threshold, such as 8 may be tested) and HEVC luma strong filter condition when computed for chroma is TRUE.

4 CE11.2: General aspects of deblocking filter

The proposals in this sub-CE have not been included into either sub-CE 1 or sub-CE3. In particular proposal CE11.2.1, experiments with derivation of tC table values for different bit depths as well as chroma deblocking. Proposal CE11.2.2 investigates adaptively choosing QP offset for deblocking depending on the average luma values adjacent to the edge.

Test	**Proponent(s)**	**Cross-checker(s)**
CE11.2.1	Christophe Gisquet christophe.gisquet@crf.canon.fr JVET-L0192	Anand Meher Kotra anand.meher.kotra@huawei.com Biao Wang: biao.wang@huawei.com
CE11.2.2	Atsuro Ichigaya ichigaya.a-go@nhk.or.jp JVET-L0414	Anand Meher Kotra anand.meher.kotra@huawei.com Biao Wang: biao.wang@huawei.com

4.1 Complexity of technologies in CE11.2

4.1.1 Luma deblocking complexity (CE11.2)

Tests	Samples from block bound. modified	Samples from block bound. fdeblocking decision	Max num. oper for filtering per line (add/mult/compar/shift)	Max number of oper. for decision for 8-sample boundary (add/mult/compar/shift)	Num. line buffers
CE11.2.1 S1	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S2	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S3	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S4	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S5	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S6	VTM	VTM	VTM	VTM	VTM
CE11.2.2 S1	VTM	VTM	VTM	VTM	VTM
CE11.2.2 S2	VTM	VTM	VTM	VTM	VTM

4.1.2 Chroma deblocking complexity (CE11.2)

Tests	Samples from block bound. modified	Samples from block bound. fdeblocking decision	Max num. oper for filtering per line (add/mult/compar/shift)	Max number of oper. for decision for 8-sample boundary (add/mult/compar/shift)	Num. line buffers
CE11.2.1 S1	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S2	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S3	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S4	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S5	VTM	VTM	VTM	VTM	VTM
CE11.2.1 S6	2+2	VTM	VTM+1 shift+2 add	VTM	VTM
CE11.2.2 S1	VTM	VTM	VTM	VTM	VTM
CE11.2.2 S2	VTM	VTM	VTM	VTM	VTM

4.1.3 Parallel processing (CE11.2)

Tests	Min unit size in luma samples that can be processed separately
CE11.2.1 S1	VTM
CE11.2.1 S2	VTM
CE11.2.1 S3	VTM
CE11.2.1 S4	VTM
CE11.2.1 S5	VTM
CE11.2.1 S6	VTM
CE11.2.2 S1	VTM
CE11.2.2 S2	VTM

4.2 Objective results of technologies in CE11.2

4.2.1 VTM 2.0.1 CTC (ALF on)

	AI					RA				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.2.1 S1	0.0%	-0.7%	-0.5%			0.0%	-1.7%	-1.8%		
CE11.2.1 S2	-0.4%	-0.6%	-0.4%			-0.1%	-0.6%	-0.6%		
CE11.2.1 S3	-0.3%	-0.6%	-0.4%			-0.1%	-0.4%	-0.3%		
CE11.2.1 S4	-0.3%	-0.6%	-0.4%			0.1%	-0.7%	-0.5%		
CE11.2.1 S5	-0.4%	-0.8%	-0.5%			-0.2%	-1.8%	-1.9%		
CE11.2.1 S6	-0.4%	-1.0%	-0.8%			-0.2%	-2.3%	-2.4%		
CE11.2.2 S1	0.0%	0.0%	0.0%		98%	0.0%	0.1%	-0.1%		99%
CE11.2.2 S2	0.1%	0.0%	0.0%		98%	0.1%	0.0%	-0.1%		98%

	LD-B					LD-P				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.2.1 S1	-0.1%	-1.9%	-2.1%			-0.1%	-1.9%	-1.8%		
CE11.2.1 S2	-0.2%	-0.3%	-0.3%			-0.2%	-0.2%	-0.2%		
CE11.2.1 S3	-0.2%	-0.4%	-0.2%			-0.2%	-0.3%	-0.3%		
CE11.2.1 S4	-0.2%	-0.4%	-0.2%			0.0%	-0.2%	-0.1%		
CE11.2.1 S5	-0.3%	-2.0%	-1.9%			-0.3%	-2.0%	-1.9%		
CE11.2.1 S6	-0.3%	-1.8%	-1.9%			-0.3%	-1.9%	-1.9%		
CE11.2.2 S1	0.0%	-0.1%	-0.1%		98%	0.0%	0.0%	0.3%		98%
CE11.2.2 S2	0.1%	0.2%	-0.1%		97%	0.1%	0.0%	0.2%		99%

4.2.2 VTM 2.0.1 (ALF off)

	AI					RA				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.2.1 S1	0.0%	-0.1%	0.2%			0.0%	-2.1%	-2.2%		
CE11.2.1 S2	-0.3%	-0.5%	-0.3%			0.0%	-0.8%	-0.6%		
CE11.2.1 S3	-0.2%	-0.5%	-0.3%			0.1%	-0.4%	-0.4%		
CE11.2.1 S4	-0.3%	-0.3%	0.0%			0.3%	-0.7%	-0.6%		
CE11.2.1 S5	-0.3%	-0.1%	0.3%			0.0%	-2.3%	-2.2%		
CE11.2.1 S6	-0.3%	-0.5%	-0.1%			0.0%	-2.6%	-2.5%		
CE11.2.2 S1	0.0%	0.0%	0.0%	100%	99%	0.0%	-0.1%	0.0%	100%	99%
CE11.2.2 S2	0.1%	0.0%	0.0%	100%	99%	0.0%	0.0%	0.0%	100%	100%

	LD-B					LD-P				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.2.1 S1	0.0%	-1.6%	-1.6%			-0.1%	-2.0%	-2.2%		
CE11.2.1 S2	-0.1%	-0.3%	-0.3%			0.1%	0.1%	-0.1%		
CE11.2.1 S3	-0.1%	0.0%	0.0%			0.2%	-0.2%	0.2%		
CE11.2.1 S4	0.0%	-0.2%	0.0%			0.6%	-0.1%	0.3%		
CE11.2.1 S5	-0.2%	-1.6%	-1.6%			0.1%	-2.0%	-1.8%		

CE11.2.1 S6	-0.2%	-1.3%	-1.5%			0.0%	-2.2%	-2.1%		
CE11.2.2 S1	0.0%	-0.1%	-0.1%	99%	101%	0.0%	0.0%	0.0%	99%	101%
CE11.2.2 S2	0.1%	0.0%	0.0%	100%	102%	0.0%	0.2%	0.1%	100%	101%

4.3 Cross-checks of tests in CE11.2

Test	Cross-checker	PSNR& BR Match (Y/N)	RunT matched ? (Y/N)	SW studied ? (Y/N)	Significant inconsistencies between description and SW? (Y/N)	Studied visual quality? (Y/N)	Other comments
CE11.2.1 S1	A.M. Kotra B.Wang	Y	N	Y	N	N	
CE11.2.1 S2		Y	N	Y	N	N	
CE11.2.1 S3		N	N	Y	N	N	Due to a VTM encoder bug (see issue #93 on VVC bug tracker)
CE11.2.1 S4		N	N	Y	N	N	Due to a VTM encoder bug (see issue #93 on VVC bug tracker)
CE11.2.1 S5		Y	N	Y	N	N	
CE11.2.1 S6		Y	N	Y	N	N	
CE11.2.2 S1		Y	Y	Y	N	N	
CE11.2.2 S2		Y	Y	Y	N	N	

Other comments from cross-checkers

4.4 Description of technologies in CE11.2

CE11.2.1 Higher-precision modifications to VVC deblocking filters JVET-L0192

S1 (formerly TEST 1): Chroma deblocking decision from luma. Chroma filtering ON/OFF decision switches on the same pixel granularity as luma one does, following CTCs (10 bits). This is related to section 1.2.3.1 of contribution JVET-K0203.

TEST 2: tc parameter link to bit depth

This corresponds to section 1.1 of contribution JVET-K0203. This test requires 3 types of encoding. The goal is to show that the rule from HEVC equation 8-353 is sub-optimal. As a consequence, encoding at 2 bit depths is required, and showcasing "traditional (as in HEVC)" vs "scaling" is required. The following encodings are thus required:

1) S2 (10 bits): Per 10 bits CTC, using an "optimal" scaling from a high-precision t_c curve
2) Encoding with an internal bit depth of 8 bits:
    a. S4 (8 bits): Using the 10-bits scaling, and dividing it by 4 to mimic HEVC rule in equation 8-353
    b. S3 (8 bits): Proposing an "optimal" 8-bit scaling instead

S5 (formerly TEST 3): combination of TEST1 and TEST2, following CTCs (10 bits)

S6 (formerly TEST 4): TEST3 with longer chroma filter, following CTCs (10 bits)

The addition in S6 (compared to S5) corresponds to what is described in section 1.3.2.2 of contribution JVET-K0203. Current filtering process for chroma samples (section 8.7.2.5.8 of H.265 v5) is as follows:

$$\Delta = Clip3(-tC, tC, ((((q0 - p0) << 2) + p1 - q1 + 4) >> 3)) \quad (8\text{-}405)$$

$$p0' = Clip1C(p0 + \Delta) \quad (8\text{-}406)$$

$$q0' = Clip1C(q0 - \Delta) \quad (8\text{-}407)$$

The added filtering is:

$$\Delta 1 = \Delta >> 1$$

$$p1' = Clip1C(p1 + \Delta 1)$$

$$q1' = Clip1C(q1 - \Delta 1)$$

CE11.2.2 Luma-adaptive deblocking filter JVET-L0414

The variable LL of the reconstructed luma level is derived as follow:

$$LL= ((p_{0,0} + p_{0,3} + q_{0,0} + q_{0,3}) >> 2) / (1 << iBitDepth), \quad (1)$$

where, the sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0 and 3 are derived the same as HEVC as shown in Figure 1.

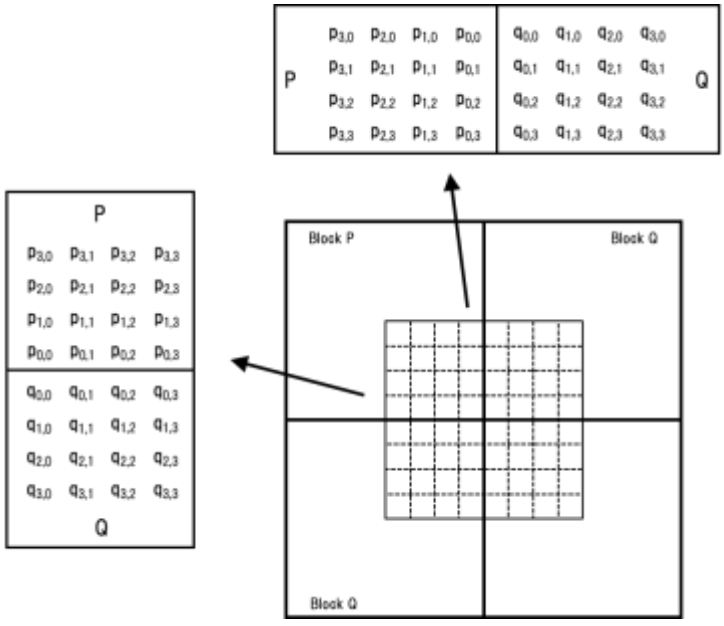


Figure 1: Sample position of $p_{i,k}$ and $q_{i,k}$

The variables Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

The variable qP_L is derived as follows:

$$qP_L = ((QP_Q + QP_P + 1) >> 1) + qpOffset, \quad (2)$$

where qpOffset is specified in Table 1.

The subsequent derivation processes of the variables β and t_C are same as HEVC.

Table 1. Specification of qpOffset as a function of LL (Test 1)

transfer_characteristics == 16 (PQ format)		transfer_characteristics == 18 (HLG format)		Otherwise (conventional SDR format)	
Conditions	qpOffset	Conditions	qpOffset	Conditions	qpOffset
LL <= 0.84	0	LL <= 0.8	0	LL <= 0.875	0
0.84 < LL <= 0.91	1	0.8 < LL <= 0.9	1	0.875 < LL	1
0.91 < LL <= 0.99	2	0.9 < LL	2		
0.99 < LL	3				

Table 2. Specification of qpOffset as a function of LL (Test2)

transfer_characteristics == 1 (conventional SDR format)		transfer_characteristics == 16 (PQ format)		transfer_characteristics == 18 (HLG format)	
Conditions	qpOffset	Conditions	qpOffset	Conditions	qpOffset
$LL \leq 0.326$	1	$LL \leq 0.608$	2	$LL \leq 0.504$	2
$0.326 < LL \leq 0.878$	0	$0.608 < LL \leq 0.685$	1	$0.504 < LL \leq 0.612$	1
$0.878 < LL$	1	$0.685 < LL \leq 0.837$	0	$0.612 < LL \leq 0.822$	0
		$0.837 < LL \leq 0.910$	1	$0.822 < LL \leq 0.926$	1
		$0.910 < LL$	2	$0.926 < LL$	2

5 CE 11.3: Deblocking at 4x4 block boundaries

The proposals in this CE study application of deblocking filtering to the 4x4 boundaries. Common feature of proposals studied in this CE is limiting number of modified samples and samples accessed for filtering decisions when the block size is 4 samples in the dimension orthogonal to the block boundary. This is done to avoid introducing a dependency of deblocking for an edge on a parallel edge preceding in a filtering order. Test CE11.3.5 investigates whether filtering any edges adjacent to the blocks of size 4 can be avoided.

Test	**Proponent(s)**	**Cross-checker(s)**
CE11.3.1	Kenneth Andersson kenneth.r.andersson@ericsson.com JVET-L0073	Dmytro Rusanovskyy dmytror@qti.qualcomm.com
CE11.3.2	Kenneth Andersson kenneth.r.andersson@ericsson.com	Chia-Ming Tsai

	JVET-L0074	chia-ming.tsai@mediatek.com
CE11.3.3	Anand Meher Kotra Anand.meher.kotra@huawei.com JVET-L0225	Kiran Misra misrak@sharplabs.com
CE11.3.4	Chia-Ming Tsai chia-ming.tsai@mediatek.com JVET-L0103	Kenneth Andersson kenneth.r.andersson@ericsson.com
CE11.3.5	Hyeongmun Jang hm.jang@lge.com JVET-L0170	Chia-Ming Tsai chia-ming.tsai@mediatek.com

5.1 Complexity of technologies in CE11.3

5.1.1 Luma deblocking complexity (*CE11.3*)

Tests	**Samples from block bound. modified**	**Samples from block bound. for deblocking decision**	**Max num. oper for filtering per line (add/mult/compar/shift)**	**Max number of oper. for decision for 8-sample boundary (add/mult/compar/shift)**	**Num. line buffers**
CE11.3.1	VTM	VTM	VTM	VTM	VTM
CE11.3.2	VTM	VTM	VTM	VTM	VTM
CE11.3.3 S1	VTM	VTM	VTM	VTM	VTM
CE11.3.3 S2	VTM	VTM	VTM	VTM	VTM
CE11.3.4	VTM	VTM	VTM	VTM	VTM
CE11.3.5	VTM	VTM	VTM	VTM	VTM

5.1.2 Chroma deblocking complexity (*CE11.3*)

Tests	**Samples from block bound. modified**	**Samples from block bound. for deblocking decision**	**Max num. oper for filtering per line (add/mult/compar/shift)**	**Max number of oper. for decision for 8-sample boundary (add/mult/compar/shift)**	**Num. line buffers**
CE11.3.1	VTM	VTM	VTM	VTM	VTM
CE11.3.2	VTM	VTM	VTM	VTM	VTM
CE11.3.3 S1	VTM	VTM	VTM	VTM	VTM
CE11.3.3 S2	VTM	VTM	VTM	VTM	VTM
CE11.3.4	VTM	VTM	VTM	VTM	VTM
CE11.3.5	VTM	VTM	VTM	VTM	VTM

5.1.3 Parallel processing (*CE11.3*)

Tests	**Min unit size in luma samples that can be processed separately**

CE11.3.1	4x4 grid
CE11.3.2	VTM
CE11.3.3 S1	4x4 grid
CE11.3.3 S2	4x4 grid
CE11.3.4	4x4 grid
CE11.3.5	4x4 grid

5.2 Objective results of technologies in CE11.3

5.2.1 VTM2.0.1 CTC (ALF on)

	AI					RA				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.3.1	0.1%	0.0%	0.0%	103%*	97%	-0.1%	0.0%	-0.1%	98%*	100%
CE11.3.2	0.0%	0.0%	0.0%	101%	100%	0.0%	0.0%	0.0%	98%*	100%
CE11.3.3 S1	0.1%	0.0%	0.0%	100%	103%	-0.1%	0.0%	-0.1%	100%	100%
CE11.3.3 S2	0.0%	0.0%	0.0%	100%	103%	-0.2%	0.0%	-0.1%	99%	100%
CE11.3.4	0.0%	0.0%	0.0%	100%	100%	-0.2%	0.0%	0.0%	100%	100%
CE11.3.5	-0.1%	0.3%	0.2%	99%	98%	0.0%	0.1%	0.0%	100%	97%

	LD-B					LD-P				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.3.1	-0.2%	-0.2%	-0.2%	99%	99%	-0.2%	0.0%	0.1%	100%	100%
CE11.3.2	-0.1%	-0.1%	0.1%	98%*	100%	0.0%	0.0%	0.1%	98%*	100%
CE11.3.3 S1	-0.2%	-0.1%	-0.2%	100%	100%	-0.2%	0.1%	0.2%	100%	101%
CE11.3.3 S2	-0.2%	-0.1%	-0.2%	100%	100%	-0.2%	0.0%	0.1%	100%	101%
CE11.3.4	-0.3%	0.0%	-0.2%	101%	100%	-0.3%	0.0%	0.0%	100%	101%
CE11.3.5	0.0%	-0.2%	-0.3%	99%	95%	0.0%	0.0%	0.1%	100%	98%

5.2.2 VTM 2.0.1 CTC (ALF off)

	AI					RA				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.3.1	0.0%	0.0%	0.0%	101%	98%	-0.3%	-0.1%	-0.1%	99%	99%
CE11.3.2	0.0%	0.0%	0.0%	99%	100%	-0.1%	0.0%	0.0%	99%	99%
CE11.3.3 S1	0.0%	0.0%	0.0%	100%	103%	-0.3%	-0.1%	-0.1%	99%	101%
CE11.3.3 S2	0.0%	0.0%	0.0%	99%	103%	-0.3%	0.0%	-0.1%	99%	100%
CE11.3.4	0.0%	0.0%	0.0%	100%	101%	-0.3%	-0.1%	0.0%	100%	101%
CE11.3.5	0.0%	-0.2%	-0.4%	100%	99%	0.0%	0.2%	0.1%	101%	100%

	LD-B					LD-P				
Test	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**	**Y**	**U**	**V**	**EncT (%)**	**DecT (%)**
CE11.3.1	-0.2%	-0.1%	-0.1%	100%	100%	-0.4%	-0.1%	-0.2%	101%	100%

CE11.3.2	-0.1%	0.0%	-0.1%	100%	100%	-0.1%	0.0%	0.0%	99%	100%
CE11.3.3 S1	-0.2%	-0.1%	-0.1%	100%	101%	-0.4%	-0.1%	-0.1%	100%	101%
CE11.3.3 S2	-0.3%	-0.1%	-0.1%	100%	101%	-0.4%	0.1%	-0.2%	99%	101%
CE11.3.4	-0.2%	0.0%	-0.1%	100%	100%	-0.4%	0.0%	0.1%	100%	100%
CE11.3.5	0.0%	-0.2%	-0.3%	101%	101%	0.0%	-0.2%	-0.4%	100%	99%

5.3 Cross-checks of tests in CE11.3

Test	Cross-checker	PSNR &BR Match (Y/N)	RunT matched? (Y/N)	SW studied? (Y/N)	Significant inconsistencies between description and SW? (Y/N)	Studied visual quality? (Y/N)	Other comments
CE11.3.1	D. Rusanovskyy	Y	Y	Y	N	N	
CE11.3.2	C.-M. Tsai	Y	Y	Y	N	N	
CE11.3.3 S1	K. Misra	Y	Y	Y	N	N	
CE11.3.3 S2	K. Misra	Y	Y	Y	N	N	
CE11.3.4	K. Andersson	Y	Y	Y	N	N	
CE11.3.5	C.-M. Tsai	Y	Y	Y	N	N	

Other comments from cross-checkers

5.4 Description of technologies in CE11.3

CE11.3.1 CU deblocking on 4x4 grid for luma JVET-L0073

Determine the number of samples to modify in block P and Q as follows:

lengthP = edge==EDGE_HOR ? (CUP_height >> 1) – 1 : (CUP_width>>1) - 1

lengthQ = edge==EDGE_HOR ? (CUQ_height >> 1) – 1 : (CUQ_width>>1) - 1

This means that only one sample will be filtered with the weak VVC filter on side P when the distance between boundaries on P side is 4 and only one sample will be filtered with the weak VVC filter on side Q when the distance between the boundaries on Q side is 4.

The weak VVC filter can be used with two sample filtering on at least one side if lengthP or lengthQ is larger than 1. The strong VVC filter can be used if both lengthP and lengthQ are larger than 2.

CE11.3.2 Filtering of sub-block boundaries for luma JVET-L0074

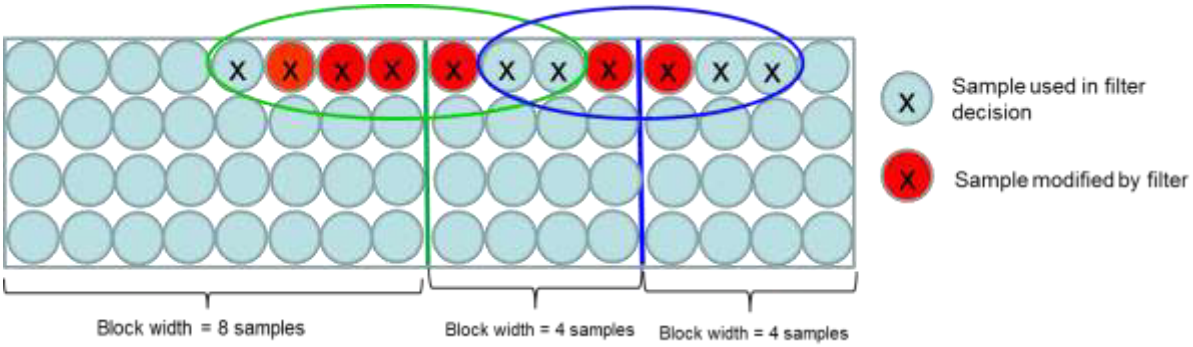
Deblocking is done on sub-block boundaries on 8x8 grid for CUs that use ATMVP.

CE11.3.3 Deblocking for 4 x N and N x 4 CUs JVET-L0225

Sub-tests: 1 and 2

Two solutions are proposed to be studied here:

Sub-test 1:



When block width/height is equal to 4 for vertical edge/horizontal edge respectively, then a maximum of three samples are used in filter decision and only one sample is modified by the filter. i.e. p_2^i is used to replace p_3^i in strong/weak filter condition checks and both strong and weak filter are only allowed to modify p_0 and q_0

Sub-test 2:

This subtests uses filtering on 4x4 grid with only one sample from the block boundary modified if at least one of the adjacent blocks has the corresponding size of 4.

The steps performed for a vertical edge are illustrated below:

- Check if filter on/off condition is evaluated to true i.e. d < *Beta*, where d = d_0 + d_3 and d_0 = d_{p0} + d_{q0} and d_3= d_{p3} + d_{q3}) is evaluated to true for the respective edge. If true then
- Check if the one of the blocks has a block width of 4 samples, if true, then
- Enforce weak filter with maximum one sample modification. Therefore the following condition is checked $|\delta| < 10(t_C)$ $where$ $\delta = (9 * (q_0 - p_0) -$ $(3 * (q_1 - p_1) + 8 \gg 4$, and if the condition is evaluated to true, then samples p_0 and q_0 are modified, otherwise no filtering is applied.

CE11.3.4 Parallel deblocking for 4 x N and N x 4 block boundaries JVET-L0103

To support deblocking all 4xN and Nx4 block boundaries and achieve parallel deblocking for all parallel block boundaries, the following changes are made to HEVC deblocking filters (DF):

1) When HEVC filtering decision and filtering operations are performed, if the CB length at p-side is equal to four, then p_1^i is used to replace p_2^i and p_3^i.
2) When HEVC filtering decision and filtering operations are performed, if the CB length at q-side is equal to four, then q_1^i is used to replace q_2^i and q_3^i.
3) When HEVC normal/weak filtering is used and CB length at p-side or q-side is equal to four, then only samples p_0^i and q_0^i are modified.
   When HEVC strong filtering is used and CB length at p-side or q-side is equal to four, then only samples p_1^i, p_0^i, q_0^i, and q_1^i are modified.

CE11.3.5 Parallel deblocking filter JVET-L0170

Deblocking filter can be processed in parallel because it is not applied to 4x4 TU and PU boundaries. Within this CE, the performance of the parallel deblocking method described in JVET-D0044 is to be studied in terms of subjective quality and computational complexity.

For the luma component, deblocking filtering process is not applied on block boundary of 4xN block for vertical edge and Nx4 block for horizontal edge

For the chroma component, deblocking filtering process is not applied on block boundary of 2xN block for vertical edge and Nx2 block for horizontal edge.

6 Annex A. Subjective test.

A subjective test has been performed before the Macao meeting.

6.1 Test material and coding conditions of the subjective test

A formal subjective test was conducted by the JVET test chair Vittorio Baroncini before the Macao meeting.

The following test sequences and conditions have been used in the subjective test. The sequence length is 10 seconds. The following sequences and QP have been used.

Num	Name	Resolution	fps	Frames	Mode	QP
01	FoodMarket	3840x2160p	60fps	600	RA	39
02	Park Running	3840x2160p	50fps	500	RA	39
03	Campfire	3840x2160p	30fps	300	RA	39
04	Kimono	1920x1080p	24fps	240	LDB	34
05	KristenAndSara	1280x720p	60fps	600	LDP	34

The anchors for the subjective viewing were provided by Huawei (Anand). The test conditions for the subjective test were CTC (VTM-2.0.1) except ALF switched off and the base QP used as in the table above.

Two anchors have been used Anchor 1 with ALF switched on and Anchor-2 with ALF switched off.

6.2 Subjective test description

The results of the formal subjective test of the proposals in this CE are reported in JVET-L0611 [4].

7 Acknowledgements

The CE would like to thank JVET test chair Vittorio Baroncini and EVATech Labs for performing a formal subjective test for evaluating performance of the proposals in this CE before the Macao meeting.

8 References

[1] “JVET common test conditions and software reference configurations”, JVET-K1010.
[2] “Algorithm description for Versatile Video Coding and Test Model 2 (VTM2)”, JVET-K1002.
[3] A. Norkin and A. M. Kotra, "Description of Core Experiment 11 (CE11): Deblocking", JVET-K1031.
[4] V. Baroncini, A. Norkin, and A.M. Kotra, “Subjective assessment of CE11 proposals”, JVET-L0611.

Joint Video Experts Team (JVET)
of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11
12th Meeting: Macao, CN, 3–12 Oct. 2018

Document: JVET-L0072

Title: **CE11: Long deblocking filters for luma (CE11.1.1) and for both luma and chroma (CE11.1.9)**

Status: Input document to JVET

Purpose: Proposal

Author(s) or Contact(s): K. Andersson, Z. Zhang, R. Sjöberg

Tel: +46730371390
Email: kenneth.r.andersson@ericsson.com

Source: Ericsson

Abstract

This contribution proposes long filters and decisions for luma in CE11.1.1 and for both luma and chroma using HEVC luma deblocking for chroma deblocking in CE11.1.9. Deblocking is applied on an 8x8 grid. Long filters for luma can be selected for boundary segments of 16 samples, symmetrically deblocking 7 or 5 samples on respective side of the block boundary for block sizes equal to or larger than 32x32 respectively 16x16. The long filters are designed to not modify a plane/ramp and to be computationally efficient to implement.

The proposed solution in CE11.1.1 is claimed to improve subjective quality at low bitrates. It also provides a luma BD rate impact of -0.01%/-0.01%/0.02%/0.02% for AI/RA/LD/LDP compared to VTM-2.0.1 and -0.03%/-0.05%/0.06%/-0.07% for AI/RA/LD/LDP compared to VTM-2.0.1 when ALF is turned off. Decoding time 99%/100%/100%/101% for AI/RA/LD/LDP compared to VTM-2.0.1 and 99%/101%/101%/101% for AI/RA/LD compared to VTM-2.0.1 when ALF is turned off.

The proposed solution in CE11.1.9 is claimed to improve subjective quality at low bitrates. It also provides a luma BD rate impact of -0.01%/-0.03%/-0.05%/-0.09% for AI/RA/LD/LDP compared to VTM-2.0.1 and -0.03%/-0.07%/-0.04%/-0.15% for AI/RA/LD/LDP compared to VTM-2.0.1 when ALF is turned off. Decoding time 103%/104%/105%/104% for AI/RA/LD/LDP compared to VTM-2.0.1 and 103%/106%/107%/105% for AI/RA/LD compared to VTM-2.0.1 when ALF is turned off.

1 Introduction

The deblocking in the current test model is based on HEVC deblocking and is applied on an 8x8 grid. The proposed long filters and decisions for luma in CE11.1.1 and CE11.1.9 [1] were originally proposed in CfP response JVET-J0012 [2], JVET-K0307 [3], JVET-K0318 [4]. The long filters and decisions on top of the HEVC filters and decisions are described in section 2. In section 3 the use of the HEVC luma filters and decisions for chroma deblocking is described, originally proposed in JVET-K0318 [4]. In section 4 objective results in comparison to VTM-2.0.1 and VTM-2.0.1 without ALF are shown. Finally some conclusions in section 5.

2 Long filters and decisions for luma

The approach is to apply a long filter from a set of long filters to symmetrically filter 5 or 7 samples on respective side of a block boundary for larger blocks and smooth regions. The filters are designed to not modify a plane/ramp and to be computationally efficient to implement.

The filtering of N samples on each side of a line of samples across the block boundary is performed as follows. First pre-filter to obtain three virtual samples, one centered at a midpoint refMiddle between

current block Q and neighbouring block P, one refP centered between two samples in block P and one refQ centered between two samples in block Q. Then two tap linear interpolation between refMiddle and refP to determined filtered samples in block P, and between refMiddle and refQ to determine filtered samples in block Q. This procedure facilitates that a plan/ramp not is modified but that discontinuities between block P and Q can be reduced.

One of the longer filters is used when the strong weak filter decision (identical with HEVC/VTM/BMS) indicates strong filter for 8 lines of a 16 sample segment of a boundary to be filtered and that samples near the boundary are smooth. This gives less frequent use of long filters and also less frequent switching between long filters and weaker filters which are determined for segments of four samples. Filter decisions are shown in the Table 1. The 7 sample filter is used if the current and the neighbouring CUs have a width and height equal or greater than 32 and when 8 samples in current and neighbouring block are smooth. If that is not the case the 5 sample filter is used if the current and neighbouring CUs have a width and height equal or greater than 16 and when 6 samples in current and neighbouring block are smooth. If that is not the case filters in VTM/BMS are used.

Table 1: Filter decisions

Size	Conditions for each of the 8 lines of a 16 sample segment (line 0, 3, 4, 7, 8, 11, 12, 15)
7	$HEVC_{strong\_weak}$&& $\left(\lvert p_0 - 2*p_2 + p_4\rvert + \lvert q_0 - 2*q_2 + q_4\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - p_2 - p_3 + p_5\rvert + \lvert q_0 - q_2 - q_3 + q_5\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - 2*p_3 + p_6\rvert + \lvert q_0 - 2*q_3 + q_6\rvert < \frac{3*\beta}{32}\right)$ && $\left(\lvert p_0 - p_3 - p_4 + p_7\rvert + \lvert q_0 - q_3 - q_4 + q_7\rvert < \frac{3*\beta}{32}\right)$
5	$HEVC_{strong\_weak}$&& $\left(\lvert p_0 - p_4\rvert + \lvert q_0 - q_4\rvert < \frac{\beta}{8}\right)$ && $\left(\lvert p_0 - p_5\rvert + \lvert q_0 - q_5\rvert < \frac{\beta}{8}\right)$
0-3	HEVC filter and decisions apply

Block boundary samples p_i and q_i for i=0 to S-1 are then replaced by linear interpolation as follows:

- $p_i' = (f_i * Middle_s + (64 - f_i) * P_s) + 32) \gg 6, clipped\ to\ p_i \pm$ tc
- $q_i' = (f_i * Middle_s + (64 - f_i) * Q_s + 32) \gg 6), clipped\ to\ q_i \pm$ tc

where f, Middle, P and Q are given below:

Table 2: Filter kernels

S	Filter kernels
7	$f_i = 59 - i * 9,$ $can\ also\ be\ described\ as\ \boldsymbol{f} = \{59,50,41,32,23,14,5\}$ $Middle_7 = (2 * (p_o + q_o) + p_1 + q_1 + p_2 + q_2 + p_3 + q_3 + p_4 + q_4 + p_5 + q_5 + p_6 + q_6 + 8) \gg 4$ $P_7 = (p_6 + p_7 + 1) \gg 1,$ $Q_7 = (q_6 + q_7 + 1) \gg 1$
5	$f_i = 58 - i * 13,$ $can\ also\ be\ described\ as\ \boldsymbol{f} = \{58,45,32,19,6\}$ $Middle_5 = (2 * (p_o + q_o + p_1 + q_1 + p_2 + q_2) + p_3 + q_3 + p_4 + q_4 + 8) \gg 4$

	$P_5 = (p_4 + p_5 + 1) \gg 1, \quad Q_5 = (q_4 + q_5 + 1) \gg 1$

2.1 Complexity analysis

Filtering is performed on an 8x8 grid as for the anchor. All filtering can be performed in parallel. Two cases of long filters are supported, either 7 or 5 samples are modified on each of a block boundary. At most 8 samples are read from both sides of a block boundary when 7 samples of filtering is selected which gives a line buffer requirement of 8 lines. The size of the blocks adjacent to the block boundary is at least 32x32 in the case of 7+7 sample filtering and at least 16x16 for the case of 5+5 sample filtering. Long filtering is performed on segments of 16 samples.

2.1.1 Filtering

First two tap filterings and one 14 or 10 tap filtering. Then bi-linear filtering on respective side of the boundary.

7+7 sample filtering of one line requires in total 120 operations including rounding (24 multiplications, 46 additions, 22 shifts, 28 cmp).

5+5 sample filtering of one line requires in total 88 operations including rounding (16 multiplications, 34 additions, 18 shifts, 20 cmp).

Worst case: 120 operations for block size P and Q of 32x32.

2.1.2 Decision

In case HEVC strong filter decision is true, long filter decisions are checked.

7+7 sample filtering decision for a line (8 of 16 lines are checked):

- Compare width and height for both current and neighbouring block with a threshold.
- Smoothness: 20 operations (12 cmp, 24 additions, 4 shifts)/2
- Total operations per line: 24

5+5 sample filtering decisions for a line (8 of 16 lines are checked):

- Compare width and height for both current and neighbouring block with a threshold.
- Smoothness: 6 operations (6 cmp, 6 additions)/2
- Total operations per line: 10

Worst case: 30 operations per line for block size P and Q of 32x32.

3 Deblocking for chroma

The HEVC luma filters and decisions are used to deblock chroma with some adaptations but with same 8x8 granularity as before.

First adaptation is to filter chroma also for inter blocks when bs=1, re-using the luma decision for motion check but adding specific cbf check for respective chroma component.

Second adaptation is to limit the maximum deblocking filter length on P side (neighbouring CU) with the length of neighbouring block orthogonal to the boundary divided by 4 and for Q side (current CU) the length of the current block orthogonal to the boundary divided by 4.

lengthP = edge==EDGE_HOR ? (CUP_height_chroma >> 2) : (CUP_width_chroma >>2)

lengthQ = edge==EDGE_HOR ? (CUQ_height_chroma >> 2) : (CUQ_width_chroma >>2)

Filtering of the boundary is only applied if both sides have a maximum deblocking filter length larger than 0. The HEVC strong filter (reading 4 samples and modifying 3 samples on respective side) is only applied if both sides have a maximum deblocking filter length larger than 2.

Thirdly the decision for strong/weak filter and number of samples to filter is determined for each line instead of for a group of 4 sample segment as in HEVC. Inter blocks are only filtered if Laplacian gradients near the boundary on respective side is below the structure threshold “beta” as in HEVC but for intra blocks that check is by passed.

Fourthly is “beta” parameter divided by 8 compared to what is used for luma. This makes the use of more than 1 pixel to filter only happening when there is very little structure near the boundary.

3.1 Complexity analysis

Filtering is performed on an 8x8 grid as for the anchor. All filtering can be performed in parallel. Same filter combinations (HEVC strong and weak filter) as for the anchor but for luma. At most 4 samples is read from both sides of a block boundary when 3 samples of filtering is selected (HEVC strong filter) which gives a line buffer requirement of 4 lines. The size of the blocks adjacent to the block boundary is at least 4xN in the case of 1 sample weak filtering, 8xN in the case of 2 sample weak filtering and at least 16xN for the case of 3 sample strong filtering. Filtering is performed for each line of chroma samples.

3.1.1 Filtering

3 sample filtering: HEVC strong filter: 52 operations (2 multiplications, 28 additions, 10 shifts, 12 cmp)

1 or 2 simple filtering: HEVC weak filter: 33 operations (2 multiplications, 16 additions, 5 shifts, 8 cmp)

Worst case: 52 operations per line for block size P 16xN and Q 16xN

3.1.2 Decisions

HEVC d decision: 4 operations (5 additions, 3 cmp)/2

HEVC strong weak decision: 10 operations (4 additions, 6 cmp)

HEVC weak decision: 4 operations (4 cmp)

Worst case: 18 operations per line for block size P 16xN and Q 16xN

4 Results

The simulations were performed using the CTC with the inclusion of LDP. We also tested the performance with ALF disabled. The decoding times for the anchor and test configurations were measured using the same machines, but encoding times were measured on different machines which make the encoding times unreliable. However, since the decoding times are magnitudes shorter and the deblocking filter is done once per picture for both encoder and decoder, we conclude the encoding time difference is negligible.

The results indicate that that the objective performance of the proposed long filters for luma as in CE11.1.1 is similar to the anchor. It has slightly better objective performance when ALF is disabled than when it is enabled. Encoding time is about 100% and decoding time is about 101% compared to the anchor.

For the case of both long luma filters and longer chroma filters as in CE11.1.9 the objective performance is slightly better than the anchor. Also here the objective performance is slightly better when ALF is disabled than when it is enabled. Encoding time is about 100% and decoding time 106% compared to the anchor.

4.1 Objective performance for CE11.1.1

All Intra Main10									
VTM-2.0.1					VTM-2.0.1 ALF=0				
Y	U	V	EncT	DecT	Y	U	V	EncT	DecT

Class A1	-0.02%	-0.01%	0.00%	106%	98%	-0.09%	-0.02%	0.00%	103%	98%
Class A2	-0.01%	0.00%	0.00%	108%	99%	-0.01%	-0.01%	0.00%	94%	100%
Class B	0.00%	0.00%	0.00%	108%	98%	-0.01%	0.00%	0.00%	100%	99%
Class C	0.00%	0.00%	0.00%	107%	101%	0.00%	0.00%	0.00%	101%	99%
Class E	-0.06%	-0.01%	0.00%	98%	100%	-0.06%	-0.01%	0.00%	98%	100%
Overall	-0.01%	0.00%	0.00%	106%	99%	-0.03%	-0.01%	0.00%	99%	99%
Class D	0.00%	0.00%	0.00%	104%	101%	0.00%	0.00%	0.00%	104%	101%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Random Access Main 10									
	VTM-2.0.1					**VTM-2.0.1 ALF=0**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.12%	0.01%	0.03%	99%	100%	-0.24%	-0.01%	0.02%	99%	104%
Class A2	0.00%	-0.10%	-0.08%	99%	101%	-0.04%	0.05%	0.04%	101%	98%
Class B	0.03%	-0.01%	-0.09%	97%	100%	-0.01%	0.04%	0.00%	100%	102%
Class C	0.02%	0.06%	-0.03%	100%	102%	0.02%	0.02%	-0.07%	101%	101%
Class E										
Overall	-0.01%	-0.01%	-0.05%	99%	100%	-0.05%	0.03%	-0.01%	100%	101%
Class D	0.03%	-0.04%	0.01%	97%	100%	-0.01%	0.00%	0.08%	99%	101%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay B Main10									
	VTM-2.0.1					**VTM-2.0.1 ALF=0**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.08%	-0.17%	0.00%	103%	100%	0.06%	0.09%	-0.10%	102%	101%
Class C	0.03%	-0.13%	-0.10%	102%	100%	0.01%	-0.15%	-0.13%	100%	102%
Class E	-0.07%	0.37%	-0.07%	95%	102%	0.14%	-0.09%	0.29%	96%	102%
Overall	0.02%	-0.02%	-0.05%	101%	100%	0.06%	-0.04%	-0.01%	100%	101%
Class D	0.00%	0.03%	-0.49%	104%	100%	-0.03%	-0.48%	-0.06%	103%	99%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay P Main10									
	VTM-2.0.1					**VTM-2.0.1 ALF=0**				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.01%	0.17%	0.18%	102%	100%	-0.06%	-0.03%	-0.09%	101%	100%
Class C	0.02%	0.02%	0.03%	100%	101%	0.03%	0.12%	0.00%	101%	100%
Class E	0.03%	0.00%	0.60%	107%	101%	-0.22%	0.23%	-0.20%	100%	102%
Overall	0.02%	0.08%	0.24%	102%	101%	-0.07%	0.08%	-0.09%	101%	101%

Class D	0.04%	0.05%	-0.32%	103%	101%	-0.01%	-0.40%	-0.18%	102%	101%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#NUM!	#VALUE!	#VALUE!	#VALUE!	#NUM!	#NUM!

4.2 Objective performance for CE11.1.9

	All Intra Main10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.02%	-0.90%	-0.93%	101%	105%	-0.09%	-1.36%	-1.24%	104%	103%
Class A2	0.00%	-0.41%	-0.32%	104%	103%	0.00%	-0.32%	-0.18%	97%	103%
Class B	0.00%	-0.72%	-0.80%	105%	103%	-0.01%	-0.66%	-0.69%	95%	103%
Class C	0.00%	-0.45%	-0.46%	102%	102%	0.00%	-0.20%	-0.18%	102%	102%
Class E	-0.05%	-0.68%	-0.53%	100%	104%	-0.06%	-0.78%	-0.65%	97%	104%
Overall	-0.01%	-0.63%	-0.62%	103%	103%	-0.03%	-0.64%	-0.58%	99%	103%
Class D	0.00%	-0.42%	-0.50%	106%	101%	0.00%	-0.10%	-0.09%	97%	103%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Random Access Main 10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1	-0.13%	-2.41%	-3.28%	98%	103%	-0.27%	-3.46%	-4.09%	99%	108%
Class A2	-0.03%	-1.62%	-1.47%	99%	105%	-0.06%	-1.63%	-1.35%	101%	105%
Class B	0.01%	-1.97%	-2.18%	97%	104%	-0.02%	-2.10%	-2.32%	99%	106%
Class C	-0.01%	-1.38%	-1.55%	99%	105%	0.01%	-1.79%	-1.96%	99%	105%
Class E										
Overall	-0.03%	-1.83%	-2.09%	98%	104%	-0.07%	-2.20%	-2.39%	99%	106%
Class D	0.00%	-1.21%	-1.25%	99%	105%	0.02%	-1.63%	-1.69%	95%	104%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay B Main10									
	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	0.00%	-1.72%	-1.98%	101%	104%	-0.01%	-1.66%	-2.24%	103%	107%
Class C	-0.06%	-1.46%	-1.57%	105%	105%	-0.07%	-1.29%	-1.44%	100%	107%
Class E	-0.13%	-3.09%	-3.03%	95%	107%	-0.04%	-2.84%	-2.45%	95%	108%
Overall	-0.05%	-1.98%	-2.10%	101%	105%	-0.04%	-1.83%	-2.03%	100%	107%
Class D	-0.01%	-0.34%	-0.82%	100%	104%	-0.06%	-0.80%	-0.84%	102%	103%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#DIV/0!	#VALUE!	#VALUE!	#VALUE!	#DIV/0!	#DIV/0!

	Low delay P Main10

	VTM-2.0.1					VTM-2.0.1 ALF=0				
	Y	U	V	EncT	DecT	Y	U	V	EncT	DecT
Class A1										
Class A2										
Class B	-0.12%	-2.22%	-2.16%	101%	103%	-0.15%	-2.65%	-2.98%	103%	106%
Class C	-0.07%	-1.34%	-1.38%	99%	105%	-0.06%	-1.17%	-1.49%	100%	105%
Class E	-0.09%	-3.15%	-2.47%	101%	104%	-0.26%	-3.28%	-3.57%	96%	103%
Overall	-0.09%	-2.16%	-1.98%	100%	104%	-0.15%	-2.32%	-2.63%	100%	105%
Class D	-0.01%	-1.03%	-1.26%	101%	104%	-0.08%	-1.71%	-1.32%	102%	98%
Class F (optional)	#VALUE!	#VALUE!	#VALUE!	#NUM!	#NUM!	#VALUE!	#VALUE!	#VALUE!	#NUM!	#NUM!

5 Conclusions

We propose to adopt the proposed long filters for luma to combat very visual artifacts for large blocks. We also propose to use HEVC luma deblocking for chroma deblocking but that mainly improves subjective performance on the Campfire sequence.

6 References

[1] A. Norkin, A. Meher Kotra, "Description of Core Experiment 11 (CE11): Deblocking", JVET-K1031, Ljubljana, July 2018

[2] R. Sjöberg, K. Andersson, R. Yu, Z. Zhang, P. Wennersten, J. Lainema, A. Hallapuro, A. Aminlou, M. Hannuksela, R. Ghaznavi-Youvalari, J. Ridge, "Description of SDR and HDR video coding technology proposal by Ericsson and Nokia", JVET-J0012, San Diego, April 2018

[3] K. Andersson, Z. Zhang, R. Sjöberg, "CE2-2.1.1: Long deblocking filters and fixes", JVET-K0307, Ljubljana, July 2018

[4] K. Andersson, Z. Zhang, R. Sjöberg, "CE2-2.1.1 related: HEVC luma filters and decisions for chroma deblocking", JVET-K0318, Ljubljana, July 2018

[5] J. Boyce, K. Suehring, X. Li, V. Seregin, "JVET common test conditions and software reference configurations", JVET-K1010, Ljubljana, July 2018

7 Patent rights declaration(s)

Ericsson may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).

Joint Video Experts Team (JVET)
of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11
12th Meeting: Macao, CN, 3–12 Oct. 2018

Document: JVET-L1001-v3

Title:	**Versatile Video Coding (Draft 3)**		
Status:	Output document approved by JVET		
Purpose:	Draft text of video coding specification		
Author(s) or Contact(s):	Benjamin Bross Jianle Chen Shan Liu	Email:	firstname.lastname@hhi.fraunhofer.de jianle.chen@huawei.com shanl@tencent.com
Source:	Editors		

Abstract

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

Draft 1 of Versatile Video Coding.

Ed. Notes:

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

Draft 2 of Versatile Video Coding.

Ed. Notes:

- Incorporated JVET-K0230: Separate trees for intra slices (without multi-DMs) with an implicit split to 64x64;
- Incorporated JVET-K0556: Prohibit ternary split of something bigger than 64 in width or height (and not send the bit to indicate ternary type at that level).
- Incorporated JVET-K0351 (test c): Keep only the TT restriction (preventing binary split with same orientation in center partition of the ternary split)
- Incorporated JVET-K0554: Implicit splitting at picture boundaries and ensure MinQTSize at boundary splits
- Fixed bug #65 typos and unused variables in section 6.4
- Fixed bug #67 implicit vertical BT split at picture boundary issue
- Incorporated JVET-K0072: Dependent quantization with fallback switch at the picture level and modified entropy coding supporting dependent quantization including:
    - adapted scaling to non-square transform blocks,
    - added binarization process for abs_remainder,
    - specified CoeffMin and CoeffMax with fixed values,
    - added 0-th order Exp-Golomb code parsing process.
- Incorporated JVET-K0310: Sign data hiding (can only be used when dependent quantization is disabled).
- Incorporated JVET-K0529: Intra prediction using 3MPM on 67 prediction modes (Planar, DC and 65 angular modes)
- Incorporated JVET-K0122: DC prediction without division.
- Incorporated JVET-K0500: Wide-angle intra prediction.

- Incorporated JVET-K0063: Position-dependent intra prediction combination
- Incorporated JVET-K0190: Cross-component linear model intra prediction
- Incorporated multiple transfrom selection (MTS) for both intra and inter, each controlled by an SPS flag.
- Incorporated transform skip.
- Fixed bug #68 various typos
- Fixed bug #71 various typos
- Fixed bug #72 on CCLM
- Incorporated JVET-K0357: adaptive motion vector resolution (AMVR)
- Incorpor,ated JVET-K0565: affine motion compensation (MC) including:
  - JVET-K0052: Affine merge bug fix
  - JVET-K0184: Affine MC (CE4.1.1a 4x4 fixed subblock size).
  - JVET-K0337: Affine MC coding and models (4.1.3a, affine MVP list construction, and 4.1.3b, MV difference coding, and 4.1.3c, 4/6 parameter model, no slice level switch).
  - JVET-K0367/JVET-K0052/JVET-K0103: Restriction of affine merge mode to CU sizes >= 8x8
- Incorporated 1/16 motion compensation (MC) including:
  - 1/16 MV storage
  - 1/16 merge and affine MVs
  - MVDs in AMVR accuracy (1/4,1,4) shifted to 1/16
  - Inter MVP candidates rounded to AMVR accuracy (1/4,1,4) and shifted to 1/16
  - 1/16 luma and 1/32 chroma interpolation filters
- Incorporated subblock-based temporal merging candidates with 8x8 motion vector storage (JVET-K0346).
- Incorporated JVET-K0371: 4x4 block classification based Adaptive Loop Filter (ALF).
- Fixed bug #75 regarding a bottom and right boundary partition issue.
- Fixed bug #90 typos in copying the control point vectors to temporal notion vectors.
- Fixed bug #86 in intra reference sample filtering.
- Incorporated JVET-K0325: High Level Syntax (HLS) starting point.
- Fixed bug #82 on zeroing-out high frequency transform coefficients for larger TUs (>32x32).
- Fixed bug #85 on MTS index coding.

Draft 3 of Versatile Video Coding.

Ed. Notes:
- Fixed bug #92 missing MV clipping in interplolation process.
- Incorporated JVET-L0664: disabling 5x5 ALF for luma component.
- Incorporated JVET-L0082/JVET-L0083: reduction of bits for ALF coefficients.
- Incorporated JVET-L0147: subsampling of ALF classifiers.
- Incorporated SAO as found in HEVC.
- Incorporated deblocking filter as found in HEVC with the following modifications:
  - adapt processing to dual tree CTU partitioning,
  - replace RQT based transfrom block processing with implicit TU split for large blocks (JVET-K0307, JVET-K0237, JVET-K0369, JVET-K0232, JVET-K0315),
  - replace prediction blocks with coding subblocks (JVET-L0074),
  - only apply deblocking on transfrom block / coding subblock edges if CU is aligned on 8x8 grid in the respecitive direction.
- Incorporated JVET-L0410: tc table fix
- Incorporated JVET-L0414: luma adaptive deblocking filter.
- Fixed bug #97 mismatch between agreed ALF text JVET-K0564 and draft by removing alf_chroma_ctb_present_flag.
- Incorporated JVET-L0285: 8-bit transform matrix.
- Incorporated JVET-L0118: MTS index fix.
- Incorporated JVET-L0059: remove dependency on number non-zero coeff. for MTS index signalling.
- Incorporated JVET-L0362: QP signalling.
- Incorporated JVET-L0428: delta QP and chroma QP offset for dual tree.
- Incorporated JVET-K0251: increase maximum QP value from 51 to 63 (including fix from JVET-L0553).
- Incorporated JVET-L0217: fix relation between QT/BT/TT syntax elements.
- Incorporated JVET-L0678: QT/BT/TT syntax overriding in slice header.
- Incorporated JVET-L0081: BT/TT constraint.
- Incorporated JVET-L0191: cclm simplification using min and max values.
- Incorporated JVET-L0136: cclm with line buffer restriction.
- Incorporated JVET-L0340: multi-directional cclm.

- Incorporated JVET-L0053/JVET-L0272: modified chroma direct mode.
- Incorporated JVET-L0628: intra 4-tap interpolation filter.
- Incorporated JVET-L0165: intra 6 MPMs.
- Incorporated JVET-L0283: multi-line intra prediction.
- Incorporated JVET-L0279: unification of intra angular prediction.
- Incorporated PCM mode from HEVC and JVET-L0209: PCM mode with dual tree partition.

CONTENTS

Page

Abstract i

1 Scope 1

2 Normative references 1
- 2.1 Identical Recommendations | International Standards 1
- 2.2 Paired Recommendations | International Standards equivalent in technical content 1
- 2.3 Additional references 1

3 Definitions 1

4 Abbreviations 5

5 Conventions 8
- 5.1 General 8
- 5.2 Arithmetic operators 8
- 5.3 Logical operators 8
- 5.4 Relational operators 8
- 5.5 Bit-wise operators 8
- 5.6 Assignment operators 9
- 5.7 Range notation 9
- 5.8 Mathematical functions 9
- 5.9 Order of operation precedence 10
- 5.10 Variables, syntax elements and tables 11
- 5.11 Text description of logical operations 12
- 5.12 Processes 13

6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 14
- 6.1 Bitstream formats 14
- 6.2 Source, decoded and output picture formats 14
- 6.3 Partitioning of pictures, slices, CTUs 16
  - 6.3.1 Partitioning of pictures into slices 16
  - 6.3.2 Block, quadtree and multi-type tree structures 17
  - 6.3.3 Spatial or component-wise partitionings 17
- 6.4 Availability processes 17
  - 6.4.1 Allowed binary split process 18
  - 6.4.2 Allowed ternary split process 19
- 6.5 Scanning processes 19
  - 6.5.1 CTB raster and scanning process 19
  - 6.5.2 Up-right diagonal scan order array initialization process 19

7 Syntax and semantics 21
- 7.1 Method of specifying syntax in tabular form 21
- 7.2 Specification of syntax functions and descriptors 22
- 7.3 Syntax in tabular form 23
  - 7.3.1 NAL unit syntax 23
  - 7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 24
  - 7.3.3 Slice header syntax 27
  - 7.3.4 Slice data syntax 29
- 7.4 Semantics 40
  - 7.4.1 General 40
  - 7.4.2 NAL unit semantics 40
  - 7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 42
  - 7.4.4 Slice header semantics 49
  - 7.4.5 Slice data semantics 54

8 Decoding process 65
- 8.1 General decoding process 65
- 8.2 Decoding process for coding units coded in intra prediction mode 65
  - 8.2.1 General decoding process for coding units coded in intra prediction mode 65

8.2.2 Derivation process for luma intra prediction mode 66
8.2.3 Derivation process for chroma intra prediction mode 70
8.2.4 Decoding process for intra blocks 70
8.3 Decoding process for coding units coded in inter prediction mode 87
8.3.1 General decoding process for coding units coded in inter prediction mode 87
8.3.2 Derivation process for motion vector components and reference indices 88
8.3.3 Derivation process for affine motion vector components and reference indices 108
8.3.4 Decoding process for inter blocks 118
8.3.5 Decoding process for the residual signal of coding blocks coded in inter prediction mode 124
8.4 Scaling, transformation and array construction process 125
8.4.1 Derivation process for quantization parameters 125
8.4.2 Scaling and transformation process 127
8.4.3 Scaling process for transform coefficients 127
8.4.4 Transformation process for scaled transform coefficients 128
8.4.5 Picture reconstruction process 136
8.5 In-loop Filter Process 136
8.5.1 General 136
8.5.2 Deblocking filter process 136
8.5.3 Sample adaptive offset process 152
8.5.4 Adaptive loop filter process 154
9 Parsing process 158
9.1 General 158
9.2 Parsing process for k-th order Exp-Golomb codes 159
9.2.1 General 159
9.2.2 Mapping process for signed Exp-Golomb codes 160
9.3 Parsing process for truncated unary codes 161
9.4 Parsing process for truncated binary codes 161
9.5 CABAC parsing process for slice segment data 161
9.5.1 General 161
9.5.2 Initialization process 161
9.5.3 Binarization process 161
9.5.4 Decoding process flow 167

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Versatile video coding

1 Scope

This Recommendation | International Standard specifies versatile video coding.

2 Normative references

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

2.1 Identical Recommendations | International Standards

– None

2.2 Paired Recommendations | International Standards equivalent in technical content

– None

2.3 Additional references

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

3 Definitions

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

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

3.1 **access unit**: A set of *NAL units* that are associated with each other according to a specified classification rule, are consecutive in *decoding order,* and contain exactly one *coded picture*.

3.2 **AC transform coefficient**: Any *transform coefficient* for which the *frequency index* in at least one of the two dimensions is non-zero.

3.3 **bin**: One bit of a *bin string*.

3.4 **binarization**: A set of *bin strings* for all possible values of a *syntax element*.

3.5 **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.

3.6 **binary split**: A split of a rectangular MxN *block* of samples into two *blocks* where a vertical split results in a first (M / 2)xN *block* and a second (M / 2)xN *block*, and a horizontal split results in a first Mx(N / 2) *block* and a second Mx(N / 2) *block*.

3.7 **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.

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

3.9 **bitstream**: A sequence of bits, in the form of a *NAL unit stream* or a *byte stream*, that forms the representation of *coded pictures* and associated data forming one or more coded video sequences *(CVSs)*.

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

3.11 **byte**: A sequence of 8 bits, within which, when written or read as a sequence of bit values, the left-most and right-most bits represent the most and least significant bits, respectively.

3.12 **byte-aligned**: A position in a *bitstream* is byte-aligned when the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream*, and a bit or *byte* or *syntax element* is said to be byte-aligned when the position at which it appears in a *bitstream* is byte-aligned.

3.13 **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex TBD.

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

3.15 **chroma**: An adjective, represented by the symbols Cb and Cr, specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours.

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

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

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

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

3.19 **coding block**: An MxN *block* of samples for some values of M and N such that the division of a *CTB* into *coding blocks* is a *partitioning*.

3.20 **coding tree block (CTB)**: An NxN *block* of samples for some value of N such that the division of a *component* into *CTBs* is a *partitioning*.

3.21 **coding tree unit (CTU)**: A *CTB* of *luma* samples, two corresponding *CTBs* of *chroma* samples of a *picture* that has three sample arrays, or a *CTB* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.

3.22 **coding unit (CU)**: A *coding block* of *luma* samples, two corresponding *coding blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.

3.23 **component**: An array or single sample from one of the three arrays (*luma* and two *chroma*) that compose a *picture* in 4:2:0, 4:2:2, or 4:4:4 colour format or the array or a single sample of the array that compose a *picture* in monochrome format.

3.24 **context variable**: A variable specified for the *adaptive binary arithmetic decoding process* of a *bin* by an equation containing recently decoded *bins*.

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

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

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

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

3.29 **emulation prevention byte**: A *byte* equal to 0x03 that is present within a *NAL unit* when the *syntax elements* of the *bitstream* form certain patterns of *byte* values in a manner that ensures that no sequence of consecutive *byte-aligned bytes* in the *NAL unit* can contain a *start code prefix*.

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

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

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

3.33 **frequency index**: A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to the application of a *transform* in the *decoding process.*

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

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

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

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

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

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

3.39 **intra random access point (IRAP) access unit**: An *access unit* in which the *coded picture* is an *IRAP picture*.

3.40 **intra random access point (IRAP) picture**: A *coded picture* for which each *VCL NAL unit* has nal_unit_type equal to IRAP_NUT.

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

3.42 **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.

3.43 **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this Specification, or the value of a *transform coefficient* prior to *scaling*.

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

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

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

3.46 **luma**: An adjective, represented by the symbol or subscript Y or L, specifying that a sample array or single sample is representing the monochrome signal related to the primary colours.

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

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

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

3.48 **motion vector**: A two-dimensional vector used for *inter prediction* that provides an offset from the coordinates in the *decoded picture* to the coordinates in a *reference picture*.

3.49 **multi-type tree**: A *tree* in which a parent node can be split either into two child nodes using a *binary split* or into three child nodes using a *ternary split*, each of which may become parent node for another split into either two or three child nodes.

3.50 **must**: A term that is used in expressing an observation about a requirement or an implication of a requirement that is specified elsewhere in this Specification (used exclusively in an *informative* context).

3.51 **network abstraction layer (NAL) unit**: A *syntax structure* containing an indication of the type of data to follow and *bytes* containing that data in the form of an *RBSP* interspersed as necessary with *emulation prevention bytes*.

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

3.53 **non-IRAP picture**: A *coded picture* for which each *VCL NAL unit* has nal_unit_type equal to NON_IRAP_NUT.

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

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

3.56 **output order**: The order in which the *decoded pictures* are output from the *decoded picture buffer* (for the *decoded pictures* that are to be output from the *decoded picture buffer*).

3.57 **parameter**: A *syntax element* of a *sequence parameter set (SPS)* or *picture parameter set (PPS)*, or the second word of the defined term *quantization parameter*.

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

3.59 **picture**: An array of *luma* samples in monochrome format or an array of *luma* samples and two corresponding arrays of *chroma* samples in 4:2:0, 4:2:2, and 4:4:4 colour format.

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

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

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

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

3.63 **prediction process**: The use of a *predictor* to provide an estimate of the data element (e.g., sample value or motion vector) currently being decoded.

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

3.65 **predictor**: A combination of specified values or previously decoded data elements (e.g., sample value or motion vector) used in the *decoding process* of subsequent data elements.

3.66 **profile**: A specified subset of the syntax of this Specification.

3.67 **pulse code modulation (PCM)**: Coding of the samples of a *block* by directly representing the sample values without *prediction* or application of a *transform*.

3.68 **quadtree**: A *tree* in which a parent node can be split into four child nodes, each of which may become parent node for another split into four child nodes.

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

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

3.71 **raster scan**: A mapping of a rectangular two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the first top row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc., rows of the pattern (going down) each scanned from left to right.

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

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

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

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

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

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

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

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

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

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

3.80 **residual**: The decoded difference between a *prediction* of a sample or data element and its decoded value.

3.81 **scaling**: The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.

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

3.83 **shall**: A term used to express mandatory requirements for conformance to this Specification.

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

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

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

3.86 **slice**: An integer number of *CTUs* ordered consecutively in the *raster scan* and contained in a single *NAL unit*.

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

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

3.89 **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit*.

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

3.90 **string of data bits (SODB)**: A sequence of some number of bits representing *syntax elements* present within a *raw byte sequence payload* prior to the *raw byte sequence payload stop bit*, where the left-most bit is considered to be the first and most significant bit, and the right-most bit is considered to be the last and least significant bit.

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

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

3.93 **ternary split**: A split of a rectangular MxN *block* of samples into three *blocks* where a vertical split results in a first (M / 4)xN *block*, a second (M / 2)xN *block*, a third (M / 4)xN *block*, and a horizontal split results in a first Mx(N / 4) *block*, a second Mx(N / 2) *block*, a third Mx(N / 4) *block*.

3.94 **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.

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

3.96 **transform coefficient**: A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in a *transform* in the *decoding process*.

3.97 **transform coefficient level**: An integer quantity representing the value associated with a particular two-dimensional frequency index in the *decoding process* prior to *scaling* for computation of a *transform coefficient* value.

3.98 **transform unit (TU)**: A *transform block* of *luma* samples and two corresponding *transform blocks* of *chroma* samples of a *picture* and *syntax structures* used to transform the *transform block* samples.

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

3.100 **unspecified**: A term that may be used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.

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

4 Abbreviations

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

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

ALF	Adaptive Loop Filter
AMVR	Adaptive Motion Vector Resolution
B	Bi-predictive
CABAC	Context-based Adaptive Binary Arithmetic Coding
CB	Coding Block
CBR	Constant Bit Rate
CPB	Coded Picture Buffer
CRC	Cyclic Redundancy Check
CTB	Coding Tree Block
CTU	Coding Tree Unit
CU	Coding Unit
CVS	Coded Video Sequence
DPB	Decoded Picture Buffer
EG	Exponential-Golomb
EGk	k-th order Exponential-Golomb
FCC	Federal Communications Commission (of the United States)
FIFO	First-In, First-Out
FIR	Finite Impulse Response
FL	Fixed-Length
GBR	Green, Blue and Red
I	Intra
IRAP	Intra Random Access Point
LPS	Least Probable Symbol
LSB	Least Significant Bit
MPS	Most Probable Symbol
MSB	Most Significant Bit
MTS	Multiple Transform Selection
MVP	Motion Vector Prediction
NAL	Network Abstraction Layer
NTSC	National Television System Committee (of the United States)
P	Predictive
PCM	Pulse Code Modulation
POC	Picture Order Count
PPS	Picture Parameter Set
QP	Quantization Parameter
RBSP	Raw Byte Sequence Payload
RGB	Same as GBR
RPS	Reference Picture Set
SAO	Sample Adaptive Offset
SAR	Sample Aspect Ratio
SEI	Supplemental Enhancement Information
SMPTE	Society of Motion Picture and Television Engineers
SODB	String Of Data Bits
SPS	Sequence Parameter Set
TR	Truncated Rice
UCS	Universal Coded Character Set

UTF	UCS Transmission Format
VBR	Variable Bit Rate
VCL	Video Coding Layer

5 Conventions

5.1 General

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

5.2 Arithmetic operators

The following arithmetic operators are defined as follows:

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

5.3 Logical operators

The following logical operators are defined as follows:

x && y	Boolean logical "and" of x and y
x || y	Boolean logical "or" of x and y
!	Boolean logical "not"
x ? y : z	If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

5.4 Relational operators

The following relational operators are defined as follows:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
= =	Equal to
!=	Not equal to

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

5.5 Bit-wise operators

The following bit-wise operators are defined as follows:

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

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

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

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

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

5.6 Assignment operators

The following arithmetic operators are defined as follows:

= Assignment operator

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

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

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

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

5.7 Range notation

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

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

5.8 Mathematical functions

The following mathematical functions are defined:

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

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

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

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

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

$$\mathrm{Clip1}_Y(x) = \mathrm{Clip3}(0, (1 << \mathrm{BitDepth}_Y) - 1, x) \tag{5-6}$$

$$Clip1_C(x) = Clip3(0, (1 << BitDepth_C) - 1, x) \quad (5\text{-}7)$$

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

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

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

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

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

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

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

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

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

$$Round(x) = Sign(x) * Floor(Abs(x) + 0.5) \quad (5\text{-}17)$$

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

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

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

$$Swap(x, y) = (y, x) \quad (5\text{-}21)$$

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

5.9 Order of operation precedence

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

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

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

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

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

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

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

5.10 Variables, syntax elements and tables

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

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

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

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

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

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

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column

(horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[x][y] or as s_{yx}. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[x].

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

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

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

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

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

5.11 Text description of logical operations

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

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

may be described in the following manner:

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

- If condition 0, statement 0
- Otherwise, if condition 1, statement 1
- ...
- Otherwise (informative remark on remaining condition), statement n

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

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

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

may be described in the following manner:

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

- If all of the following conditions are true, statement 0:
  - condition 0a
  - condition 0b
- Otherwise, if one or more of the following conditions are true, statement 1:

  - condition 1a
  - condition 1b
- ...
- Otherwise, statement n

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

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

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

5.12 Processes

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

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

- If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.
- Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

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

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

6.1 Bitstream formats

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

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

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

6.2 Source, decoded and output picture formats

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

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

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

– Luma (Y) only (monochrome).
– Luma and two chroma (YCbCr or YCgCo).
– Green, blue, and red (GBR, also known as RGB).
– Arrays representing other unspecified monochrome or tri-stimulus colour samplings (for example, YZX, also known as XYZ).

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

The variables SubWidthC and SubHeightC are specified in Table 6-1, depending on the chroma format sampling structure, which is specified through chroma_format_idc and separate_colour_plane_flag. Other values of chroma_format_idc, SubWidthC and SubHeightC may be specified in the future by ITU-T | ISO/IEC.

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

chroma_format_idc	separate_colour_plane_flag	Chroma format	SubWidthC	SubHeightC
0	0	Monochrome	1	1
1	0	4:2:0	2	2
2	0	4:2:2	2	1
3	0	4:4:4	1	1
3	1	4:4:4	1	1

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

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

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

In 4:4:4 sampling, depending on the value of separate_colour_plane_flag, the following applies:

– If separate_colour_plane_flag is equal to 0, each of the two chroma arrays has the same height and width as the luma array.
– Otherwise (separate_colour_plane_flag is equal to 1), the three colour planes are separately processed as monochrome sampled pictures.

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

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

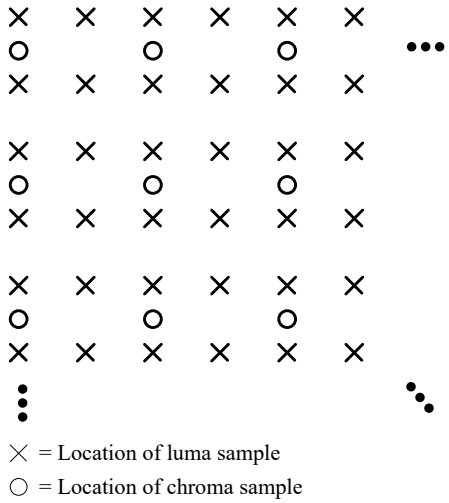


Figure 6-1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture

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

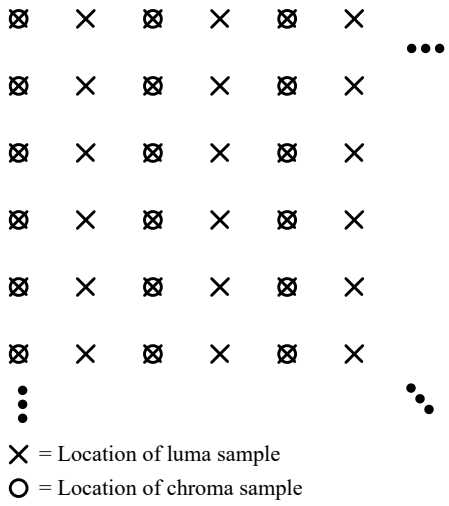


Figure 6-2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture

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

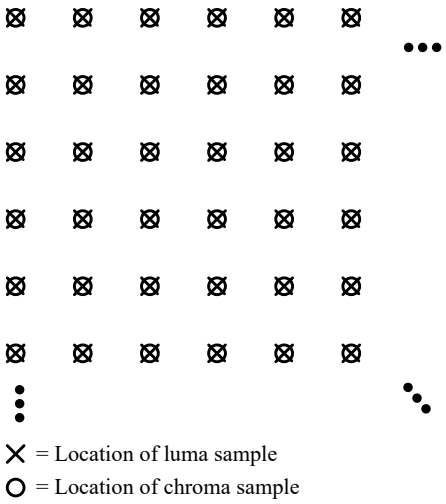


Figure 6-3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture

6.3 Partitioning of pictures, slices, CTUs

6.3.1 Partitioning of pictures into slices

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

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

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

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

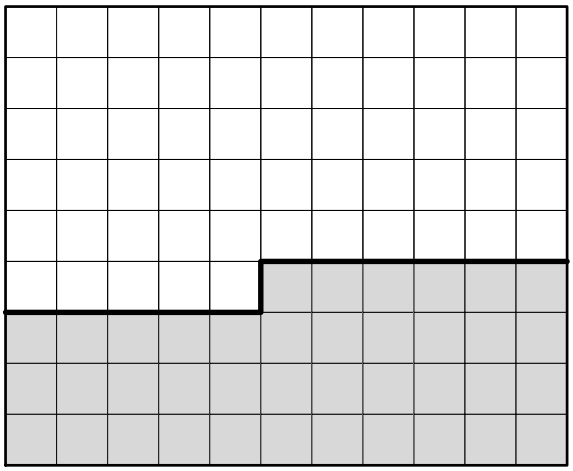


Figure 6-4 – A picture with 11 by 9 luma CTUs that is partitioned into two slices (informative)

6.3.2 Block, quadtree and multi-type tree structures

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

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

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

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

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

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

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

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

6.3.3 Spatial or component-wise partitionings

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

– The division of each picture into components

– The division of each component into CTBs

– The division of each picture into slices

– The division of each slice into CTUs

– The division of each CTU into CTBs

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

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

– The division of each coding unit into transform units

– The division of each coding unit into coding blocks

– The division of each coding block into transform blocks

– The division of each transform unit into transform blocks.

6.4 Availability processes

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

6.4.1 Allowed binary split process

Inputs to this process are:

– a binary split mode btSplit,

– a coding block width cbWidth,

– a coding block height cbHeight,

– a location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture,

– a multi-type tree depth mttDepth,

– a maximum multi-type tree depth with offset maxMttDepth,

– a maximum binary tree size maxBtSize,

– a partition index partIdx.

Output of this process is the variable allowBtSplit.

Table 6-2 – Specification of parallelTtSplit and cbSize based on btSplit.

	btSplit = = SPLIT_BT_VER	**btSplit = = SPLIT_BT_HOR**
parallelTtSplit	SPLIT_TT_VER	SPLIT_TT_HOR
cbSize	cbWidth	cbHeight

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

The variable allowBtSplit is derived as follows:

– If one or more of the following conditions are true, allowBtSplit is set equal to FALSE:
  – cbSize is less than or equal to MinBtSizeY
  – cbWidth is greater than maxBtSize
  – cbHeight is greater than maxBtSize
  – mttDepth is greater than or equal to maxMttDepth
– Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
  – btSplit is equal to SPLIT_BT_VER
  – y0 + cbHeight is greater than pic_height_in_luma_samples
– Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
  – btSplit is equal to SPLIT_BT_HOR
  – x0 + cbWidth is greater than pic_width_in_luma_samples
  – y0 + cbHeight is smaller than or equal to pic_height_in_luma_samples
– Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE:
  – mttDepth is greater than 0
  – partIdx is equal to 1
  – MttSplitMode[x0][y0][mttDepth − 1] is equal to parallelTtSplit
– Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
  – btSplit is equal to SPLIT_BT_VER
  – cbWidth is smaller than or equal to MaxTbSizeY
  – cbHeight is larger than MaxTbSizeY

– Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
  – btSplit is equal to SPLIT_BT_HOR
  – cbWidth is larger than MaxTbSizeY
  – cbHeight is smaller than or equal to MaxTbSizeY
– Otherwise, allowBtSplit is set equal to TRUE.

6.4.2 Allowed ternary split process

Inputs to this process are:

– a ternary split mode ttSplit,
– a coding block width cbWidth,
– a coding block height cbHeight,
– a location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture,
– a multi-type tree depth mttDepth
– a maximum multi-type tree depth with offset maxMttDepth,
– a maximum binary tree size maxTtSize.

Output of this process is the variable allowTtSplit.

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

	ttSplit = = SPLIT_TT_VER	**ttSplit = = SPLIT_TT_HOR**
cbSize	cbWidth	cbHeight

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

The variable allowTtSplit is derived as follows:

– If one or more of the following conditions are true, allowTtSplit is set equal to FALSE:
  – cbSize is less than or equal to 2 * MinTtSizeY
  – cbWidth is greater than Min(MaxTbSizeY, maxTtSize)
  – cbHeight is greater than Min(MaxTbSizeY, maxTtSize)
  – mttDepth is greater than or equal to maxMttDepth
  – x0 + cbWidth is greater than pic_width_in_luma_samples
  – y0 + cbHeight is greater than pic_height_in_luma_samples
– Otherwise, allowTtSplit is set equal to TRUE.

6.5 Scanning processes

6.5.1 CTB raster and scanning process

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

6.5.2 Up-right diagonal scan order array initialization process

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

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

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

7 Syntax and semantics

7.1 Method of specifying syntax in tabular form

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

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

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

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

7.2 Specification of syntax functions and descriptors

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

byte_aligned( ) is specified as follows:

- If the current position in the bitstream is on a byte boundary, i.e., the next bit in the bitstream is the first bit in a byte, the return value of byte_aligned( ) is equal to TRUE.
- Otherwise, the return value of byte_aligned( ) is equal to FALSE.

more_data_in_byte_stream( ), which is used only in the byte stream NAL unit syntax structure specified in Annex TBD, is specified as follows:

- If more data follow in the byte stream, the return value of more_data_in_byte_stream( ) is equal to TRUE.
- Otherwise, the return value of more_data_in_byte_stream( ) is equal to FALSE.

more_data_in_payload( ) is specified as follows:

- If byte_aligned( ) is equal to TRUE and the current position in the sei_payload( ) syntax structure is 8 * payloadSize bits from the beginning of the sei_payload( ) syntax structure, the return value of more_data_in_payload( ) is equal to FALSE.
- Otherwise, the return value of more_data_in_payload( ) is equal to TRUE.

more_rbsp_data( ) is specified as follows:

- If there is no more data in the raw byte sequence payload (RBSP), the return value of more_rbsp_data( ) is equal to FALSE.
- Otherwise, the RBSP data are searched for the last (least significant, right-most) bit equal to 1 that is present in the RBSP. Given the position of this bit, which is the first bit (rbsp_stop_one_bit) of the rbsp_trailing_bits( ) syntax structure, the following applies:
  - If there is more data in an RBSP before the rbsp_trailing_bits( ) syntax structure, the return value of more_rbsp_data( ) is equal to TRUE.
  - Otherwise, the return value of more_rbsp_data( ) is equal to FALSE.

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

more_rbsp_trailing_data( ) is specified as follows:

- If there is more data in an RBSP, the return value of more_rbsp_trailing_data( ) is equal to TRUE.
- Otherwise, the return value of more_rbsp_trailing_data( ) is equal to FALSE.

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

payload_extension_present( ) is specified as follows:

- If the current position in the sei_payload( ) syntax structure is not the position of the last (least significant, right-most) bit that is equal to 1 that is less than 8 * payloadSize bits from the beginning of the syntax structure (i.e., the position of the payload_bit_equal_to_one syntax element), the return value of payload_extension_present( ) is equal to TRUE.
- Otherwise, the return value of payload_extension_present( ) is equal to FALSE.

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

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

- ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in clause TBD.
- b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read_bits(8).

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

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

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

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

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

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

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

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

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

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

7.3 Syntax in tabular form

7.3.1 NAL unit syntax

7.3.1.1 General NAL unit syntax

nal_unit(NumBytesInNalUnit) {	**Descriptor**
nal_unit_header( )	
NumBytesInRbsp = 0	
for(i = 2; i < NumBytesInNalUnit; i++)	
if(i + 2 < NumBytesInNalUnit && next_bits(24) == 0x000003) {	
rbsp_byte[NumBytesInRbsp++]	b(8)
rbsp_byte[NumBytesInRbsp++]	b(8)
i += 2	
emulation_prevention_three_byte /* equal to 0x03 */	f(8)
} else	
rbsp_byte[NumBytesInRbsp++]	b(8)
}	

7.3.1.2 NAL unit header syntax

nal_unit_header( ) {	Descriptor
forbidden_zero_bit	f(1)
nal_unit_type	u(5)
nuh_temporal_id_plus1	u(3)
nuh_reserved_zero_7bits	u(7)
}	

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

7.3.2.1 Sequence parameter set RBSP syntax

seq_parameter_set_rbsp( ) {	Descriptor
sps_seq_parameter_set_id	ue(v)
chroma_format_idc	ue(v)
if(chroma_format_idc == 3)	
separate_colour_plane_flag	u(1)
pic_width_in_luma_samples	ue(v)
pic_height_in_luma_samples	ue(v)
bit_depth_luma_minus8	ue(v)
bit_depth_chroma_minus8	ue(v)
qtbtt_dual_tree_intra_flag	ue(v)
log2_ctu_size_minus2	ue(v)
log2_min_luma_coding_block_size_minus2	ue(v)
partition_constraints_override_enabled_flag	ue(v)
sps_log2_diff_min_qt_min_cb_intra_slice_luma	ue(v)
sps_log2_diff_min_qt_min_cb_inter_slice	ue(v)
sps_max_mtt_hierarchy_depth_inter_slices	ue(v)
sps_max_mtt_hierarchy_depth_intra_slices_luma	ue(v)
if(sps_max_mtt_hierarchy_depth_intra_slices_luma != 0) {	
sps_log2_diff_max_bt_min_qt_intra_slice_luma	ue(v)
sps_log2_diff_max_tt_min_qt_intra_slice_luma	ue(v)
}	
if(sps_max_mtt_hierarchy_depth_inter_slices != 0) {	
sps_log2_diff_max_bt_min_qt_inter_slice	ue(v)
sps_log2_diff_max_tt_min_qt_inter_slice	ue(v)
}	
if(qtbtt_dual_tree_intra_flag) {	
sps_log2_diff_min_qt_min_cb_intra_slice_chroma	ue(v)
sps_max_mtt_hierarchy_depth_intra_slices_chroma	ue(v)
if (sps_max_mtt_hierarchy_depth_intra_slices_chroma != 0) {	
sps_log2_diff_max_bt_min_qt_intra_slice_chroma	ue(v)
sps_log2_diff_max_tt_min_qt_intra_slice_chroma	ue(v)
}	
}	
sps_cclm_enabled_flag	u(1)
sps_sao_enabled_flag	u(1)
sps_alf_enabled_flag	u(1)

pcm_enabled_flag	u(1)
if(pcm_enabled_flag) {	
pcm_sample_bit_depth_luma_minus1	u(4)
pcm_sample_bit_depth_chroma_minus1	u(4)
log2_min_pcm_luma_coding_block_size_minus3	ue(v)
log2_diff_max_min_pcm_luma_coding_block_size	ue(v)
pcm_loop_filter_disabled_flag	u(1)
}	
sps_temporal_mvp_enabled_flag	u(1)
if(sps_temporal_mvp_enabled_flag)	
sps_sbtmvp_enabled_flag	u(1)
if(sps_sbtmvp_enabled_flag)	
log2_sbtmvp_default_size_minus2	u(1)
sps_amvr_enabled_flag	u(1)
sps_affine_enabled_flag	u(1)
if(sps_affine_enabled_flag)	
sps_affine_type_flag	u(1)
sps_mts_intra_enabled_flag	u(1)
sps_mts_inter_enabled_flag	u(1)
sps_ladf_enabled_flag	u(1)
if (sps_ladf_enabled_flag) {	
sps_num_ladf_intervals_minus2	u(2)
sps_ladf_lowest_interval_qp_offset	se(v)
for(i = 0; i < sps_num_ladf_intervals_minus2 + 1; i++) {	
sps_ladf_qp_offset[i]	se(v)
sps_ladf_delta_threshold_minus1[i]	ue(v)
}	
}	
rbsp_trailing_bits( )	
}	

7.3.2.2 Picture parameter set RBSP syntax

pic_parameter_set_rbsp( ) {	**Descriptor**
pps_pic_parameter_set_id	ue(v)
pps_seq_parameter_set_id	ue(v)
init_qp_minus26	se(v)
transform_skip_enabled_flag	u(1)
cu_qp_delta_enabled_flag	u(1)
if(cu_qp_delta_enabled_flag)	
diff_cu_qp_delta_depth	ue(v)
pps_cb_qp_offset	se(v)
pps_cr_qp_offset	se(v)
pps_slice_chroma_qp_offsets_present_flag	u(1)
deblocking_filter_control_present_flag	u(1)
if(deblocking_filter_control_present_flag) {	
deblocking_filter_override_enabled_flag	u(1)

pps_deblocking_filter_disabled_flag	u(1)
if(!pps_deblocking_filter_disabled_flag) {	
pps_beta_offset_div2	se(v)
pps_tc_offset_div2	se(v)
}	
}	
rbsp_trailing_bits( )	
}	

7.3.2.3 End of sequence RBSP syntax

end_of_seq_rbsp( ) {	**Descriptor**
}	

7.3.2.4 End of bitstream RBSP syntax

end_of_bitstream_rbsp( ) {	**Descriptor**
}	

7.3.2.5 Slice layer RBSP syntax

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

7.3.2.6 RBSP slice trailing bits syntax

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

7.3.2.7 RBSP trailing bits syntax

rbsp_trailing_bits( ) {	**Descriptor**
rbsp_stop_one_bit /* equal to 1 */	f(1)
while(!byte_aligned( ))	
rbsp_alignment_zero_bit /* equal to 0 */	f(1)
}	

7.3.2.8 Byte alignment syntax

byte_alignment( ) {	Descriptor
alignment_bit_equal_to_one /* equal to 1 */	f(1)
while(!byte_aligned( ))	
alignment_bit_equal_to_zero /* equal to 0 */	f(1)
}	

7.3.3 Slice header syntax

7.3.3.1 General slice header syntax

slice_header( ) {	Descriptor
slice_pic_parameter_set_id	ue(v)
slice_address	u(v)
slice_type	ue(v)
if(partition_constraints_override_enabled_flag) {	
partition_constraints_override_flag	ue(v)
if(partition_constraints_override_flag) {	
slice_log2_diff_min_qt_min_cb_luma	ue(v)
slice_max_mtt_hierarchy_depth_luma	ue(v)
if(slice_max_mtt_hierarchy_depth_luma != 0)	
slice_log2_diff_max_bt_min_qt_luma	ue(v)
slice_log2_diff_max_tt_min_qt_luma	ue(v)
}	
if(slice_type = = I && qtbtt_dual_tree_intra_flag) {	
slice_log2_diff_min_qt_min_cb_chroma	ue(v)
slice_max_mtt_hierarchy_depth_chroma	ue(v)
if(slice_max_mtt_hierarchy_depth_chroma != 0)	
slice_log2_diff_max_bt_min_qt_chroma	ue(v)
slice_log2_diff_max_tt_min_qt_chroma	ue(v)
}	
}	
}	
}	
if (slice_type != I) {	
if(sps_sbtmvp_enabled_flag) {	
sbtmvp_size_override_flag	u(1)
if(sbtmvp_size_override_flag)	
log2_sbtmvp_active_size_minus2	u(3)
}	
if(sps_temporal_mvp_enabled_flag)	
slice_temporal_mvp_enabled_flag	u(1)
if(slice_type = = B)	
mvd_l1_zero_flag	u(1)
if(slice_temporal_mvp_enabled_flag) {	
if(slice_type = = B)	
collocated_from_l0_flag	u(1)

}	
six_minus_max_num_merge_cand	ue(v)
}	
slice_qp_delta	se(v)
if(pps_slice_chroma_qp_offsets_present_flag) {	
slice_cb_qp_offset	se(v)
slice_cr_qp_offset	se(v)
}	
if(sps_sao_enabled_flag) {	
slice_sao_luma_flag	u(1)
if(ChromaArrayType != 0)	
slice_sao_chroma_flag	u(1)
}	
if(sps_alf_enabled_flag) {	
slice_alf_enabled_flag	u(1)
if(slice_alf_enabled_flag)	
alf_data( )	
}	
dep_quant_enabled_flag	u(1)
if(!dep_quant_enabled_flag)	
sign_data_hiding_enabled_flag	u(1)
if(deblocking_filter_override_enabled_flag)	
deblocking_filter_override_flag	u(1)
if(deblocking_filter_override_flag) {	
slice_deblocking_filter_disabled_flag	u(1)
if(!slice_deblocking_filter_disabled_flag) {	
slice_beta_offset_div2	se(v)
slice_tc_offset_div2	se(v)
}	
}	
byte_alignment( )	
}	

7.3.3.2 Adaptive loop filter data syntax

alf_data( ) {	**Descriptor**
alf_chroma_idc	tu(v)
alf_luma_num_filters_signalled_minus1	tb(v)
if(alf_luma_num_filters_signalled_minus1 > 0) {	
for(filtIdx = 0; filtIdx < NumAlfFilters; filtIdx++)	
alf_luma_coeff_delta_idx[filtIdx]	tb(v)
}	
alf_luma_coeff_delta_flag	u(1)
if (!alf_luma_coeff_delta_flag && alf_luma_num_filters_signalled_minus1 > 0)	
alf_luma_coeff_delta_prediction_flag	u(1)
alf_luma_min_eg_order_minus1	ue(v)
for(i = 0; i < 3; i++)	

alf_luma_eg_order_increase_flag[i]	u(1)
if (alf_luma_coeff_delta_flag) {	
for(sigFiltIdx = 0; sigFiltIdx <= alf_luma_num_filters_signalled_minus1; sigFiltIdx++)	
alf_luma_coeff_flag[sigFiltIdx]	u(1)
}	
for(sigFiltIdx = 0; sigFiltIdx <= alf_luma_num_filters_signalled_minus1; sigFiltIdx++) {	
if (alf_luma_coeff_flag[sigFiltIdx]) {	
for (j = 0; j < 12; j++) {	
alf_luma_coeff_delta_abs[sigFiltIdx][j]	uek(v)
if(alf_luma_coeff_delta_abs[sigFiltIdx][j])	
alf_luma_coeff_delta_sign[sigFiltIdx][j]	u(1)
}	
}	
}	
if (alf_chroma_idc > 0) {	
alf_chroma_min_eg_order_minus1	ue(v)
for(i = 0; i < 2; i++)	
alf_chroma_eg_order_increase_flag[i]	u(1)
for(j = 0; j < 6; j++) {	
alf_chroma_coeff_abs[j]	uek(v)
if(alf_chroma_coeff_abs[j] > 0)	
alf_chroma_coeff_sign[j]	u(1)
}	
}	
}	

7.3.4 Slice data syntax

7.3.4.1 General slice data syntax

slice_data( ) {	**Descriptor**
do {	
coding_tree_unit( )	
end_of_slice_flag	ae(v)
CtbAddrInRs++	
} while(!end_of_slice_flag)	
}	

7.3.4.2 Coding tree unit syntax

coding_tree_unit( ) {	Descriptor
xCtb = (CtbAddrInRs % PicWidthInCtbsY) << CtbLog2SizeY	
yCtb = (CtbAddrInRs / PicWidthInCtbsY) << CtbLog2SizeY	
if(slice_sao_luma_flag || slice_sao_chroma_flag)	
sao(xCtb >> CtbLog2SizeY, yCtb >> CtbLog2SizeY)	
if(slice_alf_enabled_flag){	
alf_ctb_flag[0][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)
if(alf_chroma_idc == 1 || alf_chroma_idc == 3)	
alf_ctb_flag[1][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)
if(alf_chroma_idc == 2 || alf_chroma_idc == 3)	
alf_ctb_flag[2][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)
}	
if(slice_type = = I && qtbtt_dual_tree_intra_flag)	
dual_tree_implicit_qt_split (xCtb, yCtb, CtbLog2SizeY, 0)	
else	
coding_quadtree(xCtb, yCtb, CtbLog2SizeY, 0, SINGLE_TREE)	
}	

dual_tree_implicit_qt_split(x0, y0, log2CbSize, cqtDepth) {	Descriptor
if(log2CbSize > 6) {	
x1 = x0 + (1 << (log2CbSize − 1))	
y1 = y0 + (1 << (log2CbSize − 1))	
dual_tree_implicit_qt_split(x0, y0, log2CbSize − 1, cqtDepth + 1)	
if(x1 < pic_width_in_luma_samples)	
dual_tree_implicit_qt_split(x1, y0, log2CbSize − 1, cqtDepth + 1)	
if(y1 < pic_height_in_luma_samples)	
dual_tree_implicit_qt_split(x0, y1, log2CbSize − 1, cqtDepth + 1)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
dual_tree_implicit_qt_split(x1, y1, log2CbSize − 1, cqtDepth + 1)	
} else {	
coding_quadtree(x0, y0, log2CbSize, cqtDepth, DUAL_TREE_LUMA)	
coding_quadtree(x0, y0, log2CbSize, cqtDepth, DUAL_TREE_CHROMA)	
}	
}	

7.3.4.3 Sample adaptive offset syntax

sao(rx, ry) {	Descriptor
if(rx > 0) {	
leftCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − 1]]	
if(leftCtbInTile)	
sao_merge_left_flag	ae(v)
}	
if(ry > 0 && !sao_merge_left_flag) {	
upCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − PicWidthInCtbsY]]	
if(upCtbInTile)	
sao_merge_up_flag	ae(v)
}	
if(!sao_merge_up_flag && !sao_merge_left_flag)	
for(cIdx = 0; cIdx < (ChromaArrayType != 0 ? 3 : 1); cIdx++)	
if((slice_sao_luma_flag && cIdx == 0) || (slice_sao_chroma_flag && cIdx > 0)) {	
if(cIdx == 0)	
sao_type_idx_luma	ae(v)
else if(cIdx == 1)	
sao_type_idx_chroma	ae(v)
if(SaoTypeIdx[cIdx][rx][ry] != 0) {	
for(i = 0; i < 4; i++)	
sao_offset_abs[cIdx][rx][ry][i]	ae(v)
if(SaoTypeIdx[cIdx][rx][ry] == 1) {	
for(i = 0; i < 4; i++)	
if(sao_offset_abs[cIdx][rx][ry][i] != 0)	
sao_offset_sign[cIdx][rx][ry][i]	ae(v)
sao_band_position[cIdx][rx][ry]	ae(v)
} else {	
if(cIdx == 0)	
sao_eo_class_luma	ae(v)
if(cIdx == 1)	
sao_eo_class_chroma	ae(v)
}	
}	
}	
}	

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

7.3.4.4 Coding quadtree syntax

coding_quadtree(x0, y0, log2CbSize, cqtDepth, treeType) {	**Descriptor**
minQtSize = (treeType = = DUAL_TREE_CHROMA) ? MinQtSizeC : MinQtSizeY	
maxBtSize = (treeType = = DUAL_TREE_CHROMA) ? MaxBtSizeC : MaxBtSizeY	
if((((x0 + (1 << log2CbSize) <= pic_width_in_luma_samples) ? 1 : 0) + ((y0 + (1 << log2CbSize) <= pic_height_in_luma_samples) ? 1 : 0) + (((1 << log2CbSize) <= maxBtSize) ? 1 : 0)) >= 2 && (1 << log2CbSize) > minQtSize)	
qt_split_cu_flag[x0][y0]	ae(v)
if(cu_qp_delta_enabled_flag && cqtDepth <= diff_cu_qp_delta_depth) {	
IsCuQpDeltaCoded = 0	
CuQpDeltaVal = 0	
CuQgTopLeftX = x0	
CuQgTopLeftY = y0	
}	
if(qt_split_cu_flag[x0][y0]) {	
x1 = x0 + (1 << (log2CbSize − 1))	
y1 = y0 + (1 << (log2CbSize − 1))	
coding_quadtree(x0, y0, log2CbSize − 1, cqtDepth + 1, treeType)	
if(x1 < pic_width_in_luma_samples)	
coding_quadtree(x1, y0, log2CbSize − 1, cqtDepth + 1, treeType)	
if(y1 < pic_height_in_luma_samples)	
coding_quadtree(x0, y1, log2CbSize − 1, cqtDepth + 1, treeType)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
coding_quadtree(x1, y1, log2CbSize − 1, cqtDepth + 1, treeType)	
} else	
multi_type_tree(x0, y0, 1 << log2CbSize, 1 << log2CbSize, cqtDepth, 0, 0, 0, treeType)	
}	

7.3.4.5 Multi-type tree syntax

multi_type_tree(x0, y0, cbWidth, cbHeight, cqtDepth, mttDepth, depthOffset, partIdx, treeType) {	**Descriptor**
if((allowSplitBtVer || allowSplitBtHor || allowSplitTtVer || allowSplitTtHor) && (x0 + cbWidth <= pic_width_in_luma_samples) && (y0 + cbHeight <= pic_height_in_luma_samples))	
mtt_split_cu_flag	ae(v)
if(cu_qp_delta_enabled_flag && (cqtDepth + mttDepth) <= diff_cu_qp_delta_depth) {	
IsCuQpDeltaCoded = 0	
CuQpDeltaVal = 0	
CuQgTopLeftX = x0	
CuQgTopLeftY = y0	
}	
if(mtt_split_cu_flag) {	
if((allowSplitBtHor || allowSplitTtHor) && (allowSplitBtVer || allowSplitTtVer))	
mtt_split_cu_vertical_flag	ae(v)

if((allowSplitBtVer && allowSplitTtVer && mtt_split_cu_vertical_flag) || (allowSplitBtHor && allowSplitTtHor && !mtt_split_cu_vertical_flag))	
mtt_split_cu_binary_flag	ae(v)
if(MttSplitMode[x0][y0][mttDepth] == SPLIT_BT_VER) {	
depthOffset += (x0 + cbWidth > pic_width_in_luma_samples) ? 1 : 0	
x1 = x0 + (cbWidth / 2)	
multi_type_tree(x0, y0, cbWidth / 2, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)	
if(x1 < pic_width_in_luma_samples)	
multi_type_tree(x1, y0, cbWidth / 2, cbHeightY, cqtDepth, mttDepth + 1, depthOffset, 1, treeType)	
} else if(MttSplitMode[x0][y0][mttDepth] == SPLIT_BT_HOR) {	
depthOffset += (y0 + cbHeight > pic_height_in_luma_samples) ? 1 : 0	
y1 = y0 + (cbHeight / 2)	
multi_type_tree(x0, y0, cbWidth, cbHeight / 2, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)	
if(y1 < pic_height_in_luma_samples)	
multi_type_tree(x0, y1, cbWidth, cbHeight / 2, cqtDepth, mttDepth + 1, depthOffset, 1, treeType)	
} else if(MttSplitMode[x0][y0][mttDepth] == SPLIT_TT_VER) {	
x1 = x0 + (cbWidth / 4)	
x2 = x0 + (3 * cbWidth / 4)	
multi_type_tree(x0, y0, cbWidth / 4, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)	
multi_type_tree(x1, y0, cbWidth / 2, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 1, treeType)	
multi_type_tree(x2, y0, cbWidth / 4, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 2, treeType)	
} else { /* SPLIT_TT_HOR */	
y1 = y0 + (cbHeight / 4)	
y2 = y0 + (3 * cbHeight / 4)	
multi_type_tree(x0, y0, cbWidth, cbHeight / 4, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)	
multi_type_tree(x0, y1, cbWidth, cbHeight / 2, cqtDepth, mttDepth + 1, depthOffset, 1, treeType)	
multi_type_tree(x0, y2, cbWidth, cbHeight / 4, cqtDepth, mttDepth + 1, depthOffset, 2 , treeType)	
}	
} else	
coding_unit(x0, y0, cbWidth, cbHeight, treeType)	
}	

7.3.4.6 Coding unit syntax

coding_unit(x0, y0, cbWidth, cbHeight, treeType) {	**Descriptor**
if(slice_type != I) {	
cu_skip_flag[x0][y0]	ae(v)
if(cu_skip_flag[x0][y0] = = 0)	
pred_mode_flag	ae(v)
}	

if(CuPredMode[x0][y0] = = MODE_INTRA) {	
if(pcm_enabled_flag && cbWidth >= MinIpcmCbSizeY && cbWidth <= MaxIpcmCbSizeY && cbHeight >= MinIpcmCbSizeY && cbHeight <= MaxIpcmCbSizeY)	
pcm_flag[x0][y0]	
if(pcm_flag[x0][y0]) {	
while(!byte_aligned( ))	
pcm_alignment_zero_bit	
pcm_sample(cbWidth, cbHeight, treeType)	
} else {	
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA) {	
if((y0 % CtbSizeY) > 0)	
intra_luma_ref_idx[x0][y0]	ae(v)
if (intra_luma_ref_idx[x0][y0] = = 0)	
intra_luma_mpm_flag[x0][y0]	ae(v)
if(intra_luma_mpm_flag[x0][y0])	
intra_luma_mpm_idx[x0][y0]	ae(v)
else	
intra_luma_mpm_remainder[x0][y0]	ae(v)
}	
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA)	
intra_chroma_pred_mode[x0][y0]	ae(v)
}	
} else { /* MODE_INTER */	
if(cu_skip_flag[x0][y0]) {	
if(sps_affine_enabled_flag && cbWidth >= 8 && cbHeight >= 8 && (MotionModelIdc[x0 − 1][y0 + cbHeight − 1] != 0 || MotionModelIdc[x0 − 1][y0 + cbHeight] != 0 || MotionModelIdc[x0 − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth][y0 − 1]] != 0))	
merge_affine_flag[x0][y0]	ae(v)
if(merge_affine_flag[x0][y0] = = 0 && MaxNumMergeCand > 1)	
merge_idx[x0][y0]	ae(v)
} else {	
merge_flag[x0][y0]	ae(v)
if(merge_flag[x0][y0]) {	
if(sps_affine_enabled_flag && cbWidth >= 8 && cbHeight >= 8 && (MotionModelIdc[x0 − 1][y0 + cbHeight − 1] != 0 || MotionModelIdc[x0 − 1][y0 + cbHeight] != 0 || MotionModelIdc[x0 − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth − 1][y0 − 1] != 0 || MotionModelIdc[x0 + cbWidth][y0 − 1]] != 0))	
merge_affine_flag[x0][y0]	ae(v)
if(merge_affine_flag[x0][y0] = = 0 && MaxNumMergeCand > 1)	
merge_idx[x0][y0]	ae(v)
} else {	
if(slice_type = = B)	
inter_pred_idc[x0][y0]	ae(v)
if(sps_affine_enabled_flag && cbWidth >= 16 && cbHeight >= 16) {	
inter_affine_flag[x0][y0]	ae(v)

if(sps_affine_type_flag && inter_affine_flag[x0][y0])	
cu_affine_type_flag[x0][y0]	ae(v)
}	
if(inter_pred_idc[x0][y0] != PRED_L1) {	
if(num_ref_idx_l0_active_minus1 > 0)	
ref_idx_l0[x0][y0]	ae(v)
mvd_coding(x0, y0, 0, 0)	
if(MotionModelIdc[x0][y0] > 0)	
mvd_coding(x0, y0, 0, 1)	
if(MotionModelIdc[x0][y0] > 1)	
mvd_coding(x0, y0, 0, 2)	
mvp_l0_flag[x0][y0]	ae(v)
} else {	
MvdL0[x0][y0][0] = 0	
MvdL0[x0][y0][1] = 0	
}	
if(inter_pred_idc[x0][y0] != PRED_L0) {	
if(num_ref_idx_l1_active_minus1 > 0)	
ref_idx_l1[x0][y0]	ae(v)
if(mvd_l1_zero_flag && inter_pred_idc[x0][y0] == PRED_BI) {	
MvdL1[x0][y0][0] = 0	
MvdL1[x0][y0][1] = 0	
MvdCpL1[x0][y0][0][0] = 0	
MvdCpL1[x0][y0][0][1] = 0	
MvdCpL1[x0][y0][1][0] = 0	
MvdCpL1[x0][y0][1][1] = 0	
MvdCpL1[x0][y0][2][0] = 0	
MvdCpL1[x0][y0][2][1] = 0	
} else {	
mvd_coding(x0, y0, 1, 0)	
if(MotionModelIdc[x0][y0] > 0)	
mvd_coding(x0, y0, 1, 1)	
if(MotionModelIdc[x0][y0] > 1)	
mvd_coding(x0, y0, 1, 2)	
mvp_l1_flag[x0][y0]	ae(v)
} else {	
MvdL1[x0][y0][0] = 0	
MvdL1[x0][y0][1] = 0	
}	
if(sps_amvr_enabled_flag && inter_affine_flag == 0 && (MvdL0[x0][y0][0] != 0 || MvdL0[x0][y0][1] != 0 || MvdL1[x0][y0][0] != 0 || MvdL1[x0][y0][1] != 0))	
amvr_mode[x0][y0]	ae(v)
}	
}	
}	
if(!pcm_flag[x0][y0]) {	
if(CuPredMode[x0][y0] != MODE_INTRA && cu_skip_flag[x0][y0] == 0)	
cu_cbf	ae(v)

if(cu_cbf)	
transform_tree(x0, y0, cbWidth, cbHeight, treeType)	
}	
}	

7.3.4.7 PCM sample syntax

pcm_sample(cbWidth, cbHeight, treeType) {	**Descriptor**
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA) {	
for(i = 0; i < cbWidth * cbHeight; i++)	
pcm_sample_luma[i]	u(v)
}	
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA) {	
for(i = 0; i < 2 * ((cbWidth * cbHeight) / (SubWidthC * SubHeightC)); i++)	
pcm_sample_chroma[i]	u(v)
}	
}	

7.3.4.8 Motion vector difference syntax

mvd_coding(x0, y0, refList ,cpIdx) {	**Descriptor**
abs_mvd_greater0_flag[0]	ae(v)
abs_mvd_greater0_flag[1]	ae(v)
if(abs_mvd_greater0_flag[0])	
abs_mvd_greater1_flag[0]	ae(v)
if(abs_mvd_greater0_flag[1])	
abs_mvd_greater1_flag[1]	ae(v)
if(abs_mvd_greater0_flag[0]) {	
if(abs_mvd_greater1_flag[0])	
abs_mvd_minus2[0]	ae(v)
mvd_sign_flag[0]	ae(v)
}	
if(abs_mvd_greater0_flag[1]) {	
if(abs_mvd_greater1_flag[1])	
abs_mvd_minus2[1]	ae(v)
mvd_sign_flag[1]	ae(v)
}	
}	

7.3.4.9 Transform tree syntax

transform_tree(x0, y0, tbWidth, tbHeight , treeType) {	Descriptor
if(tbWidth > MaxTbSizeY || tbHeight > MaxTbSizeY) {	
trafoWidth = (tbWidth > MaxTbSizeY) ? (tbWidth / 2) : tbWidth	
trafoHeight = (tbHeight > MaxTbSizeY) ? (tbHeight / 2) : tbHeight	
transform_tree(x0, y0, trafoWidth, trafoHeight)	
if(tbWidth > MaxTbSizeY)	
transform_tree(x0 + trafoWidth, y0, trafoWidth, trafoHeight, treeType)	
if(tbHeight > MaxTbSizeY)	
transform_tree(x0, y0 + trafoHeight, trafoWidth, trafoHeight, treeType)	
if(tbWidth > MaxTbSizeY && tbHeight > MaxTbSizeY)	
transform_tree(x0 + trafoWidth, y0 + trafoHeight, trafoWidth, trafoHeight, treeType)	
} else {	
transform_unit(x0, y0, tbWidth, tbHeight, treeType)	
}	
}	

7.3.4.10 Transform unit syntax

transform_unit(x0, y0, tbWidth, tbHeight, treeType) {	Descriptor
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA)	
tu_cbf_luma[x0][y0]	ae(v)
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA) {	
tu_cbf_cb[x0][y0]	ae(v)
tu_cbf_cr[x0][y0]	ae(v)
}	
if((tu_cbf_luma[x0][y0] || tu_cbf_cb[x0][y0] || tu_cbf_cr[x0][y0]) && treeType != DUAL_TREE_CHROMA) {	
if(cu_qp_delta_enabled_flag && !IsCuQpDeltaCoded) {	
cu_qp_delta_abs	ae(v)
if(cu_qp_delta_abs)	
cu_qp_delta_sign_flag	ae(v)
}	
}	
if((((CuPredMode[x0][y0] = = MODE_INTRA) && sps_mts_intra_enabled_flag) || ((CuPredMode[x0][y0] = = MODE_INTER) && sps_mts_inter_enabled_flag)) && tu_cbf_luma[x0][y0] && treeType != DUAL_TREE_CHROMA && (tbWidth <= 32) && (tbHeight <= 32))	
tu_mts_flag[x0][y0]	ae(v)
if(tu_cbf_luma[x0][y0])	
residual_coding(x0, y0, log2(tbWidth), log2(tbHeight), 0)	
if(tu_cbf_cb[x0][y0])	
residual_coding(x0, y0, log2(tbWidth / 2), log2(tbHeight / 2), 1)	
if(tu_cbf_cr[x0][y0])	
residual_coding(x0, y0, log2(tbWidth / 2), log2(tbHeight / 2), 2)	
}	

7.3.4.11 Residual coding syntax

residual_coding(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	Descriptor
if(transform_skip_enabled_flag && (cIdx != 0 || tu_mts_flag[x0][y0] == 0) && (log2TbWidth <= 2) && (log2TbHeight <= 2))	
transform_skip_flag[x0][y0][cIdx]	ae(v)
last_sig_coeff_x_prefix	ae(v)
last_sig_coeff_y_prefix	ae(v)
if(last_sig_coeff_x_prefix > 3)	
last_sig_coeff_x_suffix	ae(v)
if(last_sig_coeff_y_prefix > 3)	
last_sig_coeff_y_suffix	ae(v)
log2SbSize = (Min(log2TbWidth, log2TbHeight) < 2 ? 1 : 2)	
numSbCoeff = 1 << (log2SbSize << 1)	
lastScanPos = numSbCoeff	
lastSubBlock = (1 << (log2TbWidth + log2TbHeight − 2 * log2SbSize)) − 1	
do {	
if(lastScanPos == 0) {	
lastScanPos = numSbCoeff	
lastSubBlock− −	
}	
lastScanPos− −	
xS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize] [lastSubBlock][0]	
yS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize] [lastSubBlock][1]	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][lastScanPos][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][lastScanPos][1]	
} while((xC != LastSignificantCoeffX) || (yC != LastSignificantCoeffY))	
numSigCoeff = 0	
QState = 0	
for(i = lastSubBlock; i >= 0; i− −) {	
startQStateSb = QState	
xS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize] [lastSubBlock][0]	
yS = DiagScanOrder[log2TbWidth − log2SbSize][log2TbHeight − log2SbSize] [lastSubBlock][1]	
inferSbDcSigCoeffFlag = 0	
if((i < lastSubBlock) && (i > 0)) {	
coded_sub_block_flag[xS][yS]	ae(v)
inferSbDcSigCoeffFlag = 1	
}	
firstSigScanPosSb = numSbCoeff	
lastSigScanPosSb = −1	
for(n = (i == lastSubBlock) ? lastScanPos − 1 : numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(coded_sub_block_flag[xS][yS] && (n > 0 || !inferSbDcSigCoeffFlag)) {	

sig_coeff_flag[xC][yC]	ae(v)
if(sig_coeff_flag[xC][yC])	
inferSbDcSigCoeffFlag = 0	
}	
if(sig_coeff_flag[xC][yC]) {	
numSigCoeff++	
par_level_flag[n]	ae(v)
rem_abs_gt1_flag[n]	ae(v)
if(lastSigScanPosSb == −1)	
lastSigScanPosSb = n	
firstSigScanPosSb = n	
}	
AbsLevelPass1[xC][yC] = sig_coeff_flag[xC][yC] + par_level_flag[n] + 2 * rem_abs_gt1_flag[n]	
if(dep_quant_enabled_flag)	
QState = QStateTransTable[QState][par_level_flag[n]]	
}	
for(n = numSbCoeff − 1; n >= 0; n− −)	
if(rem_abs_gt1_flag[n])	
rem_abs_gt2_flag[n]	ae(v)
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(rem_abs_gt2_flag[n])	
abs_remainder[n]	
AbsLevel[xC][yC] = AbsLevelPass1[xC][yC] + 2 * (rem_abs_gt2_flag[n] + abs_remainder[n])	
}	
if(dep_quant_enabled_flag || !sign_data_hiding_enabled_flag)	
signHidden = 0	
else	
signHidden = (lastSigScanPosSb − firstSigScanPosSb > 3 ? 1 : 0)	
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(sig_coeff_flag[xC][yC] && (!signHidden || (n != firstSigScanPosSb)))	
coeff_sign_flag[n]	ae(v)
}	
if(dep_quant_enabled_flag) {	
QState = startQStateSb	
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(sig_coeff_flag[xC][yC])	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = (2 * AbsLevel[xC][yC] − (QState > 1 ? 1 : 0)) * (1 − 2 * coeff_sign_flag[n])	

QState = QStateTransTable[QState][par_level_flag[n]]	
} else {	
sumAbsLevel = 0	
for(n = numSbCoeff − 1; n >= 0; n− −) {	
xC = (xS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][0]	
yC = (yS << log2SbSize) + DiagScanOrder[log2SbSize][log2SbSize][n][1]	
if(sig_coeff_flag[xC][yC]) {	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = AbsLevel[xC][yC] * (1 − 2 * coeff_sign_flag[n])	
if(signHidden) {	
sumAbsLevel += AbsLevel[xC][yC]	
if((n == firstSigScanPosSb) && (sumAbsLevel % 2) == 1))	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = −TransCoeffLevel[x0][y0][cIdx][xC][yC]	
}	
}	
}	
}	
}	
if(tu_mts_flag[x0][y0] && (cIdx == 0))	
mts_idx[x0][y0][cIdx]	ae(v)
}	

7.4 Semantics

7.4.1 General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this clause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this Specification.

7.4.2 NAL unit semantics

7.4.2.1 General NAL unit semantics

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

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

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

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

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

– Otherwise, the RBSP contains the SODB as follows:

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

2) rbsp_trailing_bits( ) are present after the SODB as follows:

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

ii) The next bit consists of a single rbsp_stop_one_bit equal to 1.

iii) When the rbsp_stop_one_bit is not the last bit of a byte-aligned byte, one or more rbsp_alignment_zero_bit is present to result in byte alignment.

3) One or more cabac_zero_word 16-bit syntax elements equal to 0x0000 may be present in some RBSPs after the rbsp_trailing_bits( ) at the end of the RBSP.

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

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

emulation_prevention_three_byte is a byte equal to 0x03. When an emulation_prevention_three_byte is present in the NAL unit, it shall be discarded by the decoding process.

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

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

– 0x000000

– 0x000001

– 0x000002

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

– 0x00000300

– 0x00000301

– 0x00000302

– 0x00000303

7.4.2.2 NAL unit header semantics

forbidden_zero_bit shall be equal to 0.

nal_unit_type specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7-1.

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

nal_unit_type	Name of nal_unit_type	Content of NAL unit and RBSP syntax structure	NAL unit type class
0	NON_IRAP_NUT	Coded slice segment of a non-IRAP picture slice_layer_rbsp( )	VCL
1	IRAP_NUT	Coded slice of an IRAP picture slice_layer_rbsp( )	VCL
2-15	RSV_VCL_NUT	Reserved VCL NAL Units	VCL
16	SPS_NUT	Sequence parameter set seq_parameter_set_rbsp( )	non-VCL
17	PPS_NUT	Picture parameter set pic_parameter_set_rbsp( )	non-VCL
18	EOS_NUT	End of sequence end_of_seq_rbsp( )	non-VCL
19	EOB_NUT	End of bitstream end_of_bitstream_rbsp( )	non-VCL
20, 21	PREFIX_SEI_NUT SUFFIX_SEI_NUT	Supplemental enhancement information sei_rbsp( )	non-VCL

22-26	RSV_NVCL	Reserved	non-VCL
27-31	UNSPEC	Unspecified	non-VCL

nuh_temporal_id_plus1 minus 1 specifies a temporal identifier for the NAL unit. The value of nuh_temporal_id_plus1 shall not be equal to 0.

The variable TemporalId is derived as follows:

$$TemporalId = nuh\_temporal\_id\_plus1 - 1 \quad (7\text{-}1)$$

When nal_unit_type is equal to IRAP_NUT, the coded slice belongs to an IRAP picture, TemporalId shall be equal to 0.

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

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

– If nal_unit_type is equal to SPS_NUT, TemporalId shall be equal to 0 and the TemporalId of the access unit containing the NAL unit shall be equal to 0.

– Otherwise, if nal_unit_type is equal to EOS_NUT or EOB_NUT, TemporalId shall be equal to 0.

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

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

nuh_reserved_zero_7bits shall be equal to '0000000'. Other values of nuh_reserved_zero_7bits may be specified in the future by ITU-T | ISO/IEC. Decoders shall ignore (i.e. remove from the bitstream and discard) NAL units with values of nuh_reserved_zero_7bits not equal to '0000000'.

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

7.4.3.1 Sequence parameter set RBSP semantics

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

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

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

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

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

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

– Otherwise (separate_colour_plane_flag is equal to 1), ChromaArrayType is set equal to 0.

pic_width_in_luma_samples specifies the width of each decoded picture in units of luma samples. pic_width_in_luma_samples shall not be equal to 0 and shall be an integer multiple of MinCbSizeY.

pic_height_in_luma_samples specifies the height of each decoded picture in units of luma samples. pic_height_in_luma_samples shall not be equal to 0 and shall be an integer multiple of MinCbSizeY.

bit_depth_luma_minus8 specifies the bit depth of the samples of the luma array $BitDepth_Y$ and the value of the luma quantization parameter range offset $QpBdOffset_Y$ as follows:

$$BitDepth_Y = 8 + bit\_depth\_luma\_minus8 \quad (7\text{-}2)$$

$$QpBdOffset_Y = 6 * bit\_depth\_luma\_minus8 \quad (7\text{-}3)$$

bit_depth_luma_minus8 shall be in the range of 0 to 8, inclusive.

bit_depth_chroma_minus8 specifies the bit depth of the samples of the chroma arrays $BitDepth_C$ and the value of the chroma quantization parameter range offset $QpBdOffset_C$ as follows:

$$BitDepth_C = 8 + bit\_depth\_chroma\_minus8 \quad (7\text{-}4)$$

$$QpBdOffset_C = 6 * bit\_depth\_chroma\_minus8 \quad (7\text{-}5)$$

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

qtbtt_dual_tree_intra_flag equal to 1 specifies that for I slices, each CTU is split into coding units with 64x64 luma samples using an implicit quadtree split and that these coding units are the root of two separate coding_quadtree syntax structure for luma and chroma.

log2_ctu_size_minus2 plus 2 specifies the luma coding tree block size of each CTU.

log2_min_luma_coding_block_size_minus2 plus 2 specifies the minimum luma coding block size.

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

$$CtbLog2SizeY = log2\_ctu\_size\_minus2 + 2 \quad (7\text{-}6)$$

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

$$MinCbLog2SizeY = log2\_min\_luma\_coding\_block\_size\_minus2 + 2 \quad (7\text{-}8)$$

$$MinCbSizeY = 1 << MinCbLog2SizeY \quad (7\text{-}9)$$

$$MinTbLog2SizeY = 2 \quad (7\text{-}10)$$

$$MaxTbLog2SizeY = 6 \quad (7\text{-}11)$$

$$MinTbSizeY = 1 << MinTbLog2SizeY \quad (7\text{-}12)$$

$$MaxTbSizeY = 1 << MaxTbLog2SizeY \quad (7\text{-}13)$$

$$PicWidthInCtbsY = Ceil(pic\_width\_in\_luma\_samples \div CtbSizeY) \quad (7\text{-}14)$$

$$PicHeightInCtbsY = Ceil(pic\_height\_in\_luma\_samples \div CtbSizeY) \quad (7\text{-}15)$$

$$PicSizeInCtbsY = PicWidthInCtbsY * PicHeightInCtbsY \quad (7\text{-}16)$$

$$PicWidthInMinCbsY = pic\_width\_in\_luma\_samples / MinCbSizeY \quad (7\text{-}17)$$

$$PicHeightInMinCbsY = pic\_height\_in\_luma\_samples / MinCbSizeY \quad (7\text{-}18)$$

$$PicSizeInMinCbsY = PicWidthInMinCbsY * PicHeightInMinCbsY \quad (7\text{-}19)$$

$$PicSizeInSamplesY = pic\_width\_in\_luma\_samples * pic\_height\_in\_luma\_samples \quad (7\text{-}20)$$

$$PicWidthInSamplesC = pic\_width\_in\_luma\_samples / SubWidthC \quad (7\text{-}21)$$

$$PicHeightInSamplesC = pic\_height\_in\_luma\_samples / SubHeightC \quad (7\text{-}22)$$

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

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

– If chroma_format_idc is equal to 0 (monochrome) or separate_colour_plane_flag is equal to 1, CtbWidthC and CtbHeightC are both equal to 0.

– Otherwise, CtbWidthC and CtbHeightC are derived as follows:

$$CtbWidthC = CtbSizeY / SubWidthC \quad (7\text{-}23)$$

$$CtbHeightC = CtbSizeY / SubHeightC \quad (7\text{-}24)$$

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

partition_constraints_override_enabled_flag equal to 1 specifies the presence of partition_constraints_override_flag in the slice headers for slices referring to the SPS. partition_constraints_override_enabled_flag equal to 0 specifies the absence of partition_constraints_override_ flag in the slice headers for slices referring to the SPS.

sps_log2_diff_min_qt_min_cb_intra_slice_luma specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum coding block size in luma samples for luma CUs in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_min_qt_min_cb_luma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_min_qt_min_cb_intra_slice_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. The base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU is derived as follows:

$$MinQtLog2SizeIntraY = sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_luma + MinCbLog2SizeY \quad (7\text{-}25)$$

sps_log2_diff_min_qt_min_cb_inter_slice specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum luma coding block size in luma samples for luma CUs in slices with slice_type equal to 0 (B) or 1 (P) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_min_qt_min_cb_luma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_min_qt_min_cb_inter_slice shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. The base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU is derived as follows:

$$MinQtLog2SizeInterY = sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_slice + MinCbLog2SizeY \quad (7\text{-}26)$$

sps_max_mtt_hierarchy_depth_inter_slices specifies the default maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice_type equal to 0 (B) or 1 (P) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default maximum hierarchy depth can be overridden by slice_max_mtt_hierarchy_depth_luma present in the slice header of the slices referring to the SPS. The value of sps_max_mtt_hierarchy_depth_inter_slices shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

sps_max_mtt_hierarchy_depth_intra_slices_luma specifies the default maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default maximum hierarchy depth can be overridden by slice_max_mtt_hierarchy_depth_luma present in the slice header of the slices referring to the SPS. The value of sps_max_mtt_hierarchy_depth_intra_slices_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

sps_log2_diff_max_bt_min_qt_intra_slice_luma specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_max_bt_min_qt_luma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_max_bt_min_qt_intra_slice_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraY, inclusive. When sps_log2_diff_max_bt_min_qt_intra_slice_luma is not present, the value of sps_log2_diff_max_bt_min_qt_intra_slice_luma is inferred to be equal to 0.

sps_log2_diff_max_tt_min_qt_intra_slice_luma specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_max_tt_min_qt_luma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_max_tt_min_qt_intra_slice_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraY, inclusive. When sps_log2_diff_max_tt_min_qt_intra_slice_luma is not present, the value of sps_log2_diff_max_tt_min_qt_intra_slice_luma is inferred to be equal to 0.

sps_log2_diff_max_bt_min_qt_inter_slice specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice_type equal to 0 (B) or 1 (P) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_max_bt_min_qt_luma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_max_bt_min_qt_inter_slice shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeInterY, inclusive. When sps_log2_diff_max_bt_min_qt_inter_slice is not present, the value of sps_log2_diff_max_bt_min_qt_inter_slice is inferred to be equal to 0.

sps_log2_diff_max_tt_min_qt_inter_slice specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice_type equal to 0 (B) or 1 (P) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_max_tt_min_qt_luma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_max_tt_min_qt_inter_slice shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeInterY, inclusive. When sps_log2_diff_max_tt_min_qt_inter_slice is not present, the value of sps_log2_diff_max_tt_min_qt_inter_slice is inferred to be equal to 0.

sps_log2_diff_min_qt_min_cb_intra_slice_chroma specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL_TREE_CHROMA and the base 2 logarithm of the minimum coding block size in luma samples for chroma CUs with treeType equal to DUAL_TREE_CHROMA in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default difference can be overridden by slice_log2_diff_min_qt_min_cb_chroma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_min_qt_min_cb_intra_slice_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of sps_log2_diff_min_qt_min_cb_intra_slice_chroma is inferred to be equal to 0. The base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a CTU with treeType equal to DUAL_TREE_CHROMA is derived as follows:

$$MinQtLog2SizeIntraC = sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma + MinCbLog2SizeY \quad (7\text{-}27)$$

sps_max_mtt_hierarchy_depth_intra_slices_chroma specifies the default maximum hierarchy depth for chroma coding units resulting from multi-type tree splitting of a chroma quadtree leaf with treeType equal to DUAL_TREE_CHROMA in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_ flag is equal to 1, the default maximum hierarchy depth can be overridden by slice_max_mtt_hierarchy_depth_chroma present in the slice header of the slices referring to the SPS. The value of sps_max_mtt_hierarchy_depth_intra_slices_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of sps_max_mtt_hierarchy_depth_intra_slices_chroma is inferred to be equal to 0.

sps_log2_diff_max_bt_min_qt_intra_slice_chroma specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL_TREE_CHROMA in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_flag is equal to 1, the default difference can be overridden by slice_log2_diff_max_bt_min_qt_chroma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_max_bt_min_qt_intra_slice_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraC, inclusive. When sps_log2_diff_max_bt_min_qt_intra_slice_chroma is not present, the value of sps_log2_diff_max_bt_min_qt_intra_slice_chroma is inferred to be equal to 0.

sps_log2_diff_max_tt_min_qt_intra_slice_chroma specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL_TREE_CHROMA in slices with slice_type equal to 2 (I) referring to the SPS. When partition_constraints_override_flag is equal to 1, the default difference can be overridden by slice_log2_diff_max_tt_min_qt_chroma present in the slice header of the slices referring to the SPS. The value of sps_log2_diff_max_tt_min_qt_intra_slice_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraC,

inclusive. When sps_log2_diff_max_tt_min_qt_intra_slice_chroma is not present, the value of sps_log2_diff_max_tt_min_qt_intra_slice_chroma is inferred to be equal to 0.

sps_cclm_enabled_flag equal to 0 specifies that the cross-component linear model intra prediction from luma component to chroma component is disabled. sps_cclm_enabled_flag equal to 1 specifies that the cross-component linear model intra prediction from luma component to chroma componenent is enabled.

sps_sao_enabled_flag equal to 1 specifies that the sample adaptive offset process is applied to the reconstructed picture after the deblocking filter process. sps_sao_enabled_flag equal to 0 specifies that the sample adaptive offset process is not applied to the reconstructed picture after the deblocking filter process.

sps_alf_enabled_flag equal to 0 specifies that the adaptive loop filter is disabled. sps_alf_enabled_flag equal to 1 specifies that the adaptive loop filter is enabled.

pcm_enabled_flag equal to 0 specifies that PCM-related syntax (pcm_sample_bit_depth_luma_minus1, pcm_sample_bit_depth_chroma_minus1, log2_min_pcm_luma_coding_block_size_minus3, log2_diff_max_min_pcm_luma_coding_block_size, pcm_loop_filter_disabled_flag, pcm_flag, pcm_alignment_zero_bit syntax elements and pcm_sample( ) syntax structure) is not present in the CVS.

NOTE 2 – When MinCbLog2SizeY is equal to 6 and pcm_enabled_flag is equal to 1, PCM sample data-related syntax (pcm_flag, pcm_alignment_zero_bit syntax elements and pcm_sample( ) syntax structure) is not present in the CVS, because the maximum size of coding blocks that can convey PCM sample data-related syntax is restricted to be less than or equal to Min(CtbLog2SizeY, 5). Hence, MinCbLog2SizeY equal to 6 with pcm_enabled_flag equal to 1 is not an appropriate setting to convey PCM sample data in the CVS.

pcm_sample_bit_depth_luma_minus1 specifies the number of bits used to represent each of PCM sample values of the luma component as follows:

$$PcmBitDepth_Y = pcm\_sample\_bit\_depth\_luma\_minus1 + 1 \quad (7\text{-}28)$$

The value of $PcmBitDepth_Y$ shall be less than or equal to the value of $BitDepth_Y$.

pcm_sample_bit_depth_chroma_minus1 specifies the number of bits used to represent each of PCM sample values of the chroma components as follows:

$$PcmBitDepth_C = pcm\_sample\_bit\_depth\_chroma\_minus1 + 1 \quad (7\text{-}29)$$

The value of $PcmBitDepth_C$ shall be less than or equal to the value of $BitDepth_C$. When ChromaArrayType is equal to 0, pcm_sample_bit_depth_chroma_minus1 is not used in the decoding process and decoders shall ignore its value.

log2_min_pcm_luma_coding_block_size_minus3 plus 3 specifies the minimum size of coding blocks with pcm_flag equal to 1.

The variable Log2MinIpcmCbSizeY is set equal to log2_min_pcm_luma_coding_block_size_minus3 + 3. The value of Log2MinIpcmCbSizeY shall be in the range of Min(MinCbLog2SizeY, 5) to Min(CtbLog2SizeY, 5), inclusive.

log2_diff_max_min_pcm_luma_coding_block_size specifies the difference between the maximum and minimum size of coding blocks with pcm_flag equal to 1.

The variable Log2MaxIpcmCbSizeY is set equal to log2_diff_max_min_pcm_luma_coding_block_size + Log2MinIpcmCbSizeY. The value of Log2MaxIpcmCbSizeY shall be less than or equal to Min(CtbLog2SizeY, 5).

pcm_loop_filter_disabled_flag specifies whether the loop filter process is disabled on reconstructed samples in a coding unit with pcm_flag equal to 1 as follows:

– If pcm_loop_filter_disabled_flag is equal to 1, the deblocking filter and sample adaptive offset filter processes on the reconstructed samples in a coding unit with pcm_flag equal to 1 are disabled.

– Otherwise (pcm_loop_filter_disabled_flag value is equal to 0), the deblocking filter and sample adaptive offset filter processes on the reconstructed samples in a coding unit with pcm_flag equal to 1 are not disabled.

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

sps_temporal_mvp_enabled_flag equal to 1 specifies that slice_temporal_mvp_enabled_flag is present in the slice headers of slices with slice_type not equal to I in the CVS. sps_temporal_mvp_enabled_flag equal to 0 specifies that slice_temporal_mvp_enabled_flag is not present in slice headers and that temporal motion vector predictors are not used in the CVS.

sps_sbtmvp_enabled_flag equal to 1 specifies that subblock-based temporal motion vector predictors may be used in decoding of pictures with all slices having slice_type not equal to I in the CVS. sps_sbtmvp_enabled_flag equal to 0 specifies that subblock-based temporal motion vector predictors are not used in the CVS. When sps_sbtmvp_enabled_flag is not present, it is inferred to be equal to 0

log2_sbtmvp_default_size_minus2 specifies the inferred value of the syntax element log2_sbtmvp_active_size_minus2 in the slice headers of slices with slice_type not equal to I in the CVS when slice_sbtmvp_size_override_flag is equal to 0.

sps_amvr_enabled_flag equal to 1 specifies that adaptive motion vector difference resolution is used in motion vector coding. amvr_enabled_flag equal to 0 specifies that adaptive motion vector difference resolution is not used in motion vector coding.

sps_affine_enabled_flag specifies whether affine model based motion compensation can be used for inter prediction. If sps_affine_enabled_flag is equal to 0, the syntax shall be constrained such that no affine model based motion compensation is used in the CVS, and merge_affine_flag, inter_affine_flag and cu_affine_type_flag are not present in coding unit syntax of the CVS. Otherwise (sps_affine_enabled_flag is equal to 1), affine model based motion compensation can be used in the CVS.

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

sps_mts_intra_enabled_flag equal to 1 specifies that tu_mts_flag may be present in the residual coding syntax for intra coding units. sps_mts_intra_enabled_flag equal to 0 specifies that tu_mts_flag is not present in the residual coding syntax for intra coding units.

sps_mts_inter_enabled_flag specifies that tu_mts_flag may be present in the residual coding syntax for inter coding units. sps_mts_inter_enabled_flag equal to 0 specifies that tu_mts_flag is not present in the residual coding syntax for inter coding units.

sps_ladf_enabled_flag equal to 1, specifies that sps_num_ladf_intervals_minus2, sps_ladf_lowest_interval_qp_offset, sps_ladf_qp_offset[i], and sps_ladf_delta_threshold_minus1[i] are present in the SPS.

sps_num_ladf_intervals_minus2 plus 1 specifies the number of sps_ladf_delta_threshold_minus1[i] and sps_ladf_qp_offset[i] syntax elements that are present in the SPS. The value of sps_num_ladf_intervals_minus2 shall be in the range of 0 to 3, inclusive.

sps_ladf_lowest_interval_qp_offset specifies the offset used to derive the variable qP as specified in clause 8.5.2.6.3. The value of sps_ladf_lowest_interval_qp_offset shall be in the range of 0 to 63, inclusive.

sps_ladf_qp_offset[i] specifies the offset array used to derive the variable qP as specified in clause 8.5.2.6.3. The value of sps_ladf_qp_offset[i] shall be in the range of 0 to 63, inclusive.

sps_ladf_delta_threshold_minus1[i] is used to compute the values of SpsLadfIntervalLowerBound[i], which specifies the lower bound of the i-th luma intensity level interval. The value of sps_ladf_delta_threshold_minus1[i] shall be in the range of 0 to $2^{BitDepthY} - 3$, inclusive.

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

For each value of i in the range of 0 to sps_num_ladf_intervals_minus2, inclusive, the variable SpsLadfIntervalLowerBound[i + 1] is derived as follows:

$$\begin{aligned} SpsLadfIntervalLowerBound[\ i + 1\ ] = SpsLadfIntervalLowerBound[\ i\ ] \\ + sps\_ladf\_delta\_threshold\_minus1[\ i\ ] + 1 \end{aligned} \tag{7-30}$$

7.4.3.2 Picture parameter set RBSP semantics

pps_pic_parameter_set_id identifies the PPS for reference by other syntax elements. The value of pps_pic_parameter_set_id shall be in the range of 0 to 63, inclusive.

pps_seq_parameter_set_id specifies the value of sps_seq_parameter_set_id for the active SPS. The value of pps_seq_parameter_set_id shall be in the range of 0 to 15, inclusive.

init_qp_minus26 plus 26 specifies the initial value of $SliceQp_Y$ for each slice referring to the PPS. The initial value of $SliceQp_Y$ is modified at the slice layer when a non-zero value of slice_qp_delta is decoded. The value of init_qp_minus26 shall be in the range of $-(26 + QpBdOffset_Y)$ to +37, inclusive.

transform_skip_enabled_flag equal to 1 specifies that transform_skip_flag may be present in the residual coding syntax. transform_skip_enabled_flag equal to 0 specifies that transform_skip_flag is not present in the residual coding syntax.

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

diff_cu_qp_delta_depth specifies the coding tree depth difference between the coding units of minimum coding tree depth and coding units of maximum coding tree depth that convey cu_qp_delta_abs and cu_qp_delta_sign_flag. The value range of diff_cu_qp_delta_depth is specified as follows:

– If slice_type is equal to I, the value of diff_cu_qp_delta_depth shall be in the range of 0 to log2_ctu_size_minus2 − log2_min_qt_size_intra_slices_minus2 + MaxMttDepthY, inclusive.

– Otherwise (slice_type is not equal to I), the value of diff_cu_qp_delta_depth shall be in the range of 0 to log2_ctu_size_minus2 − log2_min_qt_size_inter_slices_minus2 + MaxMttDepthY, inclusive.

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

[Ed. (BB): The issue here is that MaxMttDepthY is derived on slice-level. In case of partition_constraints_override_enabled_flag equal to 1, one would need to parse the slice header in order to know the value of MaxMttDepthY.]

pps_cb_qp_offset and **pps_cr_qp_offset** specify the offsets to the luma quantization parameter Qp'_Y used for deriving Qp'_{Cb} and Qp'_{Cr}, respectively. The values of pps_cb_qp_offset and pps_cr_qp_offset shall be in the range of −12 to +12, inclusive. When ChromaArrayType is equal to 0, pps_cb_qp_offset and pps_cr_qp_offset are not used in the decoding process and decoders shall ignore their value.

pps_slice_chroma_qp_offsets_present_flag equal to 1 indicates that the slice_cb_qp_offset and slice_cr_qp_offset syntax elements are present in the associated slice headers. pps_slice_chroma_qp_offsets_present_flag equal to 0 indicates that these syntax elements are not present in the associated slice headers. When ChromaArrayType is equal to 0, pps_slice_chroma_qp_offsets_present_flag shall be equal to 0.

deblocking_filter_control_present_flag equal to 1 specifies the presence of deblocking filter control syntax elements in the PPS. deblocking_filter_control_present_flag equal to 0 specifies the absence of deblocking filter control syntax elements in the PPS.

deblocking_filter_override_enabled_flag equal to 1 specifies the presence of deblocking_filter_override_flag in the slice headers for pictures referring to the PPS. deblocking_filter_override_enabled_flag equal to 0 specifies the absence of deblocking_filter_override_flag in the slice headers for pictures referring to the PPS. When not present, the value of deblocking_filter_override_enabled_flag is inferred to be equal to 0.

pps_deblocking_filter_disabled_flag equal to 1 specifies that the operation of deblocking filter is not applied for slices referring to the PPS in which slice_deblocking_filter_disabled_flag is not present. pps_deblocking_filter_disabled_flag equal to 0 specifies that the operation of the deblocking filter is applied for slices referring to the PPS in which slice_deblocking_filter_disabled_flag is not present. When not present, the value of pps_deblocking_filter_disabled_flag is inferred to be equal to 0.

pps_beta_offset_div2 and **pps_tc_offset_div2** specify the default deblocking parameter offsets for β and tC (divided by 2) that are applied for slices referring to the PPS, unless the default deblocking parameter offsets are overridden by the deblocking parameter offsets present in the slice headers of the slices referring to the PPS. The values of pps_beta_offset_div2 and pps_tc_offset_div2 shall both be in the range of −6 to 6, inclusive. When not present, the value of pps_beta_offset_div2 and pps_tc_offset_div2 are inferred to be equal to 0.

7.4.3.3 End of sequence RBSP semantics

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

7.4.3.4 End of bitstream RBSP semantics

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

7.4.3.5 Slice layer RBSP semantics

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

7.4.3.6 RBSP slice trailing bits semantics

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

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

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

Let the variable RawMinCuBits be derived as follows:

$$RawMinCuBits = MinCbSizeY * MinCbSizeY * (BitDepth_Y + 2 * BitDepth_C / (SubWidthC * SubHeightC)) \quad (7\text{-}31)$$

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

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

7.4.3.7 RBSP trailing bits semantics

rbsp_stop_one_bit shall be equal to 1.

rbsp_alignment_zero_bit shall be equal to 0.

7.4.3.8 Byte alignment semantics

alignment_bit_equal_to_one shall be equal to 1.

alignment_bit_equal_to_zero shall be equal to 0.

7.4.4 Slice header semantics

7.4.4.1 General slice header semantics

When present, the value of the slice header syntax element slice_pic_parameter_set_id shall be the same in all slice headers of a coded picture.

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

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

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

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

Table 7-2 – Name association to slice_type

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

When nal_unit_type is equal to IRAP_NUT, i.e., the picture is an IRAP picture, slice_type shall be equal to 2.

partition_constraints_override_flag equal to 1 specifies that partition constraint parameters are present in the slice header. partition_constraints_override_flag equal to 0 specifies that partition constraint parameters are not present in the slice header. When not present, the value of partition_constraints_override_flag is inferred to be equal to 0.

slice_log2_diff_min_qt_min_cb_luma specifies the difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum coding

block size in luma samples for luma CUs in the current slice. The value of slice_log2_diff_min_qt_min_cb_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of slice_log2_diff_min_qt_min_cb_luma is inferred as follows:

– If slice_type equal to 2 (I), the value of slice_log2_diff_min_qt_min_cb_luma is inferred to be equal to sps_log2_diff_min_qt_min_cb_intra_slice_luma
– Otherwise (slice_type equal to 0 (B) or 1 (P)), the value of slice_log2_diff_min_qt_min_cb_luma is inferred to be equal to sps_log2_diff_min_qt_min_cb_inter_slice.

slice_max_mtt_hierarchy_depth_luma specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in the current slice. The value of slice_max_mtt_hierarchy_depth_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of slice_max_mtt_hierarchy_depth_luma is inferred as follows:

– If slice_type equal to 2 (I), the value of slice_max_mtt_hierarchy_depth_luma is inferred to be equal to sps_max_mtt_hierarchy_depth_intra_slices_luma
– Otherwise (slice_type equal to 0 (B) or 1 (P)), the value of slice_max_mtt_hierarchy_depth_luma is inferred to be equal to sps_max_mtt_hierarchy_depth_inter_slices.

slice_log2_diff_max_bt_min_qt_luma specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in the current slice. The value of slice_log2_diff_max_bt_min_qt_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeY, inclusive. When not present, the value of slice_log2_diff_max_bt_min_qt_luma is inferred as follows:

– If slice_type equal to 2 (I), the value of slice_log2_diff_max_bt_min_qt_luma is inferred to be equal to sps_log2_diff_max_bt_min_qt_intra_slice_luma
– Otherwise (slice_type equal to 0 (B) or 1 (P)), the value of slice_log2_diff_max_bt_min_qt_luma is inferred to be equal to sps_log2_diff_max_bt_min_qt_inter_slice.

slice_log2_diff_max_tt_min_qt_luma specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in in the current slice. The value of slice_log2_diff_max_tt_min_qt_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeY, inclusive. When not present, the value of slice_log2_diff_max_tt_min_qt_luma is inferred as follows:

– If slice_type equal to 2 (I), the value of slice_log2_diff_max_tt_min_qt_luma is inferred to be equal to sps_log2_diff_max_tt_min_qt_intra_slice_luma
– Otherwise (slice_type equal to 0 (B) or 1 (P)), the value of slice_log2_diff_max_tt_min_qt_luma is inferred to be equal to sps_log2_diff_max_tt_min_qt_inter_slice.

slice_log2_diff_min_qt_min_cb_chroma specifies the difference between the base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL_TREE_CHROMA and the base 2 logarithm of the minimum coding block size in luma samples for chroma CUs with treeType equal to DUAL_TREE_CHROMA in the current slice. The value of slice_log2_diff_min_qt_min_cb_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of slice_log2_diff_min_qt_min_cb_chroma is inferred to be equal to sps_log2_diff_min_qt_min_cb_intra_slice_chroma.

slice_max_mtt_hierarchy_depth_chroma specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf with treeType equal to DUAL_TREE_CHROMA in the current slice. The value of slice_max_mtt_hierarchy_depth_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the values of slice_max_mtt_hierarchy_depth_chroma is inferred to be equal to sps_max_mtt_hierarchy_depth_intra_slices_chroma.

slice_log2_diff_max_bt_min_qt_chroma specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL_TREE_CHROMA in the current slice. The value of slice_log2_diff_max_bt_min_qt_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeC, inclusive. When not present, the value of slice_log2_diff_max_bt_min_qt_chroma is inferred to be equal to sps_log2_diff_max_bt_min_qt_intra_slice_chroma

slice_log2_diff_max_tt_min_qt_chroma specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL_TREE_CHROMA in the current slice. The value of slice_log2_diff_max_tt_min_qt_chroma shall be in

the range of 0 to CtbLog2SizeY − MinQtLog2SizeC, inclusive. When not present, the value of slice_log2_diff_max_tt_min_qt_chroma is inferred to be equal to sps_log2_diff_max_tt_min_qt_intra_slice_chroma

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

MinQtLog2SizeY = MinCbLog2SizeY + slice_log2_diff_min_qt_min_cb_luma (7-32)

MinQtLog2SizeC = MinCbLog2SizeY + slice_log2_diff_min_qt_min_cb_chroma (7-33)

MinQtSizeY = 1 << MinQtLog2SizeY (7-34)

MinQtSizeC = 1 << MinQtLog2SizeC (7-35)

MaxBtSizeY = 1 << (MinQtLog2SizeY + slice_log2_diff_max_bt_min_qt_luma) (7-36)

MaxBtSizeC = 1 << (MinQtLog2SizeC + slice_log2_diff_max_bt_min_qt_chroma) (7-37)

MinBtSizeY = 1 << MinCbLog2SizeY (7-38)

MaxTtSizeY = 1 << (MinQtLog2SizeY + slice_log2_diff_max_tt_min_qt_luma) (7-39)

MaxTtSizeC = 1 << (MinQtLog2SizeC + slice_log2_diff_max_tt_min_qt_chroma) (7-40)

MinTtSizeY = 1 << MinCbLog2SizeY (7-41)

MaxMttDepthY = slice_max_mtt_hierarchy_depth_luma (7-42)

MaxMttDepthC = slice_max_mtt_hierarchy_depth_chroma (7-43)

sbtmvp_size_override_flag equal to 1 specifies that the syntax element log2_sbtmvp_active_size_minus2 is present for the current slice. sbtmvp_size_override_flag equal to 0 specifies that the syntax element log2_atmvp_active_size_minus2 is not present and log2_sbtmvp_size_active_minus2 is inferred to be equal to log2_sbtmvp_default_size_minus2.

log2_sbtmvp_active_size_minus2 plus 2 specifies the value of the subblock size that is used for deriving the motion parameters for the subblock-based TMVP of the current slice. When log2_sbtmvp_size_active_minus2 is is not present, it is inferred to be equal to log2_sbtmvp_default_size_minus2. The variable is derived as follows:

Log2SbtmvpSize = log2_sbtmvp_size_active_minus2 + 2 (7-44)

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

mvd_l1_zero_flag equal to 1 indicates that the mvd_coding(x0, y0, 1) syntax structure is not parsed and MvdL1[x0][y0][compIdx] is set equal to 0 for compIdx = 0..1. mvd_l1_zero_flag equal to 0 indicates that the mvd_coding(x0, y0, 1) syntax structure is parsed.

collocated_from_l0_flag equal to 1 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 0. collocated_from_l0_flag equal to 0 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 1. When collocated_from_l0_flag is not present, it is inferred to be equal to 1.

six_minus_max_num_merge_cand specifies the maximum number of merging motion vector prediction (MVP) candidates supported in the slice subtracted from 6. The maximum number of merging MVP candidates, MaxNumMergeCand is derived as follows:

MaxNumMergeCand = 6 − six_minus_max_num_merge_cand (7-45)

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

slice_qp_delta specifies the initial value of Qp_Y to be used for the coding blocks in the slice until modified by the value of CuQpDeltaVal in the coding unit layer. The initial value of the Qp_Y quantization parameter for the slice, $SliceQp_Y$, is derived as follows:

$SliceQp_Y = 26 + init\_qp\_minus26 + slice\_qp\_delta$ (7-46)

The value of $SliceQp_Y$ shall be in the range of −QpBdOffset$_Y$ to +63, inclusive.

slice_cb_qp_offset specifies a difference to be added to the value of pps_cb_qp_offset when determining the value of the Qp'_{Cb} quantization parameter. The value of slice_cb_qp_offset shall be in the range of −12 to +12, inclusive. When slice_cb_qp_offset is not present, it is inferred to be equal to 0. The value of pps_cb_qp_offset + slice_cb_qp_offset shall be in the range of −12 to +12, inclusive.

slice_cr_qp_offset specifies a difference to be added to the value of pps_cr_qp_offset when determining the value of the Qp'_{Cr} quantization parameter. The value of slice_cr_qp_offset shall be in the range of −12 to +12, inclusive. When slice_cr_qp_offset is not present, it is inferred to be equal to 0. The value of pps_cr_qp_offset + slice_cr_qp_offset shall be in the range of −12 to +12, inclusive.

slice_sao_luma_flag equal to 1 specifies that SAO is enabled for the luma component in the current slice; slice_sao_luma_flag equal to 0 specifies that SAO is disabled for the luma component in the current slice. When slice_sao_luma_flag is not present, it is inferred to be equal to 0.

slice_sao_chroma_flag equal to 1 specifies that SAO is enabled for the chroma component in the current slice; slice_sao_chroma_flag equal to 0 specifies that SAO is disabled for the chroma component in the current slice. When slice_sao_chroma_flag is not present, it is inferred to be equal to 0.

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

dep_quant_enabled_flag equal to 0 specifies that dependent quantization is disabled. dep_quant_enabled_flag equal to 1 specifies that dependent quantization is enabled.

sign_data_hiding_enabled_flag equal to 0 specifies that sign bit hiding is disabled. sign_data_hiding_enabled_flag equal to 1 specifies that sign bit hiding is enabled. When sign_data_hiding_enabled_flag is not present, it is inferred to be equal to 0.

deblocking_filter_override_flag equal to 1 specifies that deblocking parameters are present in the slice header. deblocking_filter_override_flag equal to 0 specifies that deblocking parameters are not present in the slice header. When not present, the value of deblocking_filter_override_flag is inferred to be equal to 0.

slice_deblocking_filter_disabled_flag equal to 1 specifies that the operation of the deblocking filter is not applied for the current slice. slice_deblocking_filter_disabled_flag equal to 0 specifies that the operation of the deblocking filter is applied for the current slice. When slice_deblocking_filter_disabled_flag is not present, it is inferred to be equal to pps_deblocking_filter_disabled_flag.

slice_beta_offset_div2 and **slice_tc_offset_div2** specify the deblocking parameter offsets for β and tC (divided by 2) for the current slice. The values of slice_beta_offset_div2 and slice_tc_offset_div2 shall both be in the range of −6 to 6, inclusive. When not present, the values of slice_beta_offset_div2 and slice_tc_offset_div2 are inferred to be equal to pps_beta_offset_div2 and pps_tc_offset_div2, respectively.

7.4.4.2 Adaptive loop filter data semantics

alf_chroma_idc equal to 0 specifies that the adaptive loop filter is not applied to Cb and Cr colour components. alf_chroma_idc equal to 1 indicates that the adaptive loop filter is applied to the Cb colour component. alf_chroma_idc equal to 2 indicates that the adaptive loop filter is applied to the Cr colour component. alf_chroma_idc equal to 3 indicates that the adaptive loop filter is applied to Cb and Cr colour components.

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

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

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

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

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

The maximum value maxVal of the truncated binary binarization tb(v) is set equal to alf_luma_num_filters_signalled_minus1.

alf_luma_coeff_delta_flag equal to 1 indicates that alf_luma_coeff_delta_prediction_flag is not signalled. alf_luma_coeff_delta_flag equal to 0 indicates that alf_luma_coeff_delta_prediction_flag may be signalled.

alf_luma_coeff_delta_prediction_flag equal to 1 specifies that the signalled luma filter coefficient deltas are predicted from the deltas of the previous luma coefficients. alf_luma_coeff_delta_prediction_flag equal to 0 specifies that the signalled luma filter coefficient deltas are not predicted from the deltas of the previous luma coefficients. When not present, alf_luma_coeff_delta_prediction_flag is inferred to be equal to 0.

alf_luma_min_eg_order_minus1 plus 1 specifies the minimum order of the exp-Golomb code for luma filter coefficient signalling. The value of alf_luma_min_eg_order_minus1 shall be in the range of 0 to 6, inclusive.

alf_luma_eg_order_increase_flag[i] equal to 1 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is incremented by 1. alf_luma_eg_order_increase_flag[i] equal to 0 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is not incremented by 1.

The order expGoOrderY[i] of the exp-Golomb code used to decode the values of alf_luma_coeff_delta_abs[sigFiltIdx][j] is derived as follows:

expGoOrderY[i] = alf_luma_min_eg_order_minus1 + 1+ alf_luma_eg_order_increase_flag[i] (7-47)

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

alf_luma_coeff_delta_abs[sigFiltIdx][j] specifies the absolute value of the j-th coefficient delta of the signalled luma filter indicated by sigFiltIdx. When alf_luma_coeff_delta_abs[sigFiltIdx][j] is not present, it is inferred to be equal 0.

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

golombOrderIdxY[ ] = { 0, 0, 1, 0, 0, 1, 2, 1, 0, 0, 1, 2 } (7-48)

k = expGoOrderY[golombOrderIdxY[j]] (7-49)

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

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

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

When alf_luma_coeff_delta_sign[sigFiltIdx][j] is not present, it is inferred to be equal to 0.

The variable filterCoefficients[sigFiltIdx][j] with sigFiltIdx = 0..alf_luma_num_filters_signalled_minus1, j = 0..11 is initialized as follows:

filterCoefficients[sigFiltIdx][j] = alf_luma_coeff_delta_abs[sigFiltIdx][j] * (1 − 2 * alf_luma_coeff_delta_sign[sigFiltIdx][j]) (7-50)

When alf_luma_coeff_delta_prediction_flag is equal 1, filterCoefficients[sigFiltIdx][j] with sigFiltIdx = 1..alf_luma_num_filters_signalled_minus1 and j = 0..11 are modified as follows:

filterCoefficients[sigFiltIdx][j] += filterCoefficients[sigFiltIdx − 1][j] (7-51)

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

$AlfCoeff_L$[filtIdx][j] = filterCoefficients[alf_luma_coeff_delta_idx[filtIdx]][j] (7-52)

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

$AlfCoeff_L$[filtIdx][12] = $128 - \Sigma_k$ ($AlfCoeff_L$[filtIdx][k] << 1), with k = 0..11 (7-53)

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

alf_chroma_min_eg_order_minus1 plus 1 specifies the minimum order of the exp-Golomb code for chroma filter coefficient signalling. The value of alf_chroma_min_eg_order_minus1 shall be in the range of 0 to 6, inclusive.

alf_chroma_eg_order_increase_flag[i] equal to 1 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is incremented by 1. alf_chroma_eg_order_increase_flag[i] equal to 0 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is not incremented by 1

The order expGoOrderC[i] of the exp-Golomb code used to decode the values of alf_chroma_coeff_abs[j] is derived as follows:

expGoOrderC[i] = alf_chroma_min_eg_order_minus1 + 1+ alf_chroma_eg_order_increase_flag[i] (7-54)

alf_chroma_coeff_abs[j] specifies the absolute value of the j-th chroma filter coefficient. When alf_chroma_coeff_abs[j] is not present, it is inferred to be equal 0. It is a requirement of bitstream conformance that the values of alf_chroma_coeff_abs[j] shall be in the range of 0 to $2^7 - 1$, inclusive.

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

golombOrderIdxC[ ] = { 0, 0, 1, 0, 0, 1 } (7-55)

k = expGoOrderC[golombOrderIdxC[j]] (7-56)

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

– If alf_chroma_coeff_sign[j] is equal to 0, the corresponding chroma filter coefficient has a positive value.

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

When alf_chroma_coeff_sign[j] is not present, it is inferred to be equal to 0.

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

$AlfCoeff_C[j]$ = alf_chroma_coeff_abs[j] * (1 − 2 * alf_chroma_coeff_sign[j]) (7-57)

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

$AlfCoeff_C[6] = 128 - \Sigma_k (AlfCoeff_C[k] << 1)$, with k = 0..5 (7-58)

It is a requirement of bitstream conformance that the values of $AlfCoeff_C[j]$ with j = 0..5 shall be in the range of $-2^7 - 1$ to $2^7 - 1$, inclusive and that the values of $AlfCoeff_C[6]$ shall be in the range of 0 to $2^8 - 1$, inclusive.

7.4.5 Slice data semantics

7.4.5.1 General slice data semantics

end_of_slice_flag equal to 0 specifies that another CTU is following in the slice. end_of_slice_flag equal to 1 specifies the end of the slice, i.e., that no further CTU follows in the slice.

7.4.5.2 Coding tree unit semantics

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

alf_ctb_flag[cIdx][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize] equal to 1 specifies that the adaptive loop filter is applied to the coding tree block of the colour component indicated by cIdx of the coding tree unit at luma location (xCtb, yCtb). alf_ctb_flag[cIdx][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize] equal to 0 specifies that the adaptive loop filter is not applied to the coding tree block of the colour component indicated by cIdx of the coding tree unit at luma location (xCtb, yCtb).

When alf_ctb_flag[cIdx][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize] is not present, it is inferred to be equal to 0.

7.4.5.3 Sample adaptive offset semantics

sao_merge_left_flag equal to 1 specifies that the syntax elements sao_type_idx_luma, sao_type_idx_chroma, sao_band_position, sao_eo_class_luma, sao_eo_class_chroma, sao_offset_abs and sao_offset_sign are derived from the corresponding syntax elements of the left CTB. sao_merge_left_flag equal to 0 specifies that these syntax elements are not

derived from the corresponding syntax elements of the left CTB. When sao_merge_left_flag is not present, it is inferred to be equal to 0.

sao_merge_up_flag equal to 1 specifies that the syntax elements sao_type_idx_luma, sao_type_idx_chroma, sao_band_position, sao_eo_class_luma, sao_eo_class_chroma, sao_offset_abs and sao_offset_sign are derived from the corresponding syntax elements of the above CTB. sao_merge_up_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the above CTB. When sao_merge_up_flag is not present, it is inferred to be equal to 0.

sao_type_idx_luma specifies the offset type for the luma component. The array SaoTypeIdx[cIdx][rx][ry] specifies the offset type as specified in Table 7-3 for the CTB at the location (rx, ry) for the colour component cIdx. The value of SaoTypeIdx[0][rx][ry] is derived as follows:

– If sao_type_idx_luma is present, SaoTypeIdx[0][rx][ry] is set equal to sao_type_idx_luma.

– Otherwise (sao_type_idx_luma is not present), SaoTypeIdx[0][rx][ry] is derived as follows:

  – If sao_merge_left_flag is equal to 1, SaoTypeIdx[0][rx][ry] is set equal to SaoTypeIdx[0][rx − 1][ry].

  – Otherwise, if sao_merge_up_flag is equal to 1, SaoTypeIdx[0][rx][ry] is set equal to SaoTypeIdx[0][rx][ry − 1].

  – Otherwise, SaoTypeIdx[0][rx][ry] is set equal to 0.

sao_type_idx_chroma specifies the offset type for the chroma components. The values of SaoTypeIdx[cIdx][rx][ry] are derived as follows for cIdx equal to 1..2:

– If sao_type_idx_chroma is present, SaoTypeIdx[cIdx][rx][ry] is set equal to sao_type_idx_chroma.

– Otherwise (sao_type_idx_chroma is not present), SaoTypeIdx[cIdx][rx][ry] is derived as follows:

  – If sao_merge_left_flag is equal to 1, SaoTypeIdx[cIdx][rx][ry] is set equal to SaoTypeIdx[cIdx][rx − 1][ry].

  – Otherwise, if sao_merge_up_flag is equal to 1, SaoTypeIdx[cIdx][rx][ry] is set equal to SaoTypeIdx[cIdx][rx][ry − 1].

  – Otherwise, SaoTypeIdx[cIdx][rx][ry] is set equal to 0.

Table 7-3 – Specification of the SAO type

SaoTypeIdx[cIdx][rx][ry]	SAO type (informative)
0	Not applied
1	Band offset
2	Edge offset

sao_offset_abs[cIdx][rx][ry][i] specifies the offset value of i-th category for the CTB at the location (rx, ry) for the colour component cIdx.

When sao_offset_abs[cIdx][rx][ry][i] is not present, it is inferred as follows:

– If sao_merge_left_flag is equal to 1, sao_offset_abs[cIdx][rx][ry][i] is inferred to be equal to sao_offset_abs[cIdx][rx − 1][ry][i].

– Otherwise, if sao_merge_up_flag is equal to 1, sao_offset_abs[cIdx][rx][ry][i] is inferred to be equal to sao_offset_abs[cIdx][rx][ry − 1][i].

– Otherwise, sao_offset_abs[cIdx][rx][ry][i] is inferred to be equal to 0.

sao_offset_sign[cIdx][rx][ry][i] specifies the sign of the offset value of i-th category for the CTB at the location (rx, ry) for the colour component cIdx.

When sao_offset_sign[cIdx][rx][ry][i] is not present, it is inferred as follows:

– If sao_merge_left_flag is equal to 1, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal to sao_offset_sign[cIdx][rx − 1][ry][i].

– Otherwise, if sao_merge_up_flag is equal to 1, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal to sao_offset_sign[cIdx][rx][ry − 1][i].

– Otherwise, if SaoTypeIdx[cIdx][rx][ry] is equal to 2, the following applies:
  – If i is equal to 0 or 1, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal 0.
  – Otherwise (i is equal to 2 or 3), sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal 1.
– Otherwise, sao_offset_sign[cIdx][rx][ry][i] is inferred to be equal 0.

The list SaoOffsetVal[cIdx][rx][ry][i] for i ranging from 0 to 4, inclusive, is derived as follows:

```
SaoOffsetVal[ cIdx ][ rx ][ ry ][ 0 ] = 0
for( i = 0; i < 4; i++ )
SaoOffsetVal[ cIdx ][ rx ][ ry ][ i + 1 ] = ( 1 − 2 * sao_offset_sign[ cIdx ][ rx ][ ry ][ i ] ) *
        sao_offset_abs[ cIdx ][ rx ][ ry ][ i ]                                    (7-59)
```

sao_band_position[cIdx][rx][ry] specifies the displacement of the band offset of the sample range when SaoTypeIdx[cIdx][rx][ry] is equal to 1.

When sao_band_position[cIdx][rx][ry] is not present, it is inferred as follows:

– If sao_merge_left_flag is equal to 1, sao_band_position[cIdx][rx][ry] is inferred to be equal to sao_band_position[cIdx][rx − 1][ry].
– Otherwise, if sao_merge_up_flag is equal to 1, sao_band_position[cIdx][rx][ry] is inferred to be equal to sao_band_position[cIdx][rx][ry − 1].
– Otherwise, sao_band_position[cIdx][rx][ry] is inferred to be equal to 0.

sao_eo_class_luma specifies the edge offset class for the luma component. The array SaoEoClass[cIdx][rx][ry] specifies the offset type as specified in Table 7-4 for the CTB at the location (rx, ry) for the colour component cIdx. The value of SaoEoClass[0][rx][ry] is derived as follows:

– If sao_eo_class_luma is present, SaoEoClass[0][rx][ry] is set equal to sao_eo_class_luma.
– Otherwise (sao_eo_class_luma is not present), SaoEoClass[0][rx][ry] is derived as follows:
  – If sao_merge_left_flag is equal to 1, SaoEoClass[0][rx][ry] is set equal to SaoEoClass[0][rx − 1][ry].
  – Otherwise, if sao_merge_up_flag is equal to 1, SaoEoClass[0][rx][ry] is set equal to SaoEoClass[0][rx][ry − 1].
  – Otherwise, SaoEoClass[0][rx][ry] is set equal to 0.

sao_eo_class_chroma specifies the edge offset class for the chroma components. The values of SaoEoClass[cIdx][rx][ry] are derived as follows for cIdx equal to 1..2:

– If sao_eo_class_chroma is present, SaoEoClass[cIdx][rx][ry] is set equal to sao_eo_class_chroma.
– Otherwise (sao_eo_class_chroma is not present), SaoEoClass[cIdx][rx][ry] is derived as follows:
  – If sao_merge_left_flag is equal to 1, SaoEoClass[cIdx][rx][ry] is set equal to SaoEoClass[cIdx][rx − 1][ry].
  – Otherwise, if sao_merge_up_flag is equal to 1, SaoEoClass[cIdx][rx][ry] is set equal to SaoEoClass[cIdx][rx][ry − 1].
  – Otherwise, SaoEoClass[cIdx][rx][ry] is set equal to 0.

Table 7-4 – Specification of the SAO edge offset class

SaoEoClass[cIdx][rx][ry]	SAO edge offset class (informative)
0	1D 0-degree edge offset
1	1D 90-degree edge offset
2	1D 135-degree edge offset
3	1D 45-degree edge offset

7.4.5.4 Coding quadtree semantics

qt_split_cu_flag[x0][y0] specifies whether a coding unit is split into coding units with half horizontal and vertical size. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When qt_split_cu_flag[x0][y0] is not present, the following applies:

– If one or more of the following conditions are true, the value of qt_split_cu_flag[x0][y0] is inferred to be equal to 1:
  – x0 + (1 << log2CbSize) is greater than pic_width_in_luma_samples and (1 << log2CbSize) is greater than MaxBtSizeC if treeType is equal to DUAL_TREE_CHROMA or greater than MaxBtSizeY otherwise.
  – y0 + (1 << log2CbSize) is greater than pic_height_in_luma_samples and (1 << log2CbSize) is greater than MaxBtSizeC if treeType is equal to DUAL_TREE_CHROMA or greater than MaxBtSizeY otherwise.
– Otherwise, if all of the following conditions are true, the value of qt_split_cu_flag[x0][y0] is inferred to be equal to 1:
  – x0 + (1 << log2CbSize) is greater than pic_width_in_luma_samples.
  – y0 + (1 << log2CbSize) is greater than pic_height_in_luma_samples.
  – (1 << log2CbSize) is greater than MinQtSizeC if treeType is equal to DUAL_TREE_CHROMA or greater than MinQtSizeY otherwise.
– Otherwise, the value of qt_split_cu_flag[x0][y0] is inferred to be equal to 0.

7.4.5.5 Multi-type tree semantics

The variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer allowSplitTtHor are derived as follows:

– The variables maxBtSize, maxTtSize and maxMttDepth are derived as follows:
  – If treeType is equal to DUAL_TREE_CHROMA, maxBtSize, maxTtSize and maxMttDepth are set equal to MaxBtSizeC, MaxTtSizeC and MaxMttDepthC + depthOffset, respectively.
  – Otherwise, maxBtSize, maxTtSize and maxMttDepth are set equal to MaxBtSizeY, MaxTtSizeY and MaxMttDepthY + depthOffset, respectively.
– The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT_BT_VER, the coding block width cbWidth, the coding block height cbHeight, the location (x0, y0), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, the maximum binary tree size maxBtSize, and the current partition index partIdx as inputs, and the output is assigned to allowSplitBtVer.
– The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT_BT_HOR, the coding block height cbHeight, the coding block width cbWidth, the location (x0, y0), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, the maximum binary tree size maxBtSize, and the current partition index partIdx as inputs, and the output is assigned to allowSplitBtHor.
– The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT_TT_VER, the coding block width cbWidth, the coding block height cbHeight, the location (x0, y0), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, and the maximum ternary tree size maxTtSize as inputs, and the output is assigned to allowSplitTtVer.
– The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT_TT_HOR, the coding block height cbHeight, the coding block width cbWidth, the location (x0, y0), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, and the maximum ternary tree size maxTtSize as inputs, and the output is assigned to allowSplitTtHor.

mtt_split_cu_flag equal to 0 specifies that a coding unit is not split. mtt_split_cu_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split or into three coding units using a ternary split as indicated by the syntax element mtt_split_cu_binary_flag. The binary or ternary split can be either vertical or horizontal as indicated by the syntax element mtt_split_cu_vertical_flag.

When mtt_split_cu_flag is not present, the value of mtt_split_cu_flag is inferred as follows:

– If one or more of the following conditions are true, the value of mtt_split_cu_flag is inferred to be equal to 1:

  - x0 + cbWidth is greater than pic_width_in_luma_samples.
  - y0 + cbHeight is greater than pic_height_in_luma_samples.
- Otherwise, the value of mtt_split_cu_flag is inferred to be equal to 0.

mtt_split_cu_vertical_flag equal to 0 specifies that a coding unit is split horizontally. mtt_split_cu_vertical_flag equal to 1 specifies that a coding unit is split vertically

When mtt_split_cu_vertical_flag is not present, it is inferred as follows:

- If allowSplitBtHor is equal to TRUE or allowSplitTtHor is equal to TRUE, the value of mtt_split_cu_vertical_flag is inferred to be equal to 0.
- Otherwise, the value of mtt_split_cu_vertical_flag is inferred to be equal to 1.

mtt_split_cu_binary_flag equal to 0 specifies that a coding unit is split into three coding units using a ternary split. mtt_split_cu_binary_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split.

When mtt_split_cu_binary_flag is not present, it is inferred as follows:

- If allowSplitBtVer is equal to FALSE and allowSplitBtHor is equal to FALSE, the value of mtt_split_cu_binary_flag is inferred to be equal to 0.
- Otherwise, if allowSplitTtVer is equal to FALSE and allowSplitTtHor is equal to FALSE, the value of mtt_split_cu_binary_flag is inferred as to be equal to 1.
- Otherwise, if allowSplitBtHor is equal to TRUE and allowSplitTtVer is equal to TRUE, the value of mtt_split_cu_binary_flag is inferred to be equal to !mtt_split_cu_vertical_flag.
- Otherwise (allowSplitBtVer is equal to TRUE and allowSplitTtHor is equal to TRUE), the value of mtt_split_cu_binary_flag is inferred to be equal to mtt_split_cu_vertical_flag.

The variable MttSplitMode[x][y][mttDepth] is derived from the value of mtt_split_cu_vertical_flag and from the value of mtt_split_cu_binary_flag as defined in Table 7-5 for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

Figure 7-1 – Multi-type tree spliting modes indicated by MttSplitMode (informative)

MttSplitMode[x0][y0][mttDepth] represents horizontal and vertical binary and ternary splittings of a coding unit within the multi-type tree as illustrated in Figure 7-1. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7-5 – Specification of MttSplitMode[x][y][mttDepth] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1

MttSplitMode[x0][y0][mttDepth]	mtt_split_cu_vertical_flag	mtt_split_cu_binary_flag
SPLIT_TT_HOR	0	0
SPLIT_BT_HOR	0	1
SPLIT_TT_VER	1	0
SPLIT_BT_VER	1	1

7.4.5.6 Coding unit semantics

The following assignments are made for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

CbPosX[x][y] = x0 (7-60)

CbPosY[x][y] = y0 (7-61)

CbWidth[x][y] = cbWidth (7-62)

CbHeight[x][y] = cbHeight (7-63)

cu_skip_flag[x0][y0] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, no more syntax elements except the merging candidate index merge_idx[x0][y0] and the merge affine flag merge_affine_flag[x0][y0] are parsed after cu_skip_flag[x0][y0]. cu_skip_flag[x0][y0] equal to 0 specifies that the coding unit is not skipped. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When cu_skip_flag[x0][y0] is not present, it is inferred to be equal to 0.

pred_mode_flag equal to 0 specifies that the current coding unit is coded in inter prediction mode. pred_mode_flag equal to 1 specifies that the current coding unit is coded in intra prediction mode. The variable CuPredMode[x][y] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

– If pred_mode_flag is equal to 0, CuPredMode[x][y] is set equal to MODE_INTER.

– Otherwise (pred_mode_flag is equal to 1), CuPredMode[x][y] is set equal to MODE_INTRA.

When pred_mode_flag is not present, the variable CuPredMode[x][y] is inferred to be equal to MODE_INTRA for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

pcm_flag[x0][y0] equal to 1 specifies that the pcm_sample( ) syntax structure is present and the transform_tree( ) syntax structure is not present in the coding unit including the luma coding block at the location (x0, y0). pcm_flag[x0][y0] equal to 0 specifies that pcm_sample( ) syntax structure is not present. When pcm_flag[x0][y0] is not present, it is inferred to be equal to 0.

The value of pcm_flag[x0 + i][y0 + j] with i = 1..cbWidth − 1, j = 1..cbHeight − 1 is inferred to be equal to pcm_flag[x0][y0].

pcm_alignment_zero_bit is a bit equal to 0.

intra_luma_ref_idx[x0][y0] specifies the intra prediction reference line index IntraLumaRefLineIdx[x][y] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1 as specified in Table 7-6.

When intra_luma_ref_idx[x0][y0] is not present it is inferred to be equal to 0.

Table 7-6 – Specification of IntraLumaRefLineIdx[x][y] based on intra_luma_ref_idx[x0][y0].

intra_luma_ref_idx[x0][y0]	IntraLumaRefLineIdx[x][y] x = x0..x0 + cbWidth − 1 y = y0..y0 + cbHeight − 1
0	0
1	1
2	3

The syntax elements **intra_luma_mpm_flag**[x0][y0], **intra_luma_mpm_idx**[x0][y0] and **intra_luma_mpm_remainder**[x0][y0] specify the intra prediction mode for luma samples. The array indices x0, y0 specify the location (x0 , y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When intra_luma_mpm_flag[x0][y0] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted coding unit according to clause 8.2.2.

When intra_luma_mpm_flag[x0][y0] is not present, it is inferred to be equal to 1.

intra_chroma_pred_mode[x0][y0] specifies the intra prediction mode for chroma samples. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

mvp_l0_flag[x0][y0] specifies the motion vector predictor index of list 0 where x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When mvp_l0_flag[x0][y0] is not present, it is inferred to be equal to 0.

mvp_l1_flag[x0][y0] has the same semantics as mvp_l0_flag, with l0 and list 0 replaced by l1 and list 1, respectively.

merge_flag[x0][y0] specifies whether the inter prediction parameters for the current coding unit are inferred from a neighbouring inter-predicted partition. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When merge_flag[x0][y0] is not present, it is inferred as follows:

– If cu_skip_flag[x0][y0] is equal to 1, merge_flag[x0][y0] is inferred to be equal to 1.

– Otherwise, merge_flag[x0][y0] is inferred to be equal to 0.

merge_idx[x0][y0] specifies the merging candidate index of the merging candidate list where x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When merge_idx[x0][y0] is not present, it is inferred to be equal to 0.

inter_pred_idc[x0][y0] specifies whether list0, list1, or bi-prediction is used for the current coding unit according to Table 7-7. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7-7 – Name association to inter prediction mode

inter_pred_idc	Name of inter_pred_idc	
	(cbWidth + cbHeight) != 12	(cbWidth + cbHeight) = = 12
0	PRED_L0	PRED_L0
1	PRED_L1	PRED_L1
2	PRED_BI	n.a.

When inter_pred_idc[x0][y0] is not present, it is inferred to be equal to PRED_L0.

ref_idx_l0[x0][y0] specifies the list 0 reference picture index for the current coding unit. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When ref_idx_l0[x0][y0] is not present it is inferred to be equal to 0.

ref_idx_l1[x0][y0] has the same semantics as ref_idx_l0, with l0 and list 0 replaced by l1 and list 1, respectively.

merge_affine_flag[x0][y0] specifies whether the affine prediction parameters for the current coding unit are inferred from a neighbouring block with affine model based motion compensation. When merge_affine_flag[x0][y0] is not present, it is inferred to be equal to 0.

inter_affine_flag[x0][y0] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, affine model based motion compensation is used to generate the prediction samples of the current coding unit. inter_affine_flag[x0][y0] equal to 0 specifies that the coding unit is not predicted by affine model based motion compensation. When inter_affine_flag[x0][y0] is not present, it is inferred to be equal to 0.

cu_affine_type_flag[x0][y0] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, 6-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit. cu_affine_type_flag[x0][y0] equal to 0 specifies that 4-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit.

MotionModelIdc[x][y] represents motion model of a coding unit as illustrated in Table 7-8. The array indices x, y specify the luma sample location (x, y) relative to the top-left luma sample of the picture.

The variable MotionModelIdc[x][y] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

– If merge_flag[x0][y0] is equal to 1, the following applies:

$$MotionModelIdc[x][y] = merge\_affine\_flag[x0][y0] \quad (7\text{-}64)$$

– Otherwise (merge_flag[x0][y0] is equal to 0), the following applies:

$$MotionModelIdc[x][y] = inter\_affine\_flag[x0][y0] + cu\_affine\_type\_flag[x0][y0] \quad (7\text{-}65)$$

Table 7-8 – Interpretation of MotionModelIdc[x0][y0]

MotionModelIdc[x][y]	**Motion model for motion compensation**
0	Translational motion
1	4-parameter affine motion
2	6-parameter affine motion

amvr_mode[x0][y0] specifies the resolution of motion vector difference. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When amvr_mode[x0][y0] is not present, it is inferred to be equal to 0.

The variable MvShift is set equal to amvr_mode[x0][y0] << 1 and the variables MvdL0[x0][y0][0], MvdL0[x0][y0][1], MvdL1[x0][y0][0], MvdL1[x0][y0][1] are modified as follows:

$$MvdL0[x0][y0][0] = MvdL0[x0][y0][0] << (MvShift + 2) \quad (7\text{-}66)$$

$$MvdL0[x0][y0][1] = MvdL0[x0][y0][1] << (MvShift + 2) \quad (7\text{-}67)$$

$$MvdL1[x0][y0][0] = MvdL1[x0][y0][0] << (MvShift + 2) \quad (7\text{-}68)$$

$$MvdL1[x0][y0][1] = MvdL1[x0][y0][1] << (MvShift + 2) \quad (7\text{-}69)$$

cu_cbf equal to 1 specifies that the transform_tree( ) syntax structure is present for the current coding unit. cu_cbf equal to 0 specifies that the transform_tree( ) syntax structure is not present for the current coding unit.

When cu_cbf is not present, it is inferred as follows:

– If cu_skip_flag[x0][y0] is equal to 1, cu_cbf is inferred to be equal to 0.

– Otherwise, cu_cbf is inferred to be equal to 1.

7.4.5.7 Motion vector difference semantics

abs_mvd_greater0_flag[compIdx] specifies whether the absolute value of a motion vector component difference is greater than 0.

abs_mvd_greater1_flag[compIdx] specifies whether the absolute value of a motion vector component difference is greater than 1.

When abs_mvd_greater1_flag[compIdx] is not present, it is inferred to be equal to 0.

abs_mvd_minus2[compIdx] plus 2 specifies the absolute value of a motion vector component difference.

When abs_mvd_minus2[compIdx] is not present, it is inferred to be equal to −1.

mvd_sign_flag[compIdx] specifies the sign of a motion vector component difference as follows:

– If mvd_sign_flag[compIdx] is equal to 0, the corresponding motion vector component difference has a positive value.

– Otherwise (mvd_sign_flag[compIdx] is equal to 1), the corresponding motion vector component difference has a negative value.

When mvd_sign_flag[compIdx] is not present, it is inferred to be equal to 0.

The motion vector difference lMvd[compIdx] for compIdx = 0..1 is derived as follows:

$$lMvd[compIdx] = abs\_mvd\_greater0\_flag[compIdx] * (abs\_mvd\_minus2[compIdx] + 2) * (1 - 2 * mvd\_sign\_flag[compIdx]) \quad (7\text{-}70)$$

The value of lMvd[compIdx] shall be in the range of -2^{15} to $2^{15} - 1$, inclusive.

Depending in the value of MotionModelIdc[x][y], motion vector differences are dervied as follows:

– If MotionModelIdc[x][y] is equal to 0, the variable MvdLX[x0][y0][compIdx], with X being 0 or 1, specifies the difference between a list X vector component to be used and its prediction. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of

the picture. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

  – If refList is equal to 0, MvdL0[x0][y0][compIdx] is set equal to lMvd[compIdx] for compIdx = 0..1.

  – Otherwise (refList is equal to 1), MvdL1[x0][y0][compIdx] is set equal to lMvd[compIdx] for compIdx = 0..1.

– Otherwise (MotionModelIdc[x][y] is not equal to 0), the variable MvdCpLX[x0][y0][cpIdx][compIdx], with X being 0 or 1, specifies the difference in 1/16 fractional-sample accuracy between a list X vector component to be used and its prediction. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, the array index cpIdx specifies the control point index. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.

  – If refList is equal to 0, MvdCpL0[x0][y0][cpIdx][compIdx] is set equal to (lMvd[compIdx] << 2) for compIdx = 0..1.

  – Otherwise (refList is equal to 1), MvdCpL1[x0][y0][cpIdx][compIdx] is set equal to (lMvd[compIdx] << 2) for compIdx = 0..1.

7.4.5.8 PCM sample semantics

pcm_sample_luma[i] represents a coded luma sample value in the raster scan within the coding unit. The number of bits used to represent each of these samples is $PcmBitDepth_Y$.

pcm_sample_chroma[i] represents a coded chroma sample value in the raster scan within the coding unit. The first half of the values represent coded Cb samples and the remaining half of the values represent coded Cr samples. The number of bits used to represent each of these samples is $PcmBitDepth_C$.

7.4.5.9 Transform tree semantics

[Ed. (BB): The transform scheme does not have any syntax for spliting a CU into TUs. However, if the height or width of a CU is larger than the current maximum transform length of 64 luma samples or the corresponding chroma sample length, the CU will be implicitly split to divide it into TUs.]

7.4.5.10 Transform unit semantics

The transform coefficient levels are represented by the arrays TransCoeffLevel[x0][y0][cIdx][xC][yC]. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for Y, 1 for Cb, and 2 for Cr. The array indices xC and yC specify the transform coefficient location (xC, yC) within the current transform block. When the value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is not specified in clause 7.3.4.11, it is inferred to be equal to 0.

tu_cbf_luma[x0][y0] equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

When tu_cbf_luma[x0][y0] is not present, its value is inferred to be equal to 0.

tu_cbf_cb[x0][y0] equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location (x0, y0) of the considered transform block.

When tu_cbf_cb[x0][y0] is not present, its value is inferred to be equal to 0.

tu_cbf_cr[x0][y0] equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location (x0, y0) of the considered transform block.

When tu_cbf_cr[x0][y0] is not present, its value is inferred to be equal to 0.

cu_qp_delta_abs specifies the absolute value of the difference CuQpDeltaVal between the quantization parameter of the current coding unit and its prediction.

cu_qp_delta_sign_flag specifies the sign of CuQpDeltaVal as follows:

– If cu_qp_delta_sign_flag is equal to 0, the corresponding CuQpDeltaVal has a positive value.

– Otherwise (cu_qp_delta_sign_flag is equal to 1), the corresponding CuQpDeltaVal has a negative value.

When cu_qp_delta_sign_flag is not present, it is inferred to be equal to 0.

When cu_qp_delta_abs is present, the variables IsCuQpDeltaCoded and CuQpDeltaVal are derived as follows:

$$IsCuQpDeltaCoded = 1 \tag{7-71}$$

$$CuQpDeltaVal = cu\_qp\_delta\_abs * (1 - 2 * cu\_qp\_delta\_sign\_flag) \tag{7-72}$$

The value of CuQpDeltaVal shall be in the range of −(32 + QpBdOffsetY / 2) to +(31 + QpBdOffsetY / 2), inclusive.

tu_mts_flag[x0][y0] equal to 1 specifies that multiple transform selection is applied to the residual samples of the associated luma transform block. tu_mts_flag[x0][y0] equal to 0 specifies that multiple transform selection is not applied to the residual samples of the associated luma transform block. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

When tu_mts_flag[x0][y0] is not present, it is inferred to be equal to 0.

7.4.5.11 Residual coding semantics

The array AbsLevel[xC][yC] represents an array of absolute values of transform coefficient levels for the current transform block and the array AbsLevelPass1[xC][yC] represents an array of partially reconstructed absolute values of transform coefficient levels for the current transform block. The array indices xC and yC specify the transform coefficient location (xC, yC) within the current transform block. When the value of AbsLevel[xC][yC] is not specified in clause 7.3.4.11, it is inferred to be equal to 0. When the value of AbsLevelPass1[xC][yC] is not specified in clause 7.3.4.11, it is inferred to be equal to 0.

The variables CoeffMin and CoeffMax specifying the minimum and maximum transform coefficient values are derived as follows:

$$CoeffMin = -(1 << 15) \tag{7-73}$$

$$CoeffMax = (1 << 15) - 1 \tag{7-74}$$

The array QStateTransTable[ ][ ] is specified as follows:

$$QStateTransTable[ ][ ] = \{ \{ 0, 2 \}, \{ 2, 0 \}, \{ 1, 3 \}, \{ 3, 1 \} \} \tag{7-75}$$

transform_skip_flag[x0][y0][cIdx] specifies whether a transform is applied to the associated transform block or not: The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb and equal to 2 for Cr. transform_skip_flag[x0][y0][cIdx] equal to 1 specifies that no transform is applied to the current transform block. transform_skip_flag[x0][y0][cIdx] equal to 0 specifies that the decision whether transform is applied to the current transform block or not depends on other syntax elements. When transform_skip_flag[x0][y0][cIdx] is not present, it is inferred to be equal to 0.

last_sig_coeff_x_prefix specifies the prefix of the column position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_x_prefix shall be in the range of 0 to (log2TbWidth << 1) − 1, inclusive.

last_sig_coeff_y_prefix specifies the prefix of the row position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_y_prefix shall be in the range of 0 to (log2TbHeight << 1) − 1, inclusive.

last_sig_coeff_x_suffix specifies the suffix of the column position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_x_suffix shall be in the range of 0 to (1 << ((last_sig_coeff_x_prefix >> 1) − 1)) − 1, inclusive.

The column position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffX is derived as follows:

– If last_sig_coeff_x_suffix is not present, the following applies:

$$LastSignificantCoeffX = last\_sig\_coeff\_x\_prefix \tag{7-76}$$

– Otherwise (last_sig_coeff_x_suffix is present), the following applies:

$$LastSignificantCoeffX = (1 << ((last\_sig\_coeff\_x\_prefix >> 1) - 1)) * (2 + (last\_sig\_coeff\_x\_prefix \& 1)) + last\_sig\_coeff\_x\_suffix \quad (7\text{-}77)$$

last_sig_coeff_y_suffix specifies the suffix of the row position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_y_suffix shall be in the range of 0 to (1 << ((last_sig_coeff_y_prefix >> 1) − 1)) − 1, inclusive.

The row position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffY is derived as follows:

– If last_sig_coeff_y_suffix is not present, the following applies:

$$LastSignificantCoeffY = last\_sig\_coeff\_y\_prefix \quad (7\text{-}78)$$

– Otherwise (last_sig_coeff_y_suffix is present), the following applies:

$$LastSignificantCoeffY = (1 << ((last\_sig\_coeff\_y\_prefix >> 1) - 1)) * (2 + (last\_sig\_coeff\_y\_prefix \& 1)) + last\_sig\_coeff\_y\_suffix \quad (7\text{-}79)$$

coded_sub_block_flag[xS][yS] specifies the following for the subblock at location (xS, yS) within the current transform block, where a subblock is a (4x4) array of 16 transform coefficient levels:

– If coded_sub_block_flag[xS][yS] is equal to 0, the 16 transform coefficient levels of the subblock at location (xS, yS) are inferred to be equal to 0.

– Otherwise (coded_sub_block_flag[xS][yS] is equal to 1), the following applies:

  – If (xS, yS) is equal to (0, 0) and (LastSignificantCoeffX, LastSignificantCoeffY) is not equal to (0, 0), at least one of the 16 sig_coeff_flag syntax elements is present for the subblock at location (xS, yS).

  – Otherwise, at least one of the 16 transform coefficient levels of the subblock at location (xS, yS) has a non-zero value.

When coded_sub_block_flag[xS][yS] is not present, it is inferred as follows:

– If one or more of the following conditions are true, coded_sub_block_flag[xS][yS] is inferred to be equal to 1:

  – (xS, yS) is equal to (0, 0).

  – (xS, yS) is equal to (LastSignificantCoeffX >> 2, LastSignificantCoeffY >> 2).

– Otherwise, coded_sub_block_flag[xS][yS] is inferred to be equal to 0.

sig_coeff_flag[xC][yC] specifies for the transform coefficient location (xC, yC) within the current transform block whether the corresponding transform coefficient level at the location (xC, yC) is non-zero as follows:

– If sig_coeff_flag[xC][yC] is equal to 0, the transform coefficient level at the location (xC, yC) is set equal to 0.

– Otherwise (sig_coeff_flag[xC][yC] is equal to 1), the transform coefficient level at the location (xC, yC) has a non-zero value.

When sig_coeff_flag[xC][yC] is not present, it is inferred as follows:

– If (xC, yC) is the last significant location (LastSignificantCoeffX, LastSignificantCoeffY) in scan order or all of the following conditions are truet[xC][yC] is inferred to be equal to 1:

  – (xC & 3, yC & 3) is equal to (0, 0).

  – inferSbDcSigCoeffFlag is equal to 1.

  – coded_sub_block_flag[xS][yS] is equal to 1.

– Otherwise, sig_coeff_flag[xC][yC] is inferred to be equal to 0.

par_level_flag[n] specifies the parity of the transform coefficient level at scanning position n. When par_level_flag[n] is not present, it is inferred to be equal to 0.

rem_abs_gt1_flag[n] specifies whether the syntax element rem_abs_gt2_flag[n] is present for the scanning position n. When rem_abs_gt1_flag[n] is not present, it is inferred to be equal to 0.

rem_abs_gt2_flag[n] specifies whether the syntax element abs_remainder[n] is present for the scanning position n. When rem_abs_gt2_flag[n] is not present, it is inferred to be equal to 0.

abs_remainder[n] is the remaining absolute value of a transform coefficient level that is coded with Golomb-Rice code at the scanning position n. When abs_remainder[n] is not present, it is inferred to be equal to 0.

It is a requirement of bitstream conformance that the value of abs_remainder[n] shall be constrained such that the corresponding value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is in the range of CoeffMin to CoeffMax, inclusive.

coeff_sign_flag[n] specifies the sign of a transform coefficient level for the scanning position n as follows:

– If coeff_sign_flag[n] is equal to 0, the corresponding transform coefficient level has a positive value.

– Otherwise (coeff_sign_flag[n] is equal to 1), the corresponding transform coefficient level has a negative value.

When coeff_sign_flag[n] is not present, it is inferred to be equal to 0.

mts_idx[x0][y0][cIdx] specifies which transform kernels are applied to theresidual samples along the horizontal and vertical direction of the current transform block. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture the array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb and equal to 2 for Cr.

When mts_idx[x0][y0][cIdx] is not present, it is inferred to be equal to −1.

8 Decoding process

8.1 General decoding process

8.2 Decoding process for coding units coded in intra prediction mode

8.2.1 General decoding process for coding units coded in intra prediction mode

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– a variable treeType specifying whether a single or a dual tree is used and if a dual tree is used, it specifies whether the current tree corresponds to the luma or chroma components.

Output of this process is a modified reconstructed picture before in-loop filtering.

The derivation process for quantization parameters as specified in clause 8.4.1 is invoked with the luma location (xCb, yCb), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight, and the variable treeType as inputs.

When treeType is equal to SINGLE_TREE or treeType is equal to DUAL_TREE_LUMA, the decoding process for luma samples is specified as follows:

– If pcm_flag[xCb][yCb] is equal to 1, the reconstructed picture is modified as follows:

$$S_L[xCb + i][yCb + j] = pcm\_sample\_luma[(cbHeight * j) + i] << (BitDepth_Y - PcmBitDepth_Y), \quad \text{with } i = 0..cbWidth - 1, j = 0..cbHeight - 1 \tag{8-1}$$

– Otherwise, the following applies:

1. The derivation process for the luma intra prediction mode as specified in clause 8.2.2 is invoked with the luma location (xCb, yCb), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.
2. The general decoding process for intra blocks as specified in clause 8.2.4.1 is invoked with the luma location (xCb, yCb), the tree type treeType, the variable nTbW set equal to cbWidth, the variable nTbH set equal to cbHeight, the variable predModeIntra set equal to IntraPredModeY[xCb][yCb], and the variable cIdx set equal to 0 as inputs, and the output is a modified reconstructed picture before in-loop filtering.

When treeType is equal to SINGLE_TREE or treeType is equal to DUAL_TREE_CHROMA, the decoding process for chroma samples is specified as follows:

– If pcm_flag[xCb][yCb] is equal to 1, the reconstructed picture is modified as follows:

$$S_{Cb}[xCb / SubWidthC + i][yCb / SubHeightC + j] = pcm\_sample\_chroma[(cbHeight / SubWidthC * j) + i] << (BitDepth_C - PcmBitDepth_C), \text{ with } i = 0..\ cbWidth / SubWidthC - 1 \text{ and } j = 0..\ cbHeight / SubHeightC - 1 \quad (8\text{-}2)$$

$$S_{Cr}[xCb / SubWidthC + i][yCb / SubHeightC + j] = pcm\_sample\_chroma[(cbHeight / SubWidthC * (j + cbHeight / SubHeightC)) + i] << (BitDepth_C - PcmBitDepth_C), \text{ with } i = 0..cbWidth / SubWidthC - 1 \text{ and } j = 0..cbHeight / SubHeightC - 1 \quad (8\text{-}3)$$

– Otherwise, the following applies:

1. The derivation process for the chroma intra prediction mode as specified in clause 8.2.3 is invoked with the luma location (xCb, yCb) , the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.
2. The general decoding process for intra blocks as specified in clause 8.2.4.1 is invoked with the chroma location (xCb / 2, yCb / 2), the tree type treeType, the variable nTbW set equal to (cbWidth / 2), the variable nTbH set equal to (cbHeight / 2), the variable predModeIntra set equal to IntraPredModeC[xCb][yCb], and the variable cIdx set equal to 1, and the output is a modified reconstructed picture before in-loop filtering.
3. The general decoding process for intra blocks as specified in clause 8.2.4.1 is invoked with the chroma location (xCb / 2 , yCb / 2), the tree type treeType, the variable nTbW set equal to (cbWidth / 2), the variable nTbH set equal to (cbHeight / 2), the variable predModeIntra set equal to IntraPredModeC[xCb][yCb], and the variable cIdx set equal to 2, and the output is a modified reconstructed picture before in-loop filtering.

8.2.2 Derivation process for luma intra prediction mode

Input to this process are:

– a luma location (xCb , yCb) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

In this process, the luma intra prediction mode IntraPredModeY[xCb][yCb] is derived.

Table 8-1 specifies the value for the intra prediction mode IntraPredModeY[xCb][yCb] and the associated names.

Table 8-1 – Specification of intra prediction mode and associated names

Intra prediction mode	Associated name
0	INTRA_PLANAR
1	INTRA_DC
2..66	INTRA_ANGULAR2..INTRA_ANGULAR66
81..83	INTRA_LT_CCLM, INTRA_L_CCLM, INTRA_T_CCLM

NOTE – : The intra prediction modes INTRA_LT_CCLM, INTRA_L_CCLM and INTRA_T_CCLM are only applicable to chroma components.

IntraPredModeY[xCb][yCb] is derived by the following ordered steps:

1. The neighbouring locations (xNbA, yNbA) and (xNbB, yNbB) are set equal to (xCb − 1, yCb + cbHeight − 1) and (xCb + cbWidth − 1, yCb − 1), respectively.
2. For X being replaced by either A or B, the variables candIntraPredModeX are derived as follows:
   – The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring location (xNbY, yNbY) set equal to (xNbX, yNbX) as inputs, and the output is assigned to availableX.

– The candidate intra prediction mode candIntraPredModeX is derived as follows:
  – If one or more of the following conditions are true, candIntraPredModeX is set equal to INTRA_PLANAR.
    – The variable availableX is equal to FALSE.
    – CuPredMode[xNbX][yNbX] is not equal to MODE_INTRA.
    – pcm_flag[xNbX][yNbX] is equal to 1.
    – X is equal to B and yCb − 1 is less than ((yCb >> CtbLog2SizeY) << CtbLog2SizeY).
  – Otherwise, candIntraPredModeX is set equal to IntraPredModeY[xNbX][yNbX].

3. The candModeList[x] with x = 0..5 is derived as follows:

– If candIntraPredModeB is equal to candIntraPredModeA and candIntraPredModeA is greater than INTRA_DC, candModeList[x] with x = 0..5 is derived as follows:
  – If IntraLumaRefLineIdx[xCb][yCb] is equal to 0, the following applies:

candModeList[0] = candIntraPredModeA (8-4)

candModeList[1] = INTRA_PLANAR (8-5)

candModeList[2] = INTRA_DC (8-6)

candModeList[3] = 2 + ((candIntraPredModeA + 61) % 64) (8-7)

candModeList[4] = 2 + ((candIntraPredModeA − 1) % 64) (8-8)

candModeList[5] = 2 + ((candIntraPredModeA + 60) % 64) (8-9)

  – Otherwise (IntraLumaRefLineIdx[xCb][yCb] is not equal to 0), the following applies:

candModeList[0] = candIntraPredModeA (8-10)

candModeList[1] = 2 + ((candIntraPredModeA + 61) % 64) (8-11)

candModeList[2] = 2 + ((candIntraPredModeA − 1) % 64) (8-12)

candModeList[3] = 2 + ((candIntraPredModeA + 60) % 64) (8-13)

candModeList[4] = 2 + (candIntraPredModeA % 64) (8-14)

candModeList[5] = 2 + ((candIntraPredModeA + 59) % 64) (8-15)

– Otherwise if candIntraPredModeB is not equal to candIntraPredModeA and candIntraPredModeA or candIntraPredModeB is greater than INTRA_DC, the following applies:
  – The variables minAB and maxAB are derived as follows:

minAB = candModeList[(candModeList[0] > candModeList[1]) ? 1 : 0] (8-16)

maxAB = candModeList[(candModeList[0] > candModeList[1]) ? 0 : 1] (8-17)

  – If candIntraPredModeA and candIntraPredModeB are both greater than INTRA_DC, candModeList[x] with x = 0..5 is derived as follows:

candModeList[0] = candIntraPredModeA (8-18)

candModeList[1] = candIntraPredModeB (8-19)

    – If IntraLumaRefLineIdx[xCb][yCb] is equal to 0, the following applies:

candModeList[2] = INTRA_PLANAR (8-20)

candModeList[3] = INTRA_DC (8-21)

– If maxAB − minAB is in the range of 2 to 62, inclusive, the following applies:

candModeList[4] = 2 + ((maxAB + 61) % 64) (8-22)

candModeList[5] = 2 + ((maxAB − 1) % 64) (8-23)

– Otherwise, the following applies:

candModeList[4] = 2 + ((maxAB + 60) % 64) (8-24)

candModeList[5] = 2 + ((maxAB) % 64) (8-25)

– Otherwise (IntraLumaRefLineIdx[xCb][yCb] is not equal to 0), the following applies:

– If maxAB − minAB is equal to 1, the following applies:

candModeList[2] = 2 + ((minAB + 61) % 64) (8-26)

candModeList[3] = 2 + ((maxAB − 1) % 64) (8-27)

candModeList[4] = 2 + ((minAB + 60) % 64) (8-28)

candModeList[5] = 2 + (maxAB % 64) (8-29)

– Otherwise if maxAB − minAB is equal to 2, the following applies:

candModeList[2] = 2 + ((minAB − 1) % 64) (8-30)

candModeList[3] = 2 + ((minAB + 61) % 64) (8-31)

candModeList[4] = 2 + ((maxAB − 1) % 64) (8-32)

candModeList[5] = 2 + ((minAB + 60) % 64) (8-33)

– Otherwise if maxAB − minAB is greater than 61, the following applies:

candModeList[2] = 2 + ((minAB − 1) % 64) (8-34)

candModeList[3] = 2 + ((maxAB + 61) % 64) (8-35)

candModeList[4] = 2 + (minAB % 64) (8-36)

candModeList[5] = 2 + ((maxAB + 60) % 64) (8-37)

– Otherwise, the following applies:

candModeList[2] = 2 + ((minAB + 61) % 64) (8-38)

candModeList[3] = 2 + ((minAB − 1) % 64) (8-39)

candModeList[4] = 2 + ((maxAB + 61) % 64) (8-40)

candModeList[5] = 2 + ((maxAB − 1) % 64) (8-41)

– Otherwise (candIntraPredModeA or candIntraPredModeB is greater than INTRA_DC), candModeList[x] with x = 0..5 is derived as follows:

– If IntraLumaRefLineIdx[xCb][yCb] is equal to 0, the following applies:

candModeList[0] = candIntraPredModeA (8-42)

candModeList[1] = candIntraPredModeB (8-43)

candModeList[2] = 1 − minAB (8-44)

candModeList[3] = 2 + ((maxAB + 61) % 64) (8-45)

candModeList[4] = 2 + ((maxAB − 1) % 64) (8-46)

candModeList[5] = 2 + ((maxAB + 60) % 64) (8-47)

– Otherwise (IntraLumaRefLineIdx[xCb][yCb] is not equal to 0), the following applies:

candModeList[0] = maxAB (8-48)

candModeList[1] = 2 + ((maxAB + 61) % 64) (8-49)

candModeList[2] = 2 + ((maxAB − 1) % 64) (8-50)

candModeList[3] = 2 + ((maxAB + 60) % 64) (8-51)

candModeList[4] = 2 + (maxAB % 64) (8-52)

candModeList[5] = 2 + ((maxAB + 59) % 64) (8-53)

– Otherwise, the following applies:

– If IntraLumaRefLineIdx[xCb][yCb] is equal to 0, the following applies:

candModeList[0] = candIntraPredModeA (8-54)

candModeList[1] = (candModeList[0] == INTRA_PLANAR) ? INTRA_DC : INTRA_PLANAR (8-55)

candModeList[2] = INTRA_ANGULAR50 (8-56)

candModeList[3] = INTRA_ANGULAR18 (8-57)

candModeList[4] = INTRA_ANGULAR46 (8-58)

candModeList[5] = INTRA_ANGULAR54 (8-59)

– Otherwise (IntraLumaRefLineIdx[xCb][yCb] is not equal to 0), the following applies:

candModeList[0] = INTRA_ANGULAR50 (8-60)

candModeList[1] = INTRA_ANGULAR18 (8-61)

candModeList[2] = INTRA_ANGULAR2 (8-62)

candModeList[3] = INTRA_ANGULAR34 (8-63)

candModeList[4] = INTRA_ANGULAR66 (8-64)

candModeList[5] = INTRA_ANGULAR26 (8-65)

4. IntraPredModeY[xCb][yCb] is derived by applying the following procedure:

– If intra_luma_mpm_flag[xCb][yCb] is equal to 1, the IntraPredModeY[xCb][yCb] is set equal to candModeList[intra_luma_mpm_idx[xCb][yCb]].

– Otherwise, IntraPredModeY[xCb][yCb] is derived by applying the following ordered steps:

1. When candModeList[i] is greater than candModeList[j] for i = 0..4 and for each i, j = (i + 1)..5, both values are swapped as follows:

(candModeList[i], candModeList[j]) = Swap(candModeList[i], candModeList[j]) (8-66)

2. IntraPredModeY[xCb][yCb] is derived by the following ordered steps:
   i. IntraPredModeY[xCb][yCb] is set equal to intra_luma_mpm_remainder[xCb][yCb].
   ii. For i equal to 0 to 5, inclusive, when IntraPredModeY[xCb][yCb] is greater than or equal to candModeList[i], the value of IntraPredModeY[xCb][yCb] is incremented by one.

The variable IntraPredModeY[x][y] with x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1 is set to be equal to IntraPredModeY[xCb][yCb].

8.2.3 Derivation process for chroma intra prediction mode

Input to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current chroma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

In this process, the chroma intra prediction mode IntraPredModeC[xCb][yCb] is derived.

The chroma intra prediction mode IntraPredModeC[xCb][yCb] is derived using intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb + cbWidth / 2][yCb + cbHeight / 2] as specified in Table 8-2 and Table 8-3.

Table 8-2 – Specification of IntraPredModeC[xCb][yCb] depending on intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb + cbWidth / 2][yCb + cbHeight / 2] when sps_cclm_enabled_flag is equal to 0

intra_chroma_pred_mode[xCb][yCb]	IntraPredModeY[xCb + cbWidth / 2][yCb + cbHeight / 2]				
	0	50	18	1	X (0 <= X <= 66)
0	66	0	0	0	0
1	50	66	50	50	50
2	18	18	66	18	18
3	1	1	1	66	1
4	0	50	18	1	X

Table 8-3 – Specification of IntraPredModeC[xCb][yCb] depending on intra_chroma_pred_mode[xCb][yCb] and IntraPredModeY[xCb + cbWidth / 2][yCb + cbHeight / 2] when sps_cclm_enabled_flag is equal to 1

intra_chroma_pred_mode[xCb][yCb]	IntraPredModeY[xCb + cbWidth / 2][yCb + cbHeight / 2]				
	0	50	18	1	X (0 <= X <= 66)
0	66	0	0	0	0
1	50	66	50	50	50
2	18	18	66	18	18
3	1	1	1	66	1
4	81	81	81	81	81
5	82	82	82	82	82
6	83	83	83	83	83
7	0	50	18	1	X

8.2.4 Decoding process for intra blocks

8.2.4.1 General decoding process for intra blocks

Inputs to this process are:

– a sample location (xTb0, yTb0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable nTbW specifying the width of the current transform block,

– a variable nTbH specifying the height of the current transform block,

– a variable predModeIntra specifying the intra prediction mode,

– a variable cIdx specifying the colour component of the current block.

Output of this process is a modified reconstructed picture before in-loop filtering.

The maximum transform block size maxTbSize is derived as follows:

$$maxTbSize = (cIdx == 0) ? MaxTbSizeY : MaxTbSizeY / 2 \quad (8\text{-}67)$$

The luma sample location is derived as follows:

$$(xTbY, yTbY) = (cIdx == 0) ? (xTb0, yTb0) : (xTb0 * 2, yTb0 * 2) \quad (8\text{-}68)$$

Depending on maxTbSize, the following applies:

– If nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables newTbW and newTbH are derived as follows:

$$newTbW = (nTbW > maxTbSize) ? (nTbW / 2) : nTbW \quad (8\text{-}69)$$

$$newTbH = (nTbH > maxTbSize) ? (nTbH / 2) : nTbH \quad (8\text{-}70)$$

2. The general decoding process for intra blocks as specified in this clause is invoked with the location (xTb0, yTb0), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

3. If nTbW is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

4. If nTbH is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0, yTb0 + newTbH), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

5. If nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0 + newTbH), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

– Otherwise, the following ordered steps apply:

1. The general intra sample prediction process as specified in clause 8.2.4.2.1 is invoked with the location (xTbCmp, yTbCmp) set equal to (xTb0, yTb0), the intra prediction mode predModeIntra, the transform block width nTbW and height nTbH, and the variable cIdx as inputs, and the output is an (nTbW)x(nTbH) array predSamples.

2. The scaling and transformation process as specified in clause 8.4.2 is invoked with the luma location (xTbY, yTbY), the variable cIdx, the transform width nTbW and the transform height nTbH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.

3. The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the transform block location (xTbComp, yTbComp) set equal to (xTb0, yTb0), the transform block width nTbW, the transform block height nTbH, the variable cIdx, the (nTbW)x(nTbH) array predSamples, and the (nTbW)x(nTbH) array resSamples as inputs, and the output is a modified reconstructed picture before in-loop filtering.

8.2.4.2 Intra sample prediction

8.2.4.2.1 General intra sample prediction

Inputs to this process are:

– a sample location (xTbCmp, yTbCmp) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable predModeIntra specifying the intra prediction mode,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable cIdx specifying the colour component of the current block.

Outputs of this process are the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables refW and refH are derived as follows:

$$refW = nTbW * 2 \tag{8-71}$$

$$refH = nTbH * 2 \tag{8-72}$$

The variable refIdx specifying the intra prediction reference line index is derived as follows:

$$refIdx = (cIdx == 0) \ ? \ IntraLumaRefLineIdx[xTbCmp][yTbCmp] \ : \ 0 \tag{8-73}$$

For the generation of the reference samples p[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, the following ordered steps apply:

1. The reference sample availability marking process as specified in clause 8.2.4.2.2 is invoked with the sample location (xTbCmp, yTbCmp), the intra prediction reference line index refIdx, the reference sample width refW, the reference sample height refH, the colour component index cIdx as inputs, and the reference samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = − refIdx..refW − 1, y = −1 − refIdx as output.
2. When at least one sample refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx is marked as "not available for intra prediction", the reference sample substitution process as specified in clause 8.2.4.2.3 is invoked with the intra prediction reference line index refIdx, the reference sample width refW, the reference sample height refH, the reference samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, and the colour component index cIdx as inputs, and the modified reference samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx as output.
3. The reference sample filtering process as specified in clause 8.2.4.2.4 is invoked with the intra prediction reference line index refIdx, the transform block width nTbW and height nTbH, the reference sample width refW, the reference sample height refH, the unfiltered samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, and the colour component index cIdx as inputs, and the reference samples p[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx as output.

The intra sample prediction process according to predModeIntra applies as follows:

– If predModeIntra is equal to INTRA_PLANAR, the corresponding intra prediction mode process specified in clause 8.2.4.2.5 is invoked with the transform block width nTbW, and the transform block height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_DC, the corresponding intra prediction mode process specified in clause 8.2.4.2.6 is invoked with the transform block width nTbW, the transform block height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.

– Otherwise, if predModeIntra is equal to INTRA_LT_CCLM, INTRA_L_CCLM or INTRA_T_CCLM, the corresponding intra prediction mode process specified in clause 8.2.4.2.8 is invoked with the intra prediction mode predModeIntra, the sample location (xTbC, yTbC) set equal to (xTbCmp, yTbCmp), the transform block width nTbW and height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.

– Otherwise, the corresponding intra prediction mode process specified in clause 8.2.4.2.7 is invoked with the intra prediction mode predModeIntra, the intra prediction reference line index refIdx, the transform block width nTbW, the transform block height nTbH, the reference sample width refW, the reference sample height refH, the colour component index cIdx, and the reference sample array p as inputs, and the modified intra prediction mode predModeIntra and the predicted sample array predSamples as outputs.

When refIdx is equal to 0 and one of the following conditions is true, the position-dependent prediction sample filtering process specified in clause 8.2.4.2.9 is invoked with the intra prediction mode predModeIntra, the transform block width nTbW, the transform block height nTbH, the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1, the reference sample width refW, the reference sample height refH, the reference samples p[x][y], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, and the colour component index cIdx as inputs, and the output is the modified predicted sample array predSamples:

– predModeIntra is equal to INTRA_PLANAR

– predModeIntra is equal to INTRA_DC

– predModeIntra is equal to INTRA_ANGULAR18

– predModeIntra is equal to INTRA_ANGULAR50

– predModeIntra is less than or equal to INTRA_ANGULAR10

– predModeIntra is greater than or equal to INTRA_ANGULAR58

8.2.4.2.2 Reference sample availability marking process

Inputs to this process are:

– a sample location (xTbCmp, yTbCmp) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable refIdx specifying the intra prediction reference line index,

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

– a variable cIdx specifying the colour component of the current block.

Outputs of this process are the reference samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction.

The refW + refH + 1 + (2 * refIdx) neighbouring samples refUnfilt[x][y] that are constructed samples prior to the in-loop filter process, with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, are derived as follows:

– The neighbouring location (xNbCmp, yNbCmp) is specified by:

$$(xNbCmp, yNbCmp) = (xTbCmp + x, yTbCmp + y) \quad (8\text{-}74)$$

– The current luma location (xTbY, yTbY) and the neighbouring luma location (xNbY, yNbY) are derived as follows:

$$(xTbY, yTbY) = (cIdx == 0) ? (xTbCmp, yTbCmp) : (xTbCmp << 1, yTbCmp << 1) \quad (8\text{-}75)$$

$$(xNbY, yNbY) = (cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp << 1, yNbCmp << 1) \quad (8\text{-}76)$$

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xTbY, yTbY) and the neighbouring luma location (xNbY, yNbY) as inputs, and the output is assigned to availableN.

– Each sample refUnfilt[x][y] is derived as follows:

  – If availableN is equal to FALSE, the sample refUnfilt[x][y] is marked as "not available for intra prediction".

  – Otherwise, the sample refUnfilt[x][y] is marked as "available for intra prediction" and the sample at the location (xNbCmp, yNbCmp) is assigned to refUnfilt[x][y].

8.2.4.2.3 Reference sample substitution process

Inputs to this process are:

– a variable refIdx specifying the intra prediction reference line index,

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

– reference samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction,

– a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified reference samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction.

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to $BitDepth_Y$.

– Otherwise, bitDepth is set equal to $BitDepth_C$.

The values of the samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are modified as follows:

– If all samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are marked as "not available for intra prediction", all values of refUnfilt[x][y] are set equal to 1 << (bitDepth − 1).

– Otherwise (at least one but not all samples refUnfilt[x][y] are marked as "not available for intra prediction"), the following ordered steps apply:

  1. When refUnfilt[−1 − refIdx][refH − 1] is marked as "not available for intra prediction", search sequentially starting from x = −1 − refIdx, y = refH − 1 to x = −1 − refIdx, y = −1 − refIdx, then from x = −refIdx, y = −1 − refIdx to x = refW − 1, y = −1 − refIdx, for a sample refUnfilt[x][y] that is marked as "available for intra prediction". Once a sample refUnfilt[x][y] marked as "available for intra prediction" is found, the search is terminated and the value of refUnfilt[−1 − refIdx][refH − 1] is set equal to the value of refUnfilt[x][y].

  2. For x = −1 − refIdx, y = refH − 2..−1 − refIdx, when refUnfilt[x][y] is marked as "not available for intra prediction", the value of refUnfilt[x][y] is set equal to the value of refUnfilt[x][y + 1].

  3. For x = 0..refW − 1, y = −1, when refUnfilt[x][y] is marked as "not available for intra prediction", the value of refUnfilt[x][y] is set equal to the value of refUnfilt[x − 1][y].

All samples refUnfilt[x][y] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are marked as "available for intra prediction".

8.2.4.2.4 Reference sample filtering process

Inputs to this process are:

– a variable refIdx specifying the intra prediction reference line index,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

– the (unfiltered) neighbouring samples refUnfilt[x][y], with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx,

– a variable cIdx specifying the colour component of the current block.

Outputs of this process are the reference samples p[x][y], with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx.

The variable filterFlag is derived as follows:

– If all of the following conditions are true, filterFlag is set equal to 1:
  – refIdx is equal to 0
  – nTbW * nTbH is greater than 32
  – cIdx is equal to 0
  – one or more of the following conditions is true:
    – predModeIntra is equal to INTRA_PLANAR
    – predModeIntra is equal to INTRA_ANGULAR34
    – predModeIntra is equal to INTRA_ANGULAR2 and nTbH is greater than or equal to nTbW
    – predModeIntra is equal to INTRA_ANGULAR66 and nTbW is greater than or equal to nTbH
– Otherwise, filterFlag is set equal to 0.

For the derivation of the reference samples p[x][y] the following applies:

– If filterFlag is equal to 1, the filtered sample values p[x][y] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are derived as follows:

$$p[-1][-1] = (refUnfilt[-1][0] + 2 * refUnfilt[-1][-1] + refUnfilt[0][-1] + 2) >> 2 \quad (8\text{-}77)$$

$$p[-1][y] = (refUnfilt[-1][y + 1] + 2 * refUnfilt[-1][y] + refUnfilt[-1][y - 1] + 2) >> 2 \quad \text{for } y = 0..refH - 2 \quad (8\text{-}78)$$

$$p[-1][refH - 1] = refUnfilt[-1][refH - 1] \quad (8\text{-}79)$$

$$p[x][-1] = (refUnfilt[x - 1][-1] + 2 * refUnfilt[x][-1] + refUnfilt[x + 1][-1] + 2) >> 2 \quad \text{for } x = 0..refW - 2 \quad (8\text{-}80)$$

$$p[refW - 1][-1] = refUnfilt[refW - 1][-1] \quad (8\text{-}81)$$

– Otherwise, the reference samples values p[x][y] are set equal to the unfiltered sample values refUnfilt[x][y] with x = −1− refIdx, y = −1− refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1− refIdx.

8.2.4.2.5 Specification of INTRA_PLANAR intra prediction mode

Inputs to this process are:

– a variable nTbW specifying the transform block width,
– a variable nTbH specifying the transform block height,
– the neighbouring samples p[x][y], with x = −1, y = −1..nTbH and x = 0..nTbW, y = −1.

Outputs of this process are the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The values of the prediction samples predSamples[x][y], with x = 0..nTbW − 1 and y = 0..nTbH − 1, are derived as follows:

$$predV[x][y] = ((nTbH - 1 - y) * p[x][-1] + (y + 1) * p[-1][nTbH]) << Log2 (nTbW) \quad (8\text{-}82)$$

$$predH[x][y] = ((nTbW - 1 - x) * p[-1][y] + (x + 1) * p[nTbW][-1]) << Log2 (nTbH) \quad (8\text{-}83)$$

$$predSamples[x][y] = (predV[x][y] + predH[x][y] + nTbW * nTbH) >> (Log2 (nTbW) + Log2 (nTbH) + 1) \quad (8\text{-}84)$$

8.2.4.2.6 Specification of INTRA_DC intra prediction mode

Inputs to this process are:

– a variable nTbW specifying the transform block width,
– a variable nTbH specifying the transform block height,
– the neighbouring samples p[x][y], with x = −1, y = −1..nTbH − 1 and x = 0..nTbW − 1 , y = −1.

Outputs of this process are the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The values of the prediction samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1, are derived by the following ordered steps:

1. A variable dcVal is derived as follows:
   – When nTbW is equal to nTbH:

$$\text{dcVal} = \left(\sum_{x'=0}^{\text{nTbW}-1} \text{p}[x'][-1] + \sum_{y'=0}^{nTbH-1} \text{p}[-1][y'] + \text{nTbW}\right) \gg (\text{Log2}(\text{nTbW}) + 1) \quad (8\text{-}85)$$

   – When nTbW is greater than nTbH:

$$\text{dcVal} = \left(\sum_{x'=0}^{\text{nTbW}-1} \text{p}[x'][-1] + (\text{nTbW} \gg 1)\right) \gg \text{Log2}(\text{nTbW}) \quad (8\text{-}86)$$

   – When nTbW is less than nTbH:

$$\text{dcVal} = \left(\sum_{y'=0}^{\text{nTbH}-1} \text{p}[-1][y'] + (\text{nTbH} \gg 1)\right) \gg \text{Log2}(\text{nTbH}) \quad (8\text{-}87)$$

2. The prediction samples predSamples[x][y] are derived as follows:

$$\text{predSamples}[\text{ x }][\text{ y }] = \text{dcVal, with x} = 0..\text{ nTbW} - 1, \text{ y} = 0..\text{ nTbH} - 1 \quad (8\text{-}88)$$

8.2.4.2.7 Specification of INTRA_ANGULAR2..INTRA_ANGULAR66 intra prediction modes

Inputs to this process are:

– the intra prediction mode predModeIntra,

– a variable refIdx specifying the intra prediction reference line index,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable refW specifying the reference samples width,

– a variable refH specifying the reference samples height,

– a variable cIdx specifying the colour component of the current block,

– the neighbouring samples p[x][y], with x = −1− refIdx, y = −1− refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1− refIdx.

Outputs of this process are the modified intra prediction mode predModeIntra and the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable nTbS is set equal to (Log2 (nTbW) + Log2 (nTbH)) >> 1.

The variable whRatio is set equal to Abs(Log2(nTbW / nTbH)).

The variable wideAngle is set equal to 0.

For non-square blocks (nTbW is not equal to nTbH), the intra prediction mode predModeIntra is modified as follows:

– If all of the following conditions are true, wideAngle is set equal to 1 and predModeIntra is set equal to (predModeIntra + 65).
  – nTbW is greater than nTbH
  – predModeIntra is greater than or equal to 2
  – predModeIntra is less than (whRatio > 1) ? (8 + 2 * whRatio) : 8

– Otherwise, if all of the following conditions are true, wideAngle is set equal to 1 and predModeIntra is set equal to (predModeIntra − 67).
  – nTbH is greater than nTbW
  – predModeIntra is less than or equal to 66
  – predModeIntra is greater than (whRatio > 1) ? (60 − 2 * whRatio) : 60

The variable filterFlag is derived as follows:

– If predModeIntra is equal to INTRA_ANGULAR2, INTRA_ANGULAR34 or INTRA_ANGULAR66 or refIdx is not equal to 0, filterFlag is set equal to 0.

– Otherwise, the following applies:

  – The variable minDistVerHor is set equal to Min(Abs(predModeIntra − 50), Abs(predModeIntra − 18)).

  – The variable intraHorVerDistThres[nTbS] is specified in .

  – The variable filterFlag is derived as follows:

    – If minDistVerHor is greater than intraHorVerDistThres[nTbS] or wideAngle is equal to 1, filterFlag is set equal to 1.

    – Otherwise, filterFlag is set equal to 0.

Table 8-4 – Specification of intraHorVerDistThres[nTbS] for various transform block sizes nTbS

	nTbS = 2	nTbS = 3	nTbS = 4	nTbS = 5	nTbS = 6	nTbS = 7
intraHorVerDistThres[nTbS]	20	14	2	0	0	0

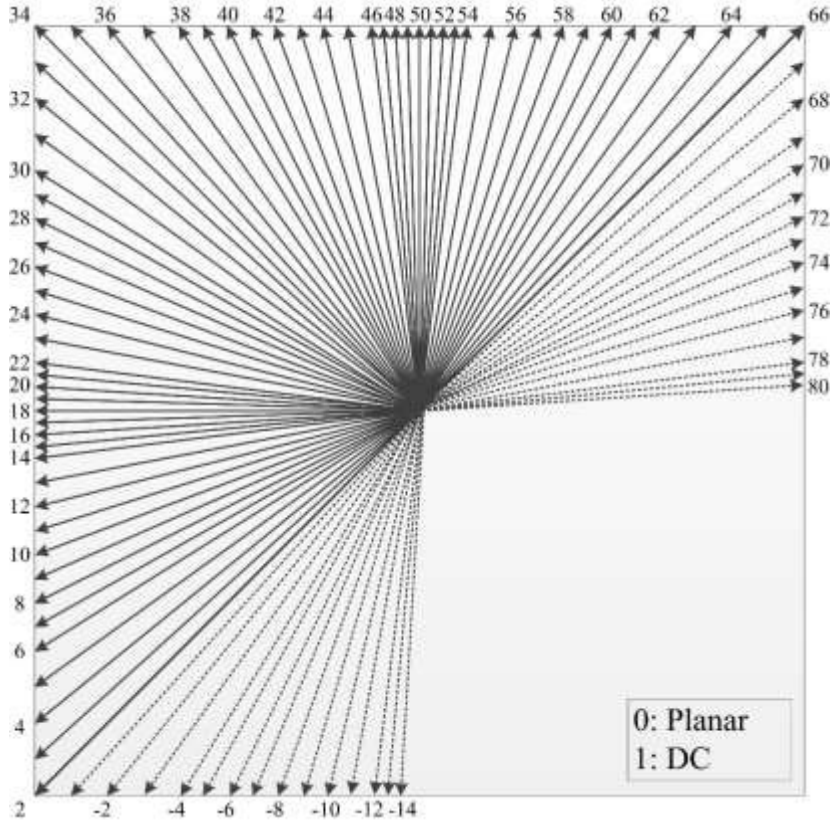


Figure 8-1 – Intra prediction directions (informative)

Figure 8-1 illustrates the 93 prediction directions, where the dashed directions are associated with the wide-angle modes that are only applied to non-square blocks.

Table 8-5 specifies the mapping table between predModeIntra and the angle parameter intraPredAngle.

Table 8-5 – Specification of intraPredAngle

predModeIntra	**−14**	**−13**	**−12**	**−11**	**−10**	**−9**	**−8**	**−7**	**−6**	**−5**	**−4**	**−3**	**−2**	**−1**	**2**	**3**	**4**
intraPredAngle	512	341	256	171	128	102	86	73	64	57	51	45	39	35	32	29	26
predModeIntra	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
intraPredAngle	23	20	18	16	14	12	10	8	6	4	3	2	1	0	−1	−2	−3
predModeIntra	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**
intraPredAngle	−4	−6	−8	−10	−12	−14	−16	−18	−20	−23	−26	−29	−32	−29	−26	−23	−20
predModeIntra	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**
intraPredAngle	−18	−16	−14	−12	−10	−8	−6	−4	−3	−2	−1	0	1	2	3	4	6
predModeIntra	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**
intraPredAngle	8	10	12	14	16	18	20	23	26	29	32	35	39	45	51	57	64
predModeIntra	**73**	**74**	**75**	**76**	**77**	**78**	**79**	**80**									
intraPredAngle	73	86	102	128	171	256	341	512									

The inverse angle parameter invAngle is derived based on intraPredAngle as follows:

$$\text{invAngle} = \text{Round}\left(\frac{256*32}{\text{intraPredAngle}}\right) \tag{8-89}$$

The interpolation filter coefficients fC[phase][j] and fG[phase][j] with phase = 0..31 and j = 0..3 are specified in Table 8-6.

Table 8-6 – Specification of interpolation filter coefficients fC and fG

Fractional sample position p	fC interpolation filter coefficients				fG interpolation filter coefficients			
	f_C[p][0]	f_C[p][1]	f_C[p][2]	f_C[p][3]	fG[p][0]	fG[p][1]	fG[p][2]	fG[p][3]
0	0	64	0	0	16	32	16	0
1	−1	63	2	0	15	29	17	3
2	−2	62	4	0	15	29	17	3
3	−2	60	7	−1	14	29	18	3
4	−2	58	10	−2	13	29	18	4
5	−3	57	12	−2	13	28	19	4
6	−4	56	14	−2	13	28	19	4
7	−4	55	15	−2	12	28	20	4
8	−4	54	16	−2	11	28	20	5
9	−5	53	18	−2	11	27	21	5
10	−6	52	20	−2	10	27	22	5
11	−6	49	24	−3	9	27	22	6
12	−6	46	28	−4	9	26	23	6
13	−5	44	29	−4	9	26	23	6
14	−4	42	30	−4	8	25	24	7
15	−4	39	33	−4	8	25	24	7
16	−4	36	36	−4	8	24	24	8
17	−4	33	39	−4	7	24	25	8
18	−4	30	42	−4	7	24	25	8
19	−4	29	44	−5	6	23	26	9
20	−4	28	46	−6	6	23	26	9
21	−3	24	49	−6	6	22	27	9
22	−2	20	52	−6	5	22	27	10
23	−2	18	53	−5	5	21	27	11
24	−2	16	54	−4	5	20	28	11
25	−2	15	55	−4	4	20	28	12
26	−2	14	56	−4	4	19	28	13
27	−2	12	57	−3	4	19	28	13
28	−2	10	58	−2	4	18	29	13
29	−1	7	60	−2	3	18	29	14
30	0	4	62	−2	3	17	29	15
31	0	2	63	−1	3	17	29	15

The values of the prediction samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

– If predModeIntra is greater than or equal to 34, the following ordered steps apply:

1. The reference sample array ref[x] is specified as follows:

   – The following applies:

$$ref[x] = p[-1 - refIdx + x][-1 - refIdx], \text{ with } x = 0..nTbW + refIdx \quad (8\text{-}90)$$

   – If intraPredAngle is less than 0, the main reference sample array is extended as follows:

     – When (nTbH * intraPredAngle) >> 5 is less than −1,

$$ref[x] = p[-1 - refIdx][-1 - refIdx + ((x * invAngle + 128) >> 8)], \text{ with } x = -1..(nTbH * intraPredAngle) >> 5 \quad (8\text{-}91)$$

$$ref[((nTbH * intraPredAngle) >> 5) - 1] = ref[(nTbH * intraPredAngle) >> 5] \quad (8\text{-}92)$$

$$ref[nTbW + 1 + refIdx] = ref[nTbW + refIdx] \quad (8\text{-}93)$$

– Otherwise,

$$ref[x] = p[-1 - refIdx + x][-1 - refIdx], \text{ with } x = nTbW + 1 + refIdx..refW + refIdx \quad (8\text{-}94)$$

$$ref[-1] = ref[0] \quad (8\text{-}95)$$

– The additional samples ref[refW + refIdx +x] with x = 1..(Max(1, nTbW / nTbH) * refIdx + 1) are derived as follows:

$$ref[refW + refIdx + x] = p[-1 + refW][-1 - refIdx] \quad (8\text{-}96)$$

2. The values of the prediction samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

– The index variable iIdx and the multiplication factor iFact are derived as follows:

$$iIdx = ((y + 1 + refIdx) * intraPredAngle) >> 5 + refIdx \quad (8\text{-}97)$$

$$iFact = ((y + 1 + refIdx) * intraPredAngle) \& 31 \quad (8\text{-}98)$$

– If cIdx is equal to 0, the following applies:

– The interpolation filter coefficients fT[j] with j = 0..3 are derived as follows:

$$fT[j] = filterFlag \ ? \ fG[iFact][j] \ : \ fC[iFact][j] \quad (8\text{-}99)$$

– The value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = Clip1Y(((\sum_{i=0}^{3} fT[i] * ref[x + iIdx + i]) + 32) >> 6) \quad (8\text{-}100)$$

– Otherwise (cIdx is not equal to 0), depending on the value of iFact, the following applies:

– If iFact is not equal to 0, the value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = ((32 - iFact) * ref[x + iIdx + 1] + iFact * ref[x + iIdx + 2] + 16) >> 5 \quad (8\text{-}101)$$

– Otherwise, the value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = ref[x + iIdx + 1] \quad (8\text{-}102)$$

– Otherwise (predModeIntra is less than 34), the following ordered steps apply:

1. The reference sample array ref[x] is specified as follows:

– The following applies:

$$ref[x] = p[-1 - refIdx][-1 - refIdx + x], \text{ with } x = 0..nTbH + refIdx \quad (8\text{-}103)$$

– If intraPredAngle is less than 0, the main reference sample array is extended as follows:

– When (nTbW * intraPredAngle) >> 5 is less than −1,

$$ref[x] = p[-1 - refIdx + ((x * invAngle + 128) >> 8)][-1 - refIdx], \text{ with } x = -1..(nTbW * intraPredAngle) >> 5 \quad (8\text{-}104)$$

$$ref[((nTbW * intraPredAngle) >> 5) - 1] = ref[(nTbW * intraPredAngle) >> 5] \quad (8\text{-}105)$$

$$ref[nTbG + 1 + refIdx] = ref[nTbH + refIdx] \quad (8\text{-}106)$$

– Otherwise,

$$ref[x] = p[-1 - refIdx][-1 - refIdx + x], \text{ with } x = nTbH + 1 + refIdx..refH + refIdx \quad (8\text{-}107)$$

$$ref[-1] = ref[0] \quad (8\text{-}108)$$

– The additional samples ref[refH + refIdx +x] with x = 1..(Max(1, nTbW / nTbH) * refIdx + 1) are derived as follows:

$$ref[refH + refIdx + x] = p[-1 + refH][-1 - refIdx] \quad (8\text{-}109)$$

2. The values of the prediction samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

– The index variable iIdx and the multiplication factor iFact are derived as follows:

$$iIdx = ((x + 1 + refIdx) * intraPredAngle) >> 5 \quad (8\text{-}110)$$

$$iFact = ((x + 1 + refIdx) * intraPredAngle) \& 31 \quad (8\text{-}111)$$

– If cIdx is equal to 0, the following applies:

– The interpolation filter coefficients fT[j] with j = 0..3 are derived as follows:

$$fT[j] = filterFlag \ ? \ fG[iFact][j] \ : \ fC[iFact][j] \quad (8\text{-}112)$$

– The value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = Clip1Y(((\sum_{i=0}^{3} fT[i] * ref[y + iIdx + i]) + 32) >> 6) \quad (8\text{-}113)$$

– Otherwise (cIdx is not equal to 0), depending on the value of iFact, the following applies:

– If iFact is not equal to 0, the value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = ((32 - iFact) * ref[y + iIdx + 1] + iFact * ref[y + iIdx + 2] + 16) >> 5 \quad (8\text{-}114)$$

– Otherwise, the value of the prediction samples predSamples[x][y] is derived as follows:

$$predSamples[x][y] = ref[y + iIdx + 1] \quad (8\text{-}115)$$

8.2.4.2.8 Specification of INTRA_LT_CCLM, INTRA_L_CCLM and INTRA_T_CCLM intra prediction mode

Inputs to this process are:

– the intra prediction mode predModeIntra,

– a sample location (xTbC, yTbC) of the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– chroma neighbouring samples p[x][y], with x = −1, y = 0..2 * nTbH − 1 and x = 0.. 2 * nTbW − 1, y = − 1.

Output of this process are predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The current luma location (xTbY, yTbY) is derived as follows:

$$(xTbY, yTbY) = (xTbC << 1, yTbC << 1) \quad (8\text{-}116)$$

The variables availL, availT and availTL are derived as follows:

– The availability of left neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location (xCurr, yCurr) set equal to (xTbC, yTbC) and the neighbouring chroma location (xTbC − 1, yTbC) as inputs, and the output is assigned to availL.

– The availability of top neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location (xCurr, yCurr) set equal to (xTbC, yTbC) and the neighbouring chroma location (xTbC, yTbC − 1) as inputs, and the output is assigned to availT.

– The availability of top-left neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location (xCurr, yCurr) set equal to (xTbC, yTbC) and the neighbouring chroma location (xTbC − 1, yTbC − 1) as inputs, and the output is assigned to availTL.

– The number of available top-right neighbouring chroma samples numTopRight is derived as follows:

  – The variable numTopRight is set equal to 0 and availTR is set equal to TRUE.

  – When predModeIntra is equal to INTRA_T_CCLM, the following applies for x = nTbW..2 * nTbW − 1 until availTR is equal to FALSE or x is equal to 2 * nTbW − 1:

    – The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location (xCurr, yCurr) set equal to (xTbC , yTbC) and the neighbouring chroma location (xTbC + x, yTbC − 1) as inputs, and the output is assigned to availableTR

    – When availableTR is equal to TRUE, numTopRight is incremented by one.

– The number of available left-below neighbouring chroma samples numLeftBelow is derived as follows:

  – The variable numLeftBelow is set equal to 0 and availLB is set equal to TRUE.

  – When predModeIntra is equal to INTRA_L_CCLM, the following applies for y = nTbH..2 * nTbH − 1 until availLB is equal to FALSE or y is equal to 2 * nTbH − 1:

    – The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location (xCurr, yCurr) set equal to (xTbC , yTbC) and the neighbouring chroma location (xTbC − 1, yTbC + y) as inputs, and the output is assigned to availableLB

    – When availableLB is equal to TRUE, numLeftBelow is incremented by one.

The number of available neighbouring chroma samples on the top and top-right numTopSamp and the number of available neighbouring chroma samples on the left and left-below nLeftSamp are derived as follows:

– If predModeIntra is equal to INTRA_LT_CCLM, the following applies:

$$numSampT = availT\ ?\ nTbW\ :\ 0 \tag{8-117}$$

$$numSampL = availL\ ?\ nTbH\ :\ 0 \tag{8-118}$$

– Otherwise, the following applies:

$$numSampT = (availT\ \&\&\ predModeIntra == INTRA\_T\_CCLM)\ ?\ (nTbW + numTopRight)\ :\ 0 \tag{8-119}$$

$$numSampL = (availL\ \&\&\ predModeIntra == INTRA\_L\_CCLM)\ ?\ (nTbH + numLeftBelow)\ :\ 0 \tag{8-120}$$

The variable bCTUboundary is derived as follows:

$$bCTUboundary = (yTbC\ \&\ (1 << (CtbLog2SizeY - 1) - 1) == 0)\ ?\ TRUE\ :\ FALSE. \tag{8-121}$$

The prediction samples predSamples[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

– If both numSampL and numSampT are equal to 0, the following applies:

$$predSamples[x][y] = 1 << (BitDepth_C - 1) \tag{8-122}$$

– Otherwise, the following ordered steps apply:

1. The collocated luma samples pY[x][y] with x = 0..nTbW * 2 − 1, y= 0..nTbH * 2 − 1 are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations (xTbY + x, yTbY + y).
2. The neighbouring luma samples samples pY[x][y] are derived as follows:
   – When numSampL is greater than 0, the neighbouring left luma samples pY[x][y] with x = −1..−3, y = 0..2 * numSampL − 1, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations (xTbY + x , yTbY +y).

– When numSampT is greater than 0, the neighbouring top luma samples pY[x][y] with x = 0..2 * numSampT − 1, y = −1, −2, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations (xTbY+ x, yTbY + y).

– When availTL is equal to TRUE, the neighbouring top-left luma samples pY[x][y] with x = −1, y = −1, −2, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations (xTbY+ x, yTbY + y).

3. The down-sampled collocated luma samples pDsY[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

– pDsY[x][y] with x = 1..nTbW − 1, y = 0..nTbH − 1 is derived as follows:

$$\begin{aligned} pDsY[x][y] = (& pY[2 * x - 1][2 * y] + pY[2 * x - 1][2 * y + 1] + \\ & 2* pY[2 * x][2 * y] + 2*pY[2 * x][2 * y + 1] + \\ & pY[2 * x + 1][2 * y] + pY[2 * x + 1][2 * y + 1] + 4) >> 3 \end{aligned} \quad (8\text{-}123)$$

– If availL is equal to TRUE, pDsY[0][y] with y = 0..nTbH − 1 is derived as follows:

$$\begin{aligned} pDsY[0][y] = (& pY[-1][2 * y] + pY[-1][2 * y + 1] + \\ & 2* pY[0][2 * y] + 2*pY[0][2*y + 1] + \\ & pY[1][2 * y] + pY[1][2 * y + 1] + 4) >> 3 \end{aligned} \quad (8\text{-}124)$$

– Otherwise, pDsY[0][y] with y = 0..nTbH − 1 is derived as follows:

$$pDsY[0][y] = (pY[0][2 * y] + pY[0][2 * y + 1] + 1) >> 1 \quad (8\text{-}125)$$

4. When numSampL is greater than 0, the down-sampled neighbouring left luma samples pLeftDsY[y] with y = 0..numSampL − 1 are derived as follows:

$$\begin{aligned} pLeftDsY[y] = (& pY[-1][2 * y] + pY[-1][2 * y + 1] + \\ & 2* pY[-2][2 * y] + 2*pY[-2][2 * y + 1] + \\ & pY[-3][2 * y] + pY[-3][2 * y + 1] + 4) >> 3 \end{aligned} \quad (8\text{-}126)$$

5. When numSampT is greater than 0, the down-sampled neighbouring top luma samples pTopDsY[x] with x = 0..numSampT − 1 are specified as follows:

– pTopDsY[x] with x = 1..numSampT − 1 is derived as follows:

– If bCTUboundary is equal to FALSE, the following applies:

$$\begin{aligned} pTopDsY[x] = (& pY[2 * x - 1][-2] + pY[2 * x - 1][-1] + \\ & 2* pY[2 * x][-2] + 2*pY[2 * x][-1] + \\ & pY[2 * x + 1][-2] + pY[2 * x + 1][-1] + 4) >> 3 \end{aligned} \quad (8\text{-}127)$$

– Otherwise (bCTUboundary is equal to TRUE), the following applies:

$$\begin{aligned} pTopDsY[x] = (& pY[2 * x - 1][-1] + \\ & 2* pY[2 * x][-1] + \\ & pY[2 * x + 1][-1] + 2) >> 2 \end{aligned} \quad (8\text{-}128)$$

– pTopDsY[0] is derived as follows:

– If availTL is equal to TRUE and bCTUboundary is equal to FALSE, the following applies:

$$\begin{aligned} pTopDsY[0] = (& pY[-1][-2] + pY[-1][-1] + \\ & 2* pY[0][-2] + 2*pY[0][-1] + \\ & pY[1][-2] + pY[1][-1] + 4) >> 3 \end{aligned} \quad (8\text{-}129)$$

– Otherwise if availTL is equal to TRUE and bCTUboundary is equal to TRUE, the following applies:

$$\begin{aligned} pTopDsY[0] = (& pY[-1][-1] + \\ & 2* pY[0][-1] + \\ & pY[1][-1] + 2) >> 2 \end{aligned} \quad (8\text{-}130)$$

– Otherwise if availTL is equal to FALSE and bCTUboundary is equal to FALSE, the following applies:

pTopDsY[0] = (pY[0][−2] + pY[0][−1] + 1) >> 1 (8-131)

– Otherwise (availTL is equal to FALSE and bCTUboundary is equal to TRUE), the following applies:

pTopDsY[0] = pY[0][−1] (8-132)

6. The variables nS, xS, yS are derived as follows:
   – If predModeIntra is equal to INTRA_LT_CCLM, the following applies:

nS = ((availL && availT) ? Min(nTbW, nTbH) : (availL ? nTbH : nTbW)) (8-133)

xS = 1 << (((nTbW > nTbH) && availL && availT) ? (Log2(nTbW) − Log2(nTbH)) : 0) (8-134)

yS = 1 << (((nTbH > nTbW) && availL && availT) ? (Log2(nTbH) − Log2(nTbW)) : 0) (8-135)

   – Otherwise if predModeIntra is equal to INTRA_L_CCLM, the following applies:

nS = numSampL (8-136)

xS = 1 (8-137)

yS = 1 (8-138)

   – Otherwise (predModeIntra is equal to INTRA_T_CCLM), the following applies:

nS = numSampT (8-139)

xS = 1 (8-140)

yS = 1 (8-141)

7. The variables minY, maxY, minC and maxC are derived as follows:
   – The variable minY is set equal to 1 << ($BitDepth_Y$) + 1 and the variable maxY is set equal to −1.
   – If availT is equal to TRUE, the variables minY, maxY, minC and maxC with x = 0..nS – 1 are derived as follows:
     – If minY is greater than pTopDsY[x * xS], the following applies:

minY = pTopDsY[x * xS] (8-142)

minC = p[x * xS][−1] (8-143)

     – If maxY is less than pTopDsY[x * xS], the following applies:

maxY = pTopDsY[x * xS] (8-144)

maxC = p[x * xS][−1] (8-145)

   – If availL is equal to TRUE, the variables minY, maxY, minC and maxC with y = 0..nS − 1 are derived as follows:
     – If minY is greater than pLeftDsY[y * yS], the following applies:

minY = pLeftDsY[y * yS] (8-146)

minC = p[−1][y * yS] (8-147)

     – If maxY is less than pLeftDsY[y * yS], the following applies:

maxY = pLeftDsY[y * yS] (8-148)

maxC = p[−1][y * yS] (8-149)

8. The variables a, b, and k are derived as follows:
   – If numSampL is greater than 0, and numSampT is greater than 0, the following applies:

$$k = 0 \tag{8-150}$$

$$a = 0 \tag{8-151}$$

$$b = 1 << (BitDepth_C - 1) \tag{8-152}$$

   – Otherwise, the following applies:

$$shift = (BitDepth_C > 8) \ ? \ BitDepth_C - 9 \ : \ 0 \tag{8-153}$$

$$add = shift \ ? \ 1 << (shift - 1) \ : \ 0 \tag{8-154}$$

$$diff = (maxY - minY + add) >> shift \tag{8-155}$$

$$k = 16 \tag{8-156}$$

   – If diff is greater than 0, the following applies:

$$div = ((maxC - minC) * (Floor(2^{32} / diff) - Floor(2^{16} / diff) * 2^{16}) + 2^{15}) >> 16 \tag{8-157}$$

$$a = ((maxC - minC) * Floor(2^{16} / diff) + div + add) >> shift \tag{8-158}$$

   – Otherwise, the following applies:

$$a = 0 \tag{8-159}$$

$$b = minC - ((a * minY) >> k) \tag{8-160}$$

9. The prediction samples predSamples[x][y] with x = 0..nTbW − 1, y = 0.. nTbH − 1 are derived as follows:

$$predSamples[x][y] = Clip1C(((pDsY[x][y] * a) >> k) + b) \tag{8-161}$$

8.2.4.2.9 Position-dependent intra prediction sample filtering process

Inputs to this process are:

– the intra prediction mode predModeIntra,
– a variable nTbW specifying the transform block width,
– a variable nTbH specifying the transform block height,
– a variable refW specifying the reference samples width,
– a variable refH specifying the reference samples height,
– the predicted samples predSamples[x][y], with x = 0..nTbW − 1, y = 0..nTbH − 1,
– the neighbouring samples p[x][y], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1,
– a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified predicted samples predSamples[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1.

Depending on the value of cIdx, the function clip1Cmp is set as follows:

– If cIdx is equal to 0, clip1Cmp is set equal to $Clip1_Y$.
– Otherwise, clip1Cmp is set equal to $Clip1_C$.

The variable nScale is set to ((Log2(nTbW) + Log2(nTbH) − 2) >> 2).

The reference sample arrays mainRef[x] and sideRef[y], with x = 0..refW − 1 and y = 0..refH − 1 are derived as follows:

mainRef[x] = p[x][−1] (8-162)
sideRef[y] = p[−1][y]

The variables refL[x][y], refT[x][y], wT[y], wL[x] and wTL[x][y] with x = 0..nTbW − 1, y =0..nTbH − 1 are derived as follows:

– If predModeIntra is equal to INTRA_PLANAR, INTRA_DC, INTRA_ANGULAR18, or INTRA_ANGULAR50, the following applies:

refL[x][y] = p[−1][y] (8-163)

refT[x][y] = p[x][−1] (8-164)

wT[y] = 32 >> ((y << 1) >> nScale) (8-165)

wL[x] = 32 >> ((x << 1) >> nScale) (8-166)

wTL[x][y] = (predModeIntra == INTRA_DC) ? ((wL[x] >> 4) + (wT[y] >> 4)) : 0 (8-167)

– Otherwise, if predModeIntra is equal to INTRA_ANGULAR2 or INTRA_ANGULAR66, the following applies:

refL[x][y] = p[−1][x + y + 1] (8-168)

refT[x][y] = p[x + y + 1][−1] (8-169)

wT[y] = (32 >> 1) >> ((y << 1) >> nScale) (8-170)

wL[x] = (32 >> 1) >> ((x << 1) >> nScale) (8-171)

wTL[x][y] = 0 (8-172)

– Otherwise, if predModeIntra is less than or equal to INTRA_ANGULAR10, the following ordered steps apply:

1. The variables dXPos[y], dXFrac[y], dXInt[y] and dX[x][y] are derived as follows using invAngle as specified in clause 8.2.4.2.7 depending on intraPredMode:

dXPos[y] = ((y + 1) * invAngle + 2) >> 2
dXFrac[y] = dXPos[y] & 63 (8-173)
dXInt[y] = dXPos [y] >> 6
dX[x][y] = x + dXInt[y]

2. The variables refL[x][y], refT[x][y], wT[y], wL[x] and wTL[x][y] are derived as follows:

refL[x][y] = 0 (8-174)

refT[x][y] = (dX[x][y] < refW − 1) ? ((64 − dXFrac[y]) * mainRef[dX[x][y]] + dXFrac[y] * mainRef[dX[x][y] + 1] + 32) >> 6 : 0 (8-175)

wT[y] = (dX[x][y] < refW − 1) ? 32 >> ((y << 1) >> nScale) : 0 (8-176)

wL[x] = 0 (8-177)

wTL[x][y] = 0 (8-178)

– Otherwise, if predModeIntra is greater than or equal to INTRA_ANGULAR58, the following ordered steps apply:

1. The variables dYPos[x], dYFrac[x], dYInt[x] and dY[x][y] are derived as follows using invAngle as specified in clause 8.2.4.2.7 depending on intraPredMode:

dYPos[x] = ((x + 1) * invAngle + 2) >> 2
dYFrac[x] = dYPos[x] & 63 (8-179)
dYInt[x] = dYPos[x] >> 6
dY[x][y] = y + dYInt[x]

2. The variables refL[x][y], refT[x][y], wT[y], wL[x] and wTL[x][y] are derived as follows:

$$refL[x][y] = (dY[x][y] < refH - 1) \ ? \ ((64 - dYFrac[x]) * sideRef[dY[x][y]] + dYFrac[x] * sideRef[dY[x][y] + 1] + 32) >> 6 \ : \ 0 \quad (8\text{-}180)$$

$$refT[x][y] = 0 \quad (8\text{-}181)$$

$$wT[y] = 0 \quad (8\text{-}182)$$

$$wL[x] = (dY[x][y] < refH - 1) \ ? \ 32 >> ((x << 1) >> nScale) \ : \ 0 \quad (8\text{-}183)$$

$$wTL[x][y] = 0 \quad (8\text{-}184)$$

– Otherwise, refL[x][y], refT[x][y], wT[y], wL[x] and wTL[x][y] are all set equal to 0.

The values of the modified predicted samples predSamples[x][y], with x = 0..nTbW − 1, y =0..nTbH − 1 are derived as follows:

$$predSamples[x][y] = clip1Cmp((refL[x][y] * wL[x] + refT[x][y] * wT[y] - p[-1][-1] * wTL[x][y] + (64 - wL[x] - wT[y] + wTL[x][y]) * predSamples[x][y] + 32) >> 6) \quad (8\text{-}185)$$

8.3 Decoding process for coding units coded in inter prediction mode

8.3.1 General decoding process for coding units coded in inter prediction mode

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Output of this process is a modified reconstructed picture before in-loop filtering.

The derivation process for quantization parameters as specified in clause 8.4.1 is invoked with the luma location (xCb, yCb), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight, and the variable treeType set equal to SINGLE_TREE as inputs.

The decoding process for coding units coded in inter prediction mode consists of the following ordered steps:

1. The motion vector components and reference indices of the current coding unit are derived as follows:
   – If MotionModelIdc[xCb][yCb] is equal to 0, the following applies:
     – The derivation process for motion vector components and reference indices as specified in clause 8.3.2.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx], the chroma motion vectors mvCL0[xSbIdx][ySbIdx] and mvCL1[xSbIdx][ySbIdx], the reference indices refIdxL0 and refIdxL1 and the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 as outputs.
   – Otherwise (MotionModelIdc[xCb][yCb] is equal to 1 or 2), the following applies:
     – The derivation process for affine motion vector components and reference indices as specified in clause 8.3.3.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight as inputs, and the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0 and predFlagL1, the luma motion vector array mvLX[xSbIdx][ySbIdx], and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, and ySbIdx = 0..(cbHeight >> 2) − 1, with X being 0 or 1 as outputs.
2. The prediction samples of the current coding unit are derived as follows:

– The variables numSbX and numSbY are modified as follows

numSbX = (MotionModelIdc[xCb][yCb] = = 0) ? numSbX : (cbWidth >> 2) (8-186

numSbY = (MotionModelIdc[xCb][yCb] = = 0) ? numSbY : (cbHeight >> 2) (8-187

– The decoding process for inter blocks as specified in clause 8.3.4.1 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight, the luma motion vectors mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx], the chroma motion vectors mvCL0[xSbIdx][ySbIdx] and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array $predSamples_L$ of prediction luma samples and two (cbWidth / 2)x(cbHeight / 2) arrays $predSamples_{Cr}$ and $predSamples_{Cr}$ of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

3. The variables NumSbX[xCb][yCb] and NumSbY[xCb][yCb] are set equal to numSbX and numSbY, respectively.

4. The residual samples of the current coding unit are derived as follows:

   – The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location (xTb0, yTb0) set equal to the luma location (xCb, yCb), the width nTbW set equal to the luma coding block width cbWidth, the height nTbH set equal to the luma coding block height cbHeight and the variable cIdxset equal to 0 as inputs, and the array $resSamples_L$ as output.

   – The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location (xTb0, yTb0) set equal to the chroma location (xCb / 2, yCb / 2), the width nTbW set equal to the chroma coding block width cbWidth / 2, the height nTbH set equal to the chroma coding block height cbHeight / 2 and the variable cIdxset equal to 1 as inputs, and the array $resSamples_{Cb}$ as output.

   – The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.3.5 is invoked with the location (xTb0, yTb0) set equal to the chroma location (xCb / 2, yCb / 2), the width nTbW set equal to the chroma coding block width cbWidth / 2, the height nTbH set equal to the chroma coding block height cbHeight / 2 and the variable cIdxset equal to 2 as inputs, and the array $resSamples_{Cr}$ as output.

5. The reconstructed samples of the current coding unit are derived as follows:

   – The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location (xB, yB) set equal to (xCb, yCb), the block width bWidth set equal to cbWidth, the block height bHeight set equal to cbHeight, the variable cIdx set equal to 0, the (cbWidth)x(cbHeight) array predSamples set equal to $predSamples_L$ and the (cbWidth)x(cbHeight) array resSamples set equal to $resSamples_L$ as inputs, and the output is a modified reconstructed picture before in-loop filtering.

   – The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location (xB, yB) set equal to (xCb / 2, yCb / 2), the block width bWidth set equal to cbWidth / 2, the block height bHeight set equal to cbHeight / 2, the variable cIdx set equal to 1, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to $predSamples_{Cb}$ and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to $resSamples_{Cb}$ as inputs, and the output is a modified reconstructed picture before in-loop filtering.

   – The picture reconstruction process for a colour component as specified in clause 8.4.5 is invoked with the block location (xB, yB) set equal to (xCb / 2, yCb / 2), the block width bWidth set equal to cbWidth / 2, the block height bHeight set equal to cbHeight / 2, the variable cIdx set equal to 2, the (cbWidth / 2)x(cbHeight / 2) array predSamples set equal to $predSamples_{Cb}$ and the (cbWidth / 2)x(cbHeight / 2) array resSamples set equal to $resSamples_{Cb}$ as inputs, and the output is a modified reconstructed picture before in-loop filtering.

8.3.2 Derivation process for motion vector components and reference indices

8.3.2.1 General

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the chroma motion vectors in 1/32 fractional-sample accuracy mvCL0[xSbIdx][ySbIdx] and mvCL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

Let the variable LX be RefPicListX, with X being 0 or 1, of the current picture.

For the derivation of the variables mvL0 and mvL1, refIdxL0 and refIdxL1, as well as predFlagL0 and predFlagL1, the following applies:

– If merge_flag[xCb][yCb] is equal to 1, the derivation process for luma motion vectors for merge mode as specified in clause 8.3.2.2 is invoked with the luma location (xCb, yCb), the variables cbWidth and cbHeight inputs, and the output being the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[xSbIdx][ySbIdx], mvL1[xSbIdx][ySbIdx], the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

– Otherwise, for X being replaced by either 0 or 1 in the variables predFlagLX, mvLX and refIdxLX, in PRED_LX, and in the syntax elements ref_idx_lX and MvdLX, the following ordered steps apply:

1. The variables numSbX and numSbY are derived as follows:

$$numSbX = 1 \tag{8-188}$$

$$numSbY = 1 \tag{8-189}$$

2. The variables refIdxLX and predFlagLX are derived as follows:

   - If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

$$refIdxLX = ref\_idx\_lX[xCb][yCb] \tag{8-190}$$

$$predFlagLX[0][0] = 1 \tag{8-191}$$

   - Otherwise, the variables refIdxLX and predFlagLX are specified by:

$$refIdxLX = -1 \tag{8-192}$$

$$predFlagLX[0][0] = 0 \tag{8-193}$$

3. The variable mvdLX is derived as follows:

$$mvdLX[0] = MvdLX[xCb][yCb][0] \tag{8-194}$$

$$mvdLX[1] = MvdLX[xCb][yCb][1] \tag{8-195}$$

4. When predFlagLX is equal to 1, the derivation process for luma motion vector prediction in clause 8.3.2.8 is invoked with the luma coding block location (xCb, yCb), the coding block width cbWidth, the coding block height cbHeight and the variable refIdxLX as inputs, and the output being mvpLX.

5. When predFlagLX is equal to 1, the luma motion vector mvLX is derived as follows:

$$uLX[0] = (mvpLX[0] + mvdLX[0] + 2^{18}) \% 2^{18} \tag{8-196}$$

$$mvLX[0][0][0] = (uLX[0] >= 2^{17}) ? (uLX[0] - 2^{18}) : uLX[0] \tag{8-197}$$

$$uLX[\ 1\ ] = (\ mvpLX[\ 1\ ] + mvdLX[\ 1\ ] + 2^{18}\ )\ \%\ 2^{18} \quad (8\text{-}198)$$

$$mvLX[\ 0\ ][0\ ][\ 1\ ] = (\ uLX[\ 1\ ]\ >=\ 2^{17}\ )\ ?\ (\ uLX[\ 1\ ] - 2^{18}\ ) : uLX[\ 1\ ] \quad (8\text{-}199)$$

NOTE 1– The resulting values of mvLX[0] and mvLX[1] as specified above will always be in the range of -2^{17} to $2^{17} - 1$, inclusive.

When predFlagLX, with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in clause 8.3.2.13 is invoked with mvLX[xSbIdx][ySbIdx] as input, and the output being mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

8.3.2.2 Derivation process for luma motion vectors for merge mode

This process is only invoked when merge_flag[xCb][yPb] is equal to 1, where (xCb, yCb) specify the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture.

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0[xSbIdx][ySbIdx] and predFlagL1[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

The motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1 and the prediction utilization flags predFlagL0 and predFlagL1 are derived by the following ordered steps:

1. The derivation process for merging candidates from neighbouring coding units as specified in clause 8.3.2.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, and the luma coding block height cbHeight as inputs, and the output being the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$ and $availableFlagB_2$, the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$ and $refIdxLXB_2$, the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$ and $predFlagLXB_2$, and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$ and $mvLXB_2$, with X being 0 or 1.

2. The derivation process for subblock-based temporal merging candidates as specified in clause 8.3.2.4 is invoked with the luma location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight , the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$, the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$, the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$ and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$ as inputs and the output being the availability flag availableFlagSbCol, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the reference indices refIdxLXSbCol, the luma motion vectors mvLXSbCol[xSbIdx][ySbIdx] and the prediction list utilization flags predFlagLXSbCol[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1.

3. The reference indices for the temporal merging candidate, refIdxLXCol, with X being 0 or 1, are set equal to 0.

4. The derivation process for temporal luma motion vector prediction as specified in in clause 8.3.2.11 is invoked with the luma location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL0Col as inputs, and the output being the availability flag availableFlagL0Col and the temporal motion vector mvL0Col. The variables availableFlagCol, predFlagL0Col and predFlagL1Col are derived as follows:

$$availableFlagCol = availableFlagL0Col \quad (8\text{-}200)$$

predFlagL0Col = availableFlagL0Col (8-201)

predFlagL1Col = 0 (8-202)

5. When slice_type is equal to B, the derivation process for temporal luma motion vector prediction as specified in clause 8.3.2.11 is invoked with the luma location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL1Col as inputs, and the output being the availability flag availableFlagL1Col and the temporal motion vector mvL1Col. The variables availableFlagCol and predFlagL1Col are derived as follows:

availableFlagCol = availableFlagL0Col || availableFlagL1Col (8-203)

predFlagL1Col = availableFlagL1Col (8-204)

6. The merging candidate list, mergeCandList, is constructed as follows:

```
i = 0
if( availableFlagA1 )
    mergeCandList[ i++ ] = A1
if( availableFlagB1 )
    mergeCandList[ i++ ] = B1
if( availableFlagB0 )
    mergeCandList[ i++ ] = B0                                   (8-205)
if( availableFlagA0 )
    mergeCandList[ i++ ] = A0
if( availableFlagSbCol )
    mergeCandList[ i++ ] = SbCol
if( availableFlagB2 )
    mergeCandList[ i++ ] = B2
if( availableFlagCol )
    mergeCandList[ i++ ] = Col
```

7. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the mergeCandList.

8. When slice_type is equal to B, the derivation process for combined bi-predictive merging candidates specified in clause 8.3.2.6 is invoked with mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList, numCurrMergeCand and numOrigMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices $refIdxL0combCand_k$ and $refIdxL1combCand_k$, the prediction list utilization flags $predFlagL0combCand_k$ and $predFlagL1combCand_k$ and the motion vectors $mvL0combCand_k$ and $mvL1combCand_k$ of every new candidate $combCand_k$ being added into mergeCandList. The number of candidates being added, numCombMergeCand, is set equal to (numCurrMergeCand − numOrigMergeCand). When numCombMergeCand is greater than 0, k ranges from 0 to numCombMergeCand − 1, inclusive.

9. The derivation process for zero motion vector merging candidates specified in clause 8.3.2.7 is invoked with the mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList and numCurrMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices $refIdxL0zeroCand_m$ and $refIdxL1zeroCand_m$, the prediction list utilization flags $predFlagL0zeroCand_m$ and $predFlagL1zeroCand_m$ and the motion vectors $mvL0zeroCand_m$ and $mvL1zeroCand_m$ of every new candidate $zeroCand_m$ being added into mergeCandList. The number of candidates being added, numZeroMergeCand, is set equal to (numCurrMergeCand − numOrigMergeCand − numCombMergeCand). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand − 1, inclusive.

10. The following applies:

– If mergeCandList[merge_idx[xCb][yCb]] is equal to SbCol, the following assignments are made for xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1

refIdxLX = refIdxLXSbCol (8-206)

predFlagLX[xSbIdx][ySbIdx] = predFlagLXSbCol[xSbIdx][ySbIdx] (8-207)

mvLX[xSbIdx][ySbIdx][0] = mvLXSbCol[xSbIdx][ySbIdx][0] (8-208)

mvLX[xSbIdx][ySbIdx][1] = mvLXSbCol[xSbIdx][ySbIdx][1] (8-209)

– Otherwise, the following assignments are made with N being the candidate at position merge_idx[xCb][yCb] in the merging candidate list mergeCandList (N = mergeCandList[merge_idx[xCb][yCb]]) and X being replaced by 0 or 1:

numSbX = 1 (8-210)

numSbY = 1 (8-211)

refIdxLX = refIdxLXN (8-212)

predFlagLX[0][0] = predFlagLXN (8-213)

mvLX[0][0][0] = mvLXN[0] (8-214)

mvLX[0][0][1] = mvLXN[1] (8-215)

8.3.2.3 Derivation process for spatial merging candidates

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are as follows, with X being 0 or 1:

– the availability flags availableFlagA_0, availableFlagA_1, availableFlagB_0, availableFlagB_1 and availableFlagB_2 of the neighbouring coding units,

– the reference indices refIdxLXA_0, refIdxLXA_1, refIdxLXB_0, refIdxLXB_1 and refIdxLXB_2 of the neighbouring coding units,

– the prediction list utilization flags predFlagLXA_0, predFlagLXA_1, predFlagLXB_0, predFlagLXB_1 and predFlagLXB_2 of the neighbouring coding units,

– the motion vectors in 1/16 fractional-sample accuracy mvLXA_0, mvLXA_1, mvLXB_0, mvLXB_1 and mvLXB_2 of the neighbouring coding units.

For the derivation of availableFlagA_1, refIdxLXA_1, predFlagLXA_1 and mvLXA_1 the following applies:

– The luma location (xNbA_1, yNbA_1) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + cbHeight − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbA_1, yNbA_1) as inputs, and the output is assigned to the block availability flag availableA_1.

– The variables availableFlagA_1, refIdxLXA_1, predFlagLXA_1 and mvLXA_1 are derived as follows:

  – If availableA_1 is equal to FALSE, availableFlagA_1 is set equal to 0, both components of mvLXA_1 are set equal to 0, refIdxLXA_1 is set equal to −1 and predFlagLXA_1 is set equal to 0, with X being 0 or 1.

  – Otherwise, availableFlagA_1 is set equal to 1 and the following assignments are made:

mvLXA_1 = MvLX[xNbA_1][yNbA_1] (8-216)

refIdxLXA_1 = RefIdxLX[xNbA_1][yNbA_1] (8-217)

predFlagLXA_1 = PredFlagLX[xNbA_1][yNbA_1] (8-218)

For the derivation of availableFlagB_1, refIdxLXB_1, predFlagLXB_1 and mvLXB_1 the following applies:

– The luma location (xNbB_1, yNbB_1) inside the neighbouring luma coding block is set equal to (xCb + cbWidth − 1, yCb − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbB_1, yNbB_1) as inputs, and the output is assigned to the block availability flag availableB_1.

– The variables availableFlagB_1, refIdxLXB_1, predFlagLXB_1 and mvLXB_1 are derived as follows:

  – If one or more of the following conditions are true, availableFlagB_1 is set equal to 0, both components of mvLXB_1 are set equal to 0, refIdxLXB_1 is set equal to −1 and predFlagLXB_1 is set equal to 0, with X being 0 or 1:

    – availableB_1 is equal to FALSE.

    – availableA_1 is equal to TRUE and the luma locations (xNbA_1, yNbA_1) and (xNbB_1, yNbB_1) have the same motion vectors and the same reference indices.

  – Otherwise, availableFlagB_1 is set equal to 1 and the following assignments are made:

$$mvLXB_1 = MvLX[\ xNbB_1\ ][\ yNbB_1\ ] \tag{8-219}$$

$$refIdxLXB_1 = RefIdxLX[\ xNbB_1\ ][\ yNbB_1\ ] \tag{8-220}$$

$$predFlagLXB_1 = PredFlagLX[\ xNbB_1\ ][\ yNbB_1\ ] \tag{8-221}$$

For the derivation of availableFlagB_0, refIdxLXB_0, predFlagLXB_0 and mvLXB_0 the following applies:

– The luma location (xNbB_0, yNbB_0) inside the neighbouring luma coding block is set equal to (xCb + cbWidth, yCb − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbB_0, yNbB_0) as inputs, and the output is assigned to the block availability flag availableB_0.

– The variables availableFlagB_0, refIdxLXB_0, predFlagLXB_0 and mvLXB_0 are derived as follows:

  – If one or more of the following conditions are true, availableFlagB_0 is set equal to 0, both components of mvLXB_0 are set equal to 0, refIdxLXB_0 is set equal to −1 and predFlagLXB_0 is set equal to 0, with X being 0 or 1:

    – availableB_0 is equal to FALSE.

    – availableB_1 is equal to TRUE and the luma locations (xNbB_1, yNbB_1) and (xNbB_0, yNbB_0) have the same motion vectors and the same reference indices.

  – Otherwise, availableFlagB_0 is set equal to 1 and the following assignments are made:

$$mvLXB_0 = MvLX[\ xNbB_0\ ][\ yNbB_0\ ] \tag{8-222}$$

$$refIdxLXB_0 = RefIdxLX[\ xNbB_0\ ][\ yNbB_0\ ] \tag{8-223}$$

$$predFlagLXB_0 = PredFlagLX[\ xNbB_0\ ][\ yNbB_0\ ] \tag{8-224}$$

For the derivation of availableFlagA_0, refIdxLXA_0, predFlagLXA_0 and mvLXA_0 the following applies:

– The luma location (xNbA_0, yNbA_0) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb + cbWidth).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbA_0, yNbA_0) as inputs, and the output is assigned to the block availability flag availableA_0.

– The variables availableFlagA_0, refIdxLXA_0, predFlagLXA_0 and mvLXA_0 are derived as follows:

  – If one or more of the following conditions are true, availableFlagA_0 is set equal to 0, both components of mvLXA_0 are set equal to 0, refIdxLXA_0 is set equal to −1 and predFlagLXA_0 is set equal to 0, with X being 0 or 1:

– availableA$_0$ is equal to FALSE.

– availableA$_1$ is equal to TRUE and the luma locations (xNbA$_1$, yNbA$_1$) and (xNbA$_0$, yNbA$_0$) have the same motion vectors and the same reference indices.

– Otherwise, availableFlagA$_0$ is set equal to 1 and the following assignments are made:

$$mvLXA_0 = MvLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}225)$$

$$refIdxLXA_0 = RefIdxLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}226)$$

$$predFlagLXA_0 = PredFlagLX[\ xNbA_0\ ][\ yNbA_0\ ] \quad (8\text{-}227)$$

For the derivation of availableFlagB$_2$, refIdxLXB$_2$, predFlagLXB$_2$ and mvLXB$_2$ the following applies:

– The luma location (xNbB$_2$, yNbB$_2$) inside the neighbouring luma coding block is set equal to (xCb − 1, yCb − 1).

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location (xNbB$_2$, yNbB$_2$) as inputs, and the output is assigned to the block availability flag availableB$_2$.

– The variables availableFlagB$_2$, refIdxLXB$_2$, predFlagLXB$_2$ and mvLXB$_2$ are derived as follows:

  – If one or more of the following conditions are true, availableFlagB$_2$ is set equal to 0, both components of mvLXB$_2$ are set equal to 0, refIdxLXB$_2$ is set equal to −1 and predFlagLXB$_2$ is set equal to 0, with X being 0 or 1:

    – availableB$_2$ is equal to FALSE.

    – availableA$_1$ is equal to TRUE and the luma locations (xNbA$_1$, yNbA$_1$) and (xNbB$_2$, yNbB$_2$) have the same motion vectors and the same reference indices.

    – availableB$_1$ is equal to TRUE and the luma locations (xNbB$_1$, yNbB$_1$) and (xNbB$_2$, yNbB$_2$) have the same motion vectors and the same reference indices.

    – availableFlagA$_0$ + availableFlagA$_1$ + availableFlagB$_0$ + availableFlagB$_1$ is equal to 4.

  – Otherwise, availableFlagB$_2$ is set equal to 1 and the following assignments are made:

$$mvLXB_2 = MvLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}228)$$

$$refIdxLXB_2 = RefIdxLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}229)$$

$$predFlagLXB_2 = PredFlagLX[\ xNbB_2\ ][\ yNbB_2\ ] \quad (8\text{-}230)$$

8.3.2.4 Derivation process for subblock-based temporal merging candidates

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

– the availability flags availableFlagA$_0$, availableFlagA$_1$, availableFlagB$_0$, and availableFlagB$_1$ of the neighbouring coding units,

– the reference indices refIdxLXA$_0$, refIdxLXA$_1$, refIdxLXB$_0$, and refIdxLXB$_1$ of the neighbouring coding units,

– the prediction list utilization flags predFlagLXA$_0$, predFlagLXA$_1$, predFlagLXB$_0$, and predFlagLXB$_1$ of the neighbouring coding units,

– the motion vectors in 1/16 fractional-sample accuracy mvLXA$_0$, mvLXA$_1$, mvLXB$_0$, and mvLXB$_1$ of the neighbouring coding units.

Outputs of this process are:

– the availability flag availableFlagSbCol,

– the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,

– the reference indices refIdxL0SbCol and refIdxL1SbCol,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0SbCol[xSbIdx][ySbIdx] and mvL1SbCol[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,

– the prediction list utilization flags predFlagL0SbCol[xSbIdx][ySbIdx] and predFlagL1SbCol[xSbIdx][ySbIdx] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

The availability flag availableFlagSbCol is derived as follows.

– If either slice_temporal_mvp_enable_flag or sps_sbtmvp_flag is equal to 0, availableFlagSbCol is set equal to 0.

– Otherwise, the following ordered steps apply.

1. The location (xCtb, yCtb) of the top-left sample of the luma coding tree block that contains the current coding block and the location (xCtr, yCtr) of the below-right center sample of the current luma coding block are derived as follows:

$$xCtb = (xCb >> CtuLog2Size) << CtuLog2Size \quad (8\text{-}231)$$

$$yCtb = (yCb >> CtuLog2Size) << CtuLog2Size \quad (8\text{-}232)$$

$$xCtr = xCb + (cbWidth / 2) \quad (8\text{-}233)$$

$$yCtr = yCb + (cbHeight / 2) \quad (8\text{-}234)$$

2. The luma location (xColCtrCb, yColCtrCb) is set equal to the top-left sample of the collocated luma coding block covering the location given by (xCtr, yCtr) inside ColPic relative to the top-left luma sample of the collocated picture specified by ColPic.

3. The derivation process for subblock-based temporal merging base motion data as specified in clause 8.3.2.5 is invoked with the location (xCtb, yCtb), the location (xColCtrCb, yColCtrCb), the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$ and $availableFlagB_1$, and the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$ and $predFlagLXB_1$, and the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$ and $refIdxLXB_1$, and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$ and $mvLXB_1$, with X being 0 and 1 as inputs and the motion vectors ctrMvLX, the prediction list utilization flags ctrPredFlagLX and the reference indices ctrRefIdxLX of the collocated block, with X being 0 and 1, and the temporal motion vector tempMV as outputs.

4. The variable availableFlagSbCol is derived as follows:

   – If both ctrPredFlagL0 and ctrPredFlagL1 are equal to 0, availableFlagSbCol is set equal to 0.

   – Otherwise, availableFlagSbCol is set equal to 1.

When availableFlagSbCol is equal to 1, the following applies:

– The variables numSbX, numSbY, sbWidth, sbHeight and refIdxLXSbCol are derived as follows:

$$numSbX = Max(cbWidth >> Log2SbtmvpSize, 1) \quad (8\text{-}235)$$

$$numSbY = Max(cbHeight >> Log2SbtmvpSize, 1) \quad (8\text{-}236)$$

$$sbWidth = cbWidth / numSbX \quad (8\text{-}237)$$

$$sbHeight = cbHeight / numSbY \quad (8\text{-}238)$$

$$refIdxLXSbCol = 0 \quad (8\text{-}239)$$

– For xSbIdx = 0..numSbX − 1 and ySbIdx = 0 .. numSbY − 1, the motion vectors mvLXSbCol[xSbIdx][ySbIdx] and prediction list utilization flags predFlagLXSbCol[xSbIdx][ySbIdx] are derived as follows:

   – The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

$$xSb = xCb + xSbIdx * sbWidth \quad (8\text{-}240)$$

$$ySb = yCb + ySbIdx * sbHeight \quad (8\text{-}241)$$

– The location (xColSb, yColSb) of the collocated subblock inside ColPic is derived as follows.

$$xColSb = Clip3(xCtb, Min(CurPicWidthInSamplesY - 1, xCtb + (1 << CtbLog2SizeY) +3), xSb + (tempMv[0] >> 4)) \quad (8\text{-}242)$$

$$yColSb = Clip3(yCtb, Min(CurPicHeightInSamplesY - 1, yCtb + (1 << CtbLog2SizeY) +3), ySb + (tempMv[1] >> 4)) \quad (8\text{-}243)$$

– The variable currCb specifies the luma coding block covering the current coding subblock inside the current picture.

– The variable colCb specifies the luma coding block covering the modified location given by ((xColSb >> 3) << 3, (yColSb >> 3) << 3) inside the ColPic.

– The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.

– The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxL0 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL0SbCol[xSbIdx][ySbIdx] and availableFlagL0SbCol.

– The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxL1 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL1SbCol[xSbIdx][ySbIdx] and availableFlagL1SbCol.

– When availableFlagL0SbCol and availableFlagL1SbCol are both equal to 0, the following applies:

$$mvLXSbCol[xSbIdx][ySbIdx] = ctrMvLX \quad (8\text{-}244)$$

$$predFlagLXSbCol[xSbIdx][ySbIdx] = ctrPredFlagLX \quad (8\text{-}245)$$

8.3.2.5 Derivation process for subblock-based temporal merging base motion data

Inputs to this process are:

– the location (xCtb, yCtb) of the top-left sample of the luma coding tree block that contains the current coding block,

– the location (xColCtrCb, yColCtrCb) of the top-left sample of the collocated luma coding block that covers the below-right center sample.

– the availability flags availableFlagA_0, availableFlagA_1, availableFlagB_0, and availableFlagB_1 of the neighbouring coding units,

– the reference indices refIdxLXA_0, refIdxLXA_1, refIdxLXB_0, and refIdxLXB_1 of the neighbouring coding units,

– the prediction list utilization flags predFlagLXA_0, predFlagLXA_1, predFlagLXB_0, and predFlagLXB_1 of the neighbouring coding units,

– the motion vectors in 1/16 fractional-sample accuracy mvLXA_0, mvLXA_1, mvLXB_0, and mvLXB_1 of the neighbouring coding units.

Outputs of this process are:

– the motion vectors ctrMvL0 and ctrMvL1,

– the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1,

– the reference indices ctrRefIdxL0 and ctrRefIdxL1,

– the temporal motion vector tempMV.

The variables tempMv, numRefLists and bTerminate are set as follows:

$$tempMv[0] = 0 \quad (8\text{-}246)$$

tempMv[0] = 0 (8-247)

numRefLists = (slice_type = = B) ? 2: 1 (8-248)

bTerminate = FALSE (8-249)

The variable currPic specifies the current picture.

For each reference picture list LX with X ranging from 0 to (numRefList − 1), mvTemp is derived as follows:

1. When all of the following conditions are equal to true, X is set equal to !X:
   – DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice,
   – slice_type is equal to B,
   – collocated_from_l0_flag is equal to 0.
2. When bTerminate is equal to FALSE, and availableFlagA0 and predFlagLXA0 are equal to 1 and DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXA0]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXA0:
3. When bTerminate is equal to FALSE, availableFlagLB0 and predFlagLXB0 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXB0]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXB0.
4. When bTerminate is equal to FALSE, availableFlagB1 and predFlagLXB1 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXB1]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set equal to mvLXB1.
5. When bTerminate is equal to FALSE, availableFlagA1 and predFlagLXB1 are equal to 1, DiffPicOrderCnt(ColPic, RefPicListX[refIdxLXA1]) is equal to 0, bTerminate is set to TRUE and mvTemporal is set to mvLXA1.

The location (xColCb, yColCb) of the collocated block inside ColPic is derived as follows.

xColCb = Clip3(xCtb,
Min(CurPicWidthInSamplesY − 1, xCtb + (1 << CtbLog2SizeY) +3), (8-250)
xColCtrCb + (tempMv[0] >> 4))

yColCb = Clip3(yCtb,
Min(CurPicHeightInSamplesY − 1, yCtb + (1 << CtbLog2SizeY) +3), (8-251)
yColCtrCb + (tempMv[1] >> 4))

The array colPredMode is set equal to the prediction mode array CuPredMode of the collocated picture specified by ColPic.

The motion vectors ctrMvL0 and ctrMvL1, the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1, and the reference indices ctrRefIdxL0 and ctrRefIdxL1 are derived as follows:

– If colPredMode[xColCb][yColCb] is equal to MODE_INTER, the following applies:
  – The variable currCb specifies the luma coding block covering (xCtrCb ,yCtrCb) inside the current picture.
  – The variable colCb specifies the luma coding block covering the modified location given by ((xColCb >> 3) << 3, (yColCb >> 3) << 3) inside the ColPic.
  – The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
  – The derivation process for temporal motion vector prediction in subclause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL0, and sbFlag set equal to 1 as inputs and the output being assigned to centerMvL0 and centerPredFlagL0.
  – The derivation process for temporal motion vector prediction in subclause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), centerRefIdxL1, and sbFlag set equal to 1 as inputs and the output being assigned to centerMvL1 and centerPredFlagL1.
– Otherwise, the following applies:

ctrPredFlagL0 = 0 (8-252)

ctrPredFlagL1 = 0 (8-253)

8.3.2.6 Derivation process for combined bi-predictive merging candidates

Inputs to this process are:

– a merging candidate list mergeCandList,

– the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,

– the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,

– the motion vectors in 1/16 fractional-sample accuracy mvL0N and mvL1N of every candidate N in mergeCandList,

– the number of elements numCurrMergeCand within mergeCandList,

– the number of elements numOrigMergeCand within the mergeCandList after the spatial and temporal merge candidate derivation process.

Outputs of this process are:

– the merging candidate list mergeCandList,

– the number of elements numCurrMergeCand within mergeCandList,

– the reference indices refIdxL0combCand$_k$ and refIdxL1combCand$_k$ of every new candidate combCand$_k$ added into mergeCandList during the invocation of this process,

– the prediction list utilization flags predFlagL0combCand$_k$ and predFlagL1combCand$_k$ of every new candidate combCand$_k$ added into mergeCandList during the invocation of this process,

– the motion vectors in 1/16 fractional-sample accuracy mvL0combCand$_k$ and mvL1combCand$_k$ of every new candidate combCand$_k$ added into mergeCandList during the invocation of this process.

When numOrigMergeCand is greater than 1 and less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable combIdx is set equal to 0, the variable combStop is set equal to FALSE and the following ordered steps are repeated until combStop is equal to TRUE:

1. The variables l0CandIdx and l1CandIdx are derived using combIdx as specified in Table 8-7.
2. The following assignments are made, with l0Cand being the candidate at position l0CandIdx and l1Cand being the candidate at position l1CandIdx in the merging candidate list mergeCandList:
   – l0Cand = mergeCandList[l0CandIdx]
   – l1Cand = mergeCandList[l1CandIdx]
3. When all of the following conditions are true:
   – predFlagL0l0Cand == 1
   – predFlagL1l1Cand == 1
   – (DiffPicOrderCnt(RefPicList0[refIdxL0l0Cand], RefPicList1[refIdxL1l1Cand]) != 0) || (mvL0l0Cand != mvL1l1Cand)

   the candidate combCand$_k$ with k equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to combCand$_k$, and the reference indices, the prediction list utilization flags and the motion vectors of combCand$_k$ are derived as follows and numCurrMergeCand is incremented by 1:

refIdxL0combCand$_k$ = refIdxL0l0Cand (8-254)

refIdxL1combCand$_k$ = refIdxL1l1Cand (8-255)

predFlagL0combCand$_k$ = 1 (8-256)

predFlagL1combCand$_k$ = 1 (8-257)

mvL0combCand$_k$[0] = mvL0l0Cand[0] (8-258)

$mvL0combCand_k[1] = mvL0l0Cand[1]$ (8-259)

$mvL1combCand_k[0] = mvL1l1Cand[0]$ (8-260)

$mvL1combCand_k[1] = mvL1l1Cand[1]$ (8-261)

numCurrMergeCand = numCurrMergeCand + 1 (8-262)

4. The variable combIdx is incremented by 1.
5. When combIdx is equal to (numOrigMergeCand * (numOrigMergeCand − 1)) or numCurrMergeCand is equal to MaxNumMergeCand, combStop is set equal to TRUE.

Table 8-7 – Specification of l0CandIdx and l1CandIdx

combIdx	0	1	2	3	4	5	6	7	8	9	10	11
l0CandIdx	0	1	0	2	1	2	0	3	1	3	2	3
l1CandIdx	1	0	2	0	2	1	3	0	3	1	3	2

8.3.2.7 Derivation process for zero motion vector merging candidates

Inputs to this process are:

– a merging candidate list mergeCandList,
– the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,
– the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,
– the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList,
– the number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

– the merging candidate list mergeCandList,
– the number of elements numCurrMergeCand within mergeCandList,
– the reference indices $refIdxL0zeroCand_m$ and $refIdxL1zeroCand_m$ of every new candidate $zeroCand_m$ added into mergeCandList during the invocation of this process,
– the prediction list utilization flags $predFlagL0zeroCand_m$ and $predFlagL1zeroCand_m$ of every new candidate $zeroCand_m$ added into mergeCandList during the invocation of this process,
– the motion vectors $mvL0zeroCand_m$ and $mvL1zeroCand_m$ of every new candidate $zeroCand_m$ added into mergeCandList during the invocation of this process.

The variable numRefIdx is derived as follows:

– If slice_type is equal to P, numRefIdx is set equal to num_ref_idx_l0_active_minus1 + 1.
– Otherwise (slice_type is equal to B), numRefIdx is set equal to Min(num_ref_idx_l0_active_minus1 + 1, num_ref_idx_l1_active_minus1 + 1).

When numCurrMergeCand is less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable zeroIdx is set equal to 0 and the following ordered steps are repeated until numCurrMergeCand is equal to MaxNumMergeCand:

1. For the derivation of the reference indices, the prediction list utilization flags and the motion vectors of the zero motion vector merging candidate, the following applies:
   – If slice_type is equal to P, the candidate $zeroCand_m$ with m equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $zeroCand_m$, and the reference indices, the prediction list utilization flags and the motion vectors of $zeroCand_m$ are derived as follows and numCurrMergeCand is incremented by 1:

$$refIdxL0zeroCand_m = (zeroIdx < numRefIdx) ? zeroIdx : 0 \quad (8\text{-}263)$$

$$refIdxL1zeroCand_m = -1 \quad (8\text{-}264)$$

$$predFlagL0zeroCand_m = 1 \quad (8\text{-}265)$$

$$predFlagL1zeroCand_m = 0 \quad (8\text{-}266)$$

$$mvL0zeroCand_m[0] = 0 \quad (8\text{-}267)$$

$$mvL0zeroCand_m[1] = 0 \quad (8\text{-}268)$$

$$mvL1zeroCand_m[0] = 0 \quad (8\text{-}269)$$

$$mvL1zeroCand_m[1] = 0 \quad (8\text{-}270)$$

$$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}271)$$

– Otherwise (slice_type is equal to B), the candidate $zeroCand_m$ with m equal to (numCurrMergeCand − numInputMergeCand) is added at the end of mergeCandList, i.e., mergeCandList[numCurrMergeCand] is set equal to $zeroCand_m$, and the reference indices, the prediction list utilization flags and the motion vectors of $zeroCand_m$ are derived as follows and numCurrMergeCand is incremented by 1:

$$refIdxL0zeroCand_m = (zeroIdx < numRefIdx) ? zeroIdx : 0 \quad (8\text{-}272)$$

$$refIdxL1zeroCand_m = (zeroIdx < numRefIdx) ? zeroIdx : 0 \quad (8\text{-}273)$$

$$predFlagL0zeroCand_m = 1 \quad (8\text{-}274)$$

$$predFlagL1zeroCand_m = 1 \quad (8\text{-}275)$$

$$mvL0zeroCand_m[0] = 0 \quad (8\text{-}276)$$

$$mvL0zeroCand_m[1] = 0 \quad (8\text{-}277)$$

$$mvL1zeroCand_m[0] = 0 \quad (8\text{-}278)$$

$$mvL1zeroCand_m[1] = 0 \quad (8\text{-}279)$$

$$numCurrMergeCand = numCurrMergeCand + 1 \quad (8\text{-}280)$$

2. The variable zeroIdx is incremented by 1.

8.3.2.8 Derivation process for luma motion vector prediction

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the prediction mvpLX in 1/16 fractional-sample accuracy of the motion vector mvLX, with X being 0 or 1.

The motion vector predictor mvpLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for motion vector predictor candidate list as specified in clause 8.3.2.9 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height

cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1, as output.

2. The motion vector predictor mvpLX with X being 0 or 1 is derived as follows:

$$mvpLX = mvpListLX[\ mvp\_lX\_flag[\ xCb\ ][\ yCb\ ]\ ] \tag{8-281}$$

8.3.2.9 Derivation process for motion vector predictor candidate list

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is motion vector predictor candidate list mvpListLX in 1/16 fractional-sample accuracy with X being 0 or 1.

The motion vector predictor candidate list mvpListLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for spatial motion vector predictor candidates from neighbouring coding unit partitions as specified in clause 8.3.2.10 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the availability flags availableFlagLXN and the motion vectors mvLXN, with N being replaced by A or B, as output.
2. The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLXN, with N being replaced by A or B, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXN, with N being replaced by A or B, as output.
3. If both availableFlagLXA and availableFlagLXB are equal to 1 and mvLXA is not equal to mvLXB, availableFlagLXCol is set equal to 0.
4. Otherwise, the following applies:
   – The derivation process for temporal luma motion vector prediction as specified in clause 8.3.2.11 is with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
   – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLXCol, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXCol as output.
5. The motion vector predictor candidate list, mvpListLX, is constructed as follows:

```
i = 0
if( availableFlagLXA ) {
    mvpListLX[ i++ ] = mvLXA
    if( availableFlagLXB  &&  ( mvLXA  !=  mvLXB ) )
        mvpListLX[ i++ ] = mvLXB
} else if( availableFlagLXB )
    mvpListLX[ i++ ] = mvLXB                                    (8-282)
if( i < 2  &&  availableFlagLXCol )
    mvpListLX[ i++ ] = mvLXCol
while( i < 2 ) {
    mvpListLX[ i ][ 0 ] = 0
    mvpListLX[ i ][ 1 ] = 0
    i++
}
```

8.3.2.10 Derivation process for spatial motion vector predictor candidates

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Outputs of this process are (with N being replaced by A or B):

– the motion vectors mvLXN in 1/16 fractional-sample accuracy of the neighbouring coding units,

– the availability flags availableFlagLXN of the neighbouring coding units.

Figure 8-2 provides an overview of spatial motion vector neighbours.

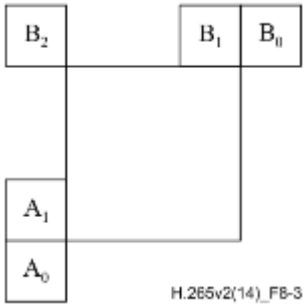


Figure 8-2 – Spatial motion vector neighbours (informative)

The variable currCb specifies the current luma coding block at luma location (xCb, yCb) and the variable currPic specifies the current picture.

The variable isScaledFlagLX, with X being 0 or 1, is set equal to 0.

The motion vector mvLXA and the availability flag availableFlagLXA are derived in the following ordered steps:

1. The sample location ($xNbA_0$, $yNbA_0$) is set equal to (xCb − 1, yCb + cbHeight) and the sample location ($xNbA_1$, $yNbA_1$) is set equal to ($xNbA_0$, $yNbA_0$ − 1).
2. The availability flag availableFlagLXA is set equal to 0 and both components of mvLXA are set equal to 0.
3. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbA_0$, $yNbA_0$) as inputs, and the output is assigned to the block availability flag $availableA_0$.
4. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbA_1$, $yNbA_1$) as inputs, and the output is assigned to the block availability flag $availableA_1$.
5. When $availableA_0$ or $availableA_1$ is equal to TRUE, the variable isScaledFlagLX is set equal to 1.
6. The following applies for ($xNbA_k$, $yNbA_k$) from ($xNbA_0$, $yNbA_0$) to ($xNbA_1$, $yNbA_1$):
   – When $availableA_k$ is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
     – If PredFlagLX[$xNbA_k$][$yNbA_k$] is equal to 1 and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbA_k$][$yNbA_k$]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

$$mvLXA = MvLX[\ xNbA_k\ ][\ yNbA_k\ ] \tag{8-283}$$

– Otherwise, when PredFlagLY[$xNbA_k$][$yNbA_k$] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[$xNbA_k$][$yNbA_k$]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

$$mvLXA = MvLY[xNbA_k][yNbA_k] \quad (8\text{-}284)$$

7. When availableFlagLXA is equal to 0, the following applies for ($xNbA_k$, $yNbA_k$) from ($xNbA_0$, $yNbA_0$) to ($xNbA_1$, $yNbA_1$) or until availableFlagLXA is equal to 1:

– When $availableA_k$ is equal to TRUE and availableFlagLXA is equal to 0, the following applies:

– If PredFlagLX[$xNbA_k$][$yNbA_k$] is equal to 1 , availableFlagLXA is set equal to 1 and the following assignments are made:

$$mvLXA = MvLX[xNbA_k][yNbA_k] \quad (8\text{-}285)$$

$$refIdxA = RefIdxLX[xNbA_k][yNbA_k] \quad (8\text{-}286)$$

$$refPicListA = RefPicListX \quad (8\text{-}287)$$

– Otherwise, if PredFlagLY[$xNbA_k$][$yNbA_k$] (with Y = !X) is equal to 1, availableFlagLXA is set equal to 1 and the following assignments are made:

$$mvLXA = MvLY[xNbA_k][yNbA_k] \quad (8\text{-}288)$$

$$refIdxA = RefIdxLY[xNbA_k][yNbA_k] \quad (8\text{-}289)$$

$$refPicListA = RefPicListY \quad (8\text{-}290)$$

– When availableFlagLXA is equal to 1, DiffPicOrderCnt(refPicListA[refIdxA], RefPicListX[refIdxLX]) is not equal to 0, mvLXA is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8\text{-}291)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8\text{-}292)$$

$$mvLXA = Clip3(-32768, 32767, Sign(distScaleFactor * mvLXA) * ((Abs(distScaleFactor * mvLXA) + 127) >> 8)) \quad (8\text{-}293)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, refPicListA[refIdxA])) \quad (8\text{-}294)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8\text{-}295)$$

The motion vector mvLXB and the availability flag availableFlagLXB are derived in the following ordered steps:

1. The sample locations ($xNbB_0$, $yNbB_0$), ($xNbB_1$, $yNbB_1$) and ($xNbB_2$, $yNbB_2$) are set equal to (xCb + cbWidth, yCb − 1), (xCb + cbWidth − 1, yCb − 1) and (xCb − 1, yCb − 1), respectively.
2. The availability flag availableFlagLXB is set equal to 0 and the both components of mvLXB are set equal to 0.
3. The following applies for ($xNbB_k$, $yNbB_k$) from ($xNbB_0$, $yNbB_0$) to ($xNbB_2$, $yNbB_2$):

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_k$, $yNbB_k$) as inputs, and the output is assigned to the block availability flag $availableB_k$.

– When $availableB_k$ is equal to TRUE and availableFlagLXB is equal to 0, the following applies:

– If PredFlagLX[$xNbB_k$][$yNbB_k$] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[$xNbB_k$][$yNbB_k$]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

$$mvLXB = MvLX[xNbB_k][yNbB_k] \quad (8\text{-}296)$$

refIdxB = RefIdxLX[$xNbB_k$][$yNbB_k$] (8-297)

– Otherwise, when PredFlagLY[$xNbB_k$][$yNbB_k$] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[$xNbB_k$][$yNbB_k$]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[$xNbB_k$][$yNbB_k$] (8-298)

refIdxB = RefIdxLY[$xNbB_k$][$yNbB_k$] (8-299)

4. When isScaledFlagLX is equal to 0 and availableFlagLXB is equal to 1, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = mvLXB (8-300)

5. When isScaledFlagLX is equal to 0, availableFlagLXB is set equal to 0 and the following applies for ($xNbB_k$, $yNbB_k$) from ($xNbB_0$, $yNbB_0$) to ($xNbB_2$, $yNbB_2$) or until availableFlagLXB is equal to 1:

– The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring luma location ($xNbB_k$, $yNbB_k$) as inputs, and the output is assigned to the block availability flag $availableB_k$.

– When $availableB_k$ is equal to TRUE and availableFlagLXB is equal to 0, the following applies:

– If PredFlagLX[$xNbB_k$][$yNbB_k$] is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLX[$xNbB_k$][$yNbB_k$] (8-301)

refIdxB = RefIdxLX[$xNbB_k$][$yNbB_k$] (8-302)

refPicListB = RefPicListX (8-303)

– Otherwise, when PredFlagLY[$xNbB_k$][$yNbB_k$] (with Y = !X) is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[$xNbB_k$][$yNbB_k$] (8-304)

refIdxB = RefIdxLY[$xNbB_k$][$yNbB_k$] (8-305)

refPicListB = RefPicListY (8-306)

– When availableFlagLXB is equal to 1, DiffPicOrderCnt(refPicListB[refIdxB], RefPicListX[refIdxLX]) is not equal to 0, mvLXB is derived as follows:

tx = (16384 + (Abs(td) >> 1)) / td (8-307)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-308)

mvLXB = Clip3(−32768, 32767, Sign(distScaleFactor * mvLXB) *
((Abs(distScaleFactor * mvLXB) + 127) >> 8)) (8-309)

where td and tb are derived as follows:

td = Clip3(−128, 127, DiffPicOrderCnt(currPic, refPicListB[refIdxB])) (8-310)

tb = Clip3(−128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) (8-311)

8.3.2.11 Derivation process for temporal luma motion vector prediction

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– a reference index refIdxLX, with X being 0 or 1.

Outputs of this process are:

– the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,

– the availability flag availableFlagLXCol.

The variable currCb specifies the current luma coding block at luma location (xCb, yCb).

The variables mvLXCol and availableFlagLXCol are derived as follows:

– If slice_temporal_mvp_enabled_flag is equal to 0, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

– Otherwise (slice_temporal_mvp_enabled_flag is equal to 1), the following ordered steps apply:

1. The bottom right collocated motion vector is derived as follows:

$$xColBr = xCb + cbWidth \tag{8-312}$$

$$yColBr = yCb + cbHeight \tag{8-313}$$

   – If yCb >> CtbLog2SizeY is equal to yColBr >> CtbLog2SizeY, yColBr is less than pic_height_in_luma_samples and xColBr is less than pic_width_in_luma_samples, the following applies:

     – The variable colCb specifies the luma coding block covering the modified location given by ((xColBr >> 3) << 3, (yColBr >> 3) << 3) inside the collocated picture specified by ColPic.

     – The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.

     – The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

   – Otherwise, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

2. When availableFlagLXCol is equal to 0, the central collocated motion vector is derived as follows:

$$xColCtr = xCb + (cbWidth >> 1) \tag{8-314}$$

$$yColCtr = yCb + (cbHeight >> 1) \tag{8-315}$$

   – The variable colCb specifies the luma coding block covering the modified location given by ((xColCtr >> 3) << 3, (yColCtr >> 3) << 3) inside the collocated picture specified by ColPic.

   – The luma location (xColCb, yColCb) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.

   – The derivation process for collocated motion vectors as specified in clause 8.3.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

8.3.2.12 Derivation process for collocated motion vectors

Inputs to this process are:

– a variable currCb specifying the current coding block,

– a variable colCb specifying the collocated coding block inside the collocated picture specified by ColPic,

– a luma location (xColCb, yColCb) specifying the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic,

– a reference index refIdxLX, with X being 0 or 1,

– a flag indicating a subblock temporal merging candidate sbFlag.

Outputs of this process are:

– the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,

– the availability flag availableFlagLXCol.

The variable currPic specifies the current picture.

The arrays predFlagL0Col[x][y], mvL0Col[x][y] and refIdxL0Col[x][y] are set equal to PredFlagL0[x][y], MvL0[x][y] and RefIdxL0[x][y], respectively, of the collocated picture specified by ColPic, and the arrays predFlagL1Col[x][y], mvL1Col[x][y] and refIdxL1Col[x][y] are set equal to PredFlagL1[x][y], MvL1[x][y] and RefIdxL1[x][y], respectively, of the collocated picture specified by ColPic.

[Ed. (BB): Define ColPic NoBackwardPredFlag.]

The variables mvLXCol and availableFlagLXCol are derived as follows:

– If colCb is coded in an intra prediction mode, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

– Otherwise, the motion vector mvCol, the reference index refIdxCol and the reference list identifier listCol are derived as follows:

  – If sbFlag is equal to 0, availableFlagLXCol is set to 1 and the following applies:

    – If predFlagL0Col[xColCb][yColCb] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL1Col[xColCb][yColCb], refIdxL1Col[xColCb][yColCb] and L1, respectively.

    – Otherwise, if predFlagL0Col[xColCb][yColCb] is equal to 1 and predFlagL1Col[xColCb][yColCb] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL0Col[xColCb][yColCb], refIdxL0Col[xColCb][yColCb] and L0, respectively.

    – Otherwise (predFlagL0Col[xColCb][yColCb] is equal to 1 and predFlagL1Col[xColCb][yColCb] is equal to 1), the following assignments are made:

      – If NoBackwardPredFlag is equal to 1, mvCol, refIdxCol and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb] and LX, respectively.

      – Otherwise, mvCol, refIdxCol and listCol are set equal to mvLNCol[xColCb][yColCb], refIdxLNCol[xColCb][yColCb] and LN, respectively, with N being the value of collocated_from_l0_flag.

  – Otherwise (sbFlag is equal to 1), the following applies:

    – If PredFlagLXCol[xColCb][yColCb] is equal to 1, mvCol, refIdxCol, and listCol are set equal to mvLXCol[xColCb][yColCb], refIdxLXCol[xColCb][yColCb], and LX, respectively, availableFlagLXCol is set to 1.

    – Otherwise (PredFlagLXCol[xColCb][yColCb] is equal to 0), the following applies:

      – If DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice and PredFlagLYCol[xColCb][yColCb] is equal to 1, mvCol, refIdxCol, and listCol are set to mvLYCol[xColCb][yColCb], refIdxLYCol[xColCb][yColCb] and LY, respectively, with Y being equal to !X where X being the value of X this process is invoked for. availableFlagLXCol is set to 1.

      – Both the components of mvLXCol are set to 0 and availableFlagLXCol is set equal to 0.

  – When availableFlagLXCol is equal to TRUE, mvLXCol and availableFlagLXCol are derived as follows:

    – If LongTermRefPic(currPic, currCb, refIdxLX, LX) is not equal to LongTermRefPic(ColPic, colCb, refIdxCol, listCol), both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

– Otherwise, the variable availableFlagLXCol is set equal to 1, refPicListCol[refIdxCol] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the slice containing coding block colCb in the collocated picture specified by ColPic, and the following applies:

colPocDiff = DiffPicOrderCnt(ColPic, refPicListCol[refIdxCol]) (8-316)

currPocDiff = DiffPicOrderCnt(currPic, RefPicListX[refIdxLX]) (8-317)

– If RefPicListX[refIdxLX] is a long-term reference picture, or colPocDiff is equal to currPocDiff, mvLXCol is derived as follows:

mvLXCol = mvCol (8-318)

– Otherwise, mvLXCol is derived as a scaled version of the motion vector mvCol as follows:

tx = (16384 + (Abs(td) >> 1)) / td (8-319)

distScaleFactor = Clip3(−4096, 4095, (tb * tx + 32) >> 6) (8-320)

mvLXCol = Clip3(−32768, 32767, Sign(distScaleFactor * mvCol) *
((Abs(distScaleFactor * mvCol) + 127) >> 8)) (8-321)

where td and tb are derived as follows:

td = Clip3(−128, 127, colPocDiff) (8-322)

tb = Clip3(−128, 127, currPocDiff) (8-323)

8.3.2.13 Derivation process for chroma motion vectors

Input to this process is a luma motion vector in 1/16 fractional-sample accuracy mvLX.

Output of this process is a chroma motion vector in 1/32 fractional-sample accuracy mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvCLX, the following applies:

mvCLX[0] = mvLX[0] * 2 / SubWidthC (8-324)

mvCLX[1] = mvLX[1] * 2 / SubHeightC (8-325)

8.3.2.14 Rounding process for motion vectors

Inputs to this process are

– the motion vector mvX,

– the right shift parameter rightShift for rounding,

– the left shift parameter leftShift for resolution increase.

Output of this process is the rounded motion vector mvX.

For the rounding of mvX, the following applies:

offset = 1 << (rightShift − 1) (8-326)

mvX[0] = (mvX[0] >= 0 ? (mvX[0] + offset) >> rightShift :
− ((− mvX[0] + offset) >>rightShift)) << leftShift (8-327)

mvX[1] = (mvX[1] >= 0 ? (mvX[1] + offset) >> rightShift :
− ((− mvX[1] + offset) >> rightShift)) << leftShift (8-328)

8.3.3 Derivation process for affine motion vector components and reference indices

8.3.3.1 General

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the luma subblock motion vector array in 1/16 fractional-sample accuracy mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1,

– the chroma subblock motion vector array in 1/32 fractional-sample accuracy mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1.

For the derivation of the variables mvLX[xSbIdx][ySbIdx] and mvCLC[xSbIdx][ySbIdx], the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1, the following applies:

– For the derivation of the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, refIdxL0, refIdxL1, predFlagL0 and predFlagL1, the following applies:

  – If merge_affine_flag[xCb][yCb] is equal to 1, the derivation process for affine control point motion vectors and reference indices in affine merge mode as specified in 8.3.3.2 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL0[cpIdx], cpMvL1[cpIdx], with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0, refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as output.

  – Otherwise (merge_affine_flag[xCb][yCb] is equal to 0), for X being replaced by either 0 or 1 in the variables predFlagLX, cpMvLX, MvdCpLX, and refIdxLX, in PRED_LX, and in the syntax element ref_idx_lX, the following ordered steps apply:

    1. The number of control point motion vectors numCpMv is set equal to MotionModelIdc[xCb][yCb] + 1.

    2. The variables refIdxLX and predFlagLX with X being 0 or 1 are derived as follows:

       - If inter_pred_idc[xCb][yCb] is equal to PRED_LX or PRED_BI,

         refIdxLX = ref_idx_lX[xCb][yCb] (8-329)

         predFlagLX = 1 (8-330)

       - Otherwise, the variables refIdxLX and predFlagLX are specified by:

         refIdxLX = −1 (8-331)

         predFlagLX = 0 (8-332)

    3. The variable mvdCpLX[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, is derived as follows:

       mvdCpLX[cpIdx][0] = MvdCpLX[xCb][yCb][cpIdx][0] (8-333)

       mvdCpLX[cpIdx][1] = MvdCpLX[xCb][yCb][cpIdx][1] (8-334)

    4. When predFlagLX is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in clause 8.3.3.4 is invoked with the luma coding block location (xCb, yCb), and the variables cbWidth, cbHeight, refIdxLX, and the number of control point motion vectors numCpMv as inputs, and the output being mvpCpLX[cpIdx] with cpIdx ranging from 0 to numCpMv − 1.

    5. When predFlagLX is equal to 1, the luma motion vectors cpMvLX[cpIdx] with cpIdx ranging from 0 to NumCpMv − 1, are derived as follows:

       uLX[cpIdx][0] = (mvpCpLX[cpIdx][0] + mvdCpLX[cpIdx][0] + 2^{18}) % 2^{18} (8-335)

$$cpMvLX[cpIdx][0] = (uLX[cpIdx][0] >= 2^{17}) ? (uLX[cpIdx][0] - 2^{18}) : uLX[cpIdx][0] \quad (8\text{-}336)$$

$$uLX[cpIdx][1] = (mvpCpLX[cpIdx][1] + mvdCpLX[cpIdx][1] + 2^{18}) \% 2^{18} \quad (8\text{-}337)$$

$$cpMvLX[cpIdx][1] = (uLX[cpIdx][1] >= 2^{17}) ? (uLX[cpIdx][1] - 2^{18}) : uLX[cpIdx][1] \quad (8\text{-}338)$$

– The derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.3.3.7 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma prediction block height cbHeight, the number of control point motion vectors numCpMv, the control point motion vectors cpMvL1[cpIdx], cpMvL0[cpIdx] with cpIdx being 0..2, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 as inputs, the luma motion vector array mvLX[xSbIdx][ySbIdx] and the chroma motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1 as output.

– For xSbIdx = 0..(cbWidth >> 2) − 1 and ySbIdx = 0..(cbHeight >> 2) − 1, the motion vectors MvLX with X being 0 and 1 are derived as follows:

  – The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

$$(xSb, ySb) = (xCb + xSbIdx * 4, yCb + ySbIdx * 4) \quad (8\text{-}339)$$

  – If xSbIdx and ySbIdx are both equal to 0 (top-left subblock), the following applies for x = 0..3 and y = 0..3:

$$MvL0[xSb + x][ySb + y] = cpMvL0[0] \quad (8\text{-}340)$$

$$MvL1[xSb + x][ySb + y] = cpMvL1[0] \quad (8\text{-}341)$$

  – Otherwise, if xSbIdx is equal to (cbWidth >> 2) − 1 and ySbIdx is equal to 0 (top-right subblock), the following applies for x = 0..3 and y = 0..3:

$$MvL0[xSb + x][ySb + y] = cpMvL0[1] \quad (8\text{-}342)$$

$$MvL1[xSb + x][ySb + y] = cpMvL1[1] \quad (8\text{-}343)$$

  – Otherwise, if numCpMv is equal to 3 and xSbIdx is equal to 0 and ySbIdx is equal to (cbHeight >> 2) − 1 (belowleft subblock), the following applies for x = 0..3 and y = 0..3:

$$MvL0[xSb + x][ySb + y] = cpMvL0[2] \quad (8\text{-}344)$$

$$MvL1[xSb + x][ySb + y] = cpMvL1[2] \quad (8\text{-}345)$$

  – Otherwise, the following applies for x = 0..3 and y = 0..3:

$$MvL0[xSb + x][ySb + y] = mvL0[xSbIdx][ySbIdx] \quad (8\text{-}346)$$

$$MvL1[xSb + x][ySb + y] = mvL1[xSbIdx][ySbIdx] \quad (8\text{-}347)$$

– For x = 0..cbWidth − 1 and y = 0..cbHeight − 1, the reference indices refIdxLX and the prediction list utilization flags PredFlagLX with X being 0 and 1 are derived as follows:

$$RefIdxL0[xCb + x][yCb + y] = refIdxL0 \quad (8\text{-}348)$$

$$RefIdxL1[xCb + x][yCb + y] = refIdxL1 \quad (8\text{-}349)$$

$$PredFlagL0[xCb + x][yCb + y] = predFlagL0 \quad (8\text{-}350)$$

$$PredFlagL1[xCb + x][yCb + y] = predFlagL1 \quad (8\text{-}351)$$

8.3.3.2 Derivation process for luma affine control point motion vectors and reference indices in affine merge mode

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

– the number of control point motion vectors numCpMv,

– the luma affine control point motion vectors cpMvL0[cpIdx], cpMvL1[cpIdx], with cpIdx ranging from 0 to numCpMv − 1

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

The derivation process for affine motion information in the neighbouring blocks of the current luma coding block as specified in 8.3.3.5 is invoked with the luma location (xCb, yCb), the variables cbWidth and cbHeight as inputs, the output being availablity flags $availableFlag_k$, luma locations of neighbouring blocks (xNb_k, yNb_k), the neighbouring luma coding block width nbW_k, and the neighbouring luma coding block height nbH_k with k ranging from 0 to 4.

The number of control point motion vectors numCpMv, luma affine control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx] with cpIdx ranging from 0 to numCpMv − 1, the reference indices refIdxL0 and refIdxL1, and the prediction list utilization flags predFlagL0 and predFlagL1 are derived as follows:

– The variable availableFlag is set equal to 0.

– For (xNb_k, yNb_k) with k ranging from 0 to 4, the following applies:

  – When availableFlag is equal to 0 and $availableFlag_k$ is equal to 1 the following applies:

    – The reference indices refIdxLX and the prediction list utilization flags predFlagLX with X being 0 and 1, and numCpMv are derived as follows

      $refIdxLX = RefIdxLX[xNb_k][yNb_k]$ (8-352)

      $predFlagLX = PredFlagLX[xNb_k][yNb_k]$ (8-353)

      $numCpMv = MotionModelIdc[xNb_k][yNb_k] + 1$ (8-354)

    – The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

      $MotionModelIdc[x][y] = MotionModelIdc[xNb_k][yNb_k]$ (8-355)

    – The derivation process for luma affine control point motion vectors from a neighbouring block in as specified clause 8.3.3.3 is invoked with the coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNb, yNb) set equal to the neighbouring luma coding block location (xNb_k, yNb_k), the variables nNbW and nNbH set equal to the neighbouring luma coding block width nbW_k and the neighbouring luma coding block height nbH_k, and the number of control point motion vectors numCpMv as inputs, and the luma affine control point motion vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 as output.

    – The variable availableFlag is set equal to 1.

8.3.3.3 Derivation process for luma affine control point motion vectors from a neighbouring block

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,

– a luma location (xNb, yNb) specifying the top-left sample of the neighbouring luma coding block relative to the top-left luma sample of the current picture,

– two variables nNbW and nNbH specifying the width and the height of the neighbouring luma coding block,

– the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1.

The variables log2NbW and log2NbH are derived as follows:

log2NbW = Log2(nNbW) (8-356)

log2NbH = Log2(nNbH) (8-357)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

mvScaleHor = MvLX[xNb][yNb][0] << 7 (8-358)

mvScaleVer = MvLX[xNb][yNb][1] << 7 (8-359)

dHorX = (MvLX[xNb + nNbW − 1][yNb][0] − MvLX[xNb][yNb][0]) << (7 − log2NbW) (8-360)

dVerX = (MvLX[xNb + nNbW − 1][yNb][1] − MvLX[xNb][yNb][1]) << (7 − log2NbW) (8-361)

The variables dHorY and dVerY are derived as follows:

– If MotionModelIdc[xNb][yNb] is equal to 2, the following applies:

dHorY = (MvLX[xNb][yNb + nNbH − 1][0] − MvLX[xNb][yNb][0]) << (7 − log2NbH) (8-362)

dVerY = (MvLX[xNb][yNb + nNbH − 1][1] − MvLX[xNb][yNb][1]) << (7 − log2NbH) (8-363)

– Ohterwise (MotionModelIdc[xNb][yNb] is equal to 1), the following apply,

dHorY = − dVerX (8-364)

dVerY = dHorX (8-365)

The luma affine control point motion vectors cpMvLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 are derived as follows:

– The first two control point motion vectors cpMvLX[0] and cpMvLX[1] are derived as follows:

cpMvLX[0][0] = (mvScaleHor + dHorX * (xCb − xNb) + dHorY * (yCb − yNb)) (8-366)

cpMvLX[0][1] = (mvScaleVer + dVerX * (xCb − xNb) + dVerY * (yCb − yNb)) 8-367)

cpMvLX[1][0] = (mvScaleHor + dHorX * (xCb + cbWidth − xNb) + dHorY * (yCb − yNb)) (8-368)

cpMvLX[1][1] = (mvScaleVer + dVerX * (xCb + cbWidth − xNb) + dVerY * (yCb − yNb)) (8-369)

– If numCpMv is equal to 3, the third control point vector cpMvLX[2] is derived as follows:

cpMvLX[2][0] = (mvScaleHor + dHorX * (xCb − xNb) + dHorY * (yCb + cbHeight − yNb)) (8-370)

cpMvLX[2][1] = (mvScaleVer + dVerX * (xCb − xNb) + dVerY * (yCb + cbHeight − yNb)) (8-371)

– The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[cpIdx], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[cpIdx] as output, with X being 0 or 1 and cpIdx = 0 .. numCpMv − 1.

8.3.3.4 Derivation process for luma affine control point motion vector predictors

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,

– the reference index of the current coding unit refIdxLX, with X being 0 or 1,

– the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point motion vector predictors mvpCpLX[cpIdx] with X being 0 or 1, and cpIdx = 0 .. numCpMv − 1.

The derivation process for the partitioning information of neighbouring blocks in clause 8.3.3.5 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the output being the availability flags $availableFlag_k$, the luma locations of neighbouring blocks (xNb_k, yNb_k) and the widths nbW_k and the heights nbH_k of the neighbouring coding blocks with k = 0..4.

For the derivation of the control point motion vectors predictor candidate list, cpMvpListLX with X being 0 or 1, the following ordered steps apply:

1. The number of control point motion vector predictor candidates in the list numCpMvpCandLX is set equal to 0.
2. The variable availableFlagLX is set equal to 0.
3. For k ranging from 0 to 4, inclusive, the following applies:
   – When $availableFlag_k$ is equal to 1 and numCpMvpCandLX is less than 2, the following applies:
      – When $PredFlagLX[xNb_k][yNb_k]$ is equal to 1 and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNb_k][yNb_k]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX is set equal to 1, and the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.3.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (xNb_k, yNb_k), the neighbouring luma coding block width and height ($nNbW_k$, $nNbH_k$), and the number of control point motion vectors numCpMv as inputs, the control point motion vector predictor candidates cpMvpLX[cpIdx] with cpIdx = 0 .. numCpMv − 1 as output.
      – When availableFlagLX is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvpLX[cpIdx], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvpLX[cpIdx] as output, with cpIdx = 0 .. numCpMv − 1.
      – If availableFlagLX is equal to 1 and numCpMvpCandLX is equal to 0, the following assignments are made:

        cpMvpListLX[0][0] = cpMvpLX[0] (8-372)

        cpMvpListLX[0][1] = cpMvpLX[1] (8-373)

        cpMvpListLX[0][2] = cpMvpLX[2] (8-374)

        numCpMvpCandLX = 1 (8-375)

      – Otherwise, if availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,
         – numCpMv is equal to 3 and one of the following conditions is true:
            – cpMvpListLX[0][0] is not equal to cpMvpLX[0]
            – cpMvpListLX[0][1] is not equal to cpMvpLX[1]
            – cpMvpListLX[0][2] is not equal to cpMvpLX[2]
         – numCpMv is equal to 2 and one of the following conditions is true:
            – cpMvpListLX[0][0] is not equal to cpMvpLX[0]
            – cpMvpListLX[0][1] is not equal to cpMvpLX[1]

        the following assignments are made:

        cpMvpListLX[1][0] = cpMvpLX[0] (8-376)

        cpMvpListLX[1][1] = cpMvpLX[1] (8-377)

        cpMvpListLX[1][2] = cpMvpLX[2] (8-378)

        numCpMvpCandLX = 2 (8-379)

      – The variable availableFlagLX is set equal to 0.
4. For k ranging from 0 to 4, inclusive, the following applies:
   – When $availableFlag_k$ is equal to 1 and numCpMvpCandLX is less than 2, the following applies:
      – When $PredFlagLY[xNb_k][yNb_k]$ (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNb_k][yNb_k]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX is set equal to 1, and the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.3.3.3 is invoked with the luma coding block location (xCb, yCb), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location (xNb_k, yNb_k), the neighbouring luma coding block width and height ($nNbW_k$, $nNbH_k$), and the number of control point motion vectors numCpMv as inputs, the control point motion vector predictor candidates cpMvpLY[cpIdx] with cpIdx = 0 .. numCpMv − 1 as output.

– When availableFlagLX is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvpLY[cpIdx], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvpLY[cpIdx] as output, with cpIdx = 0 .. numCpMv − 1.

– If availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[0][0] = cpMvpLY[0] (8-380)

cpMvpListLX[0][1] = cpMvpLY[1] (8-381)

cpMvpListLX[0][2] = cpMvpLY[2] (8-382)

numCpMvpCandLX = 1 (8-383)

– Otherwise, if availableFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,

– numCpMv is equal to 3 and one of the following conditions is true:

– cpMvpListLX[0][0] is not equal to cpMvpLY[0]

– cpMvpListLX[0][1] is not equal to cpMvpLY[1]

– cpMvpListLX[0][2] is not equal to cpMvpLY[2]

– numCpMv is equal to 2 and one of the following conditions is true:

– cpMvpListLX[0][0] is not equal to cpMvpLY[0]

– cpMvpListLX[0][1] is not equal to cpMvpLY[1]

the following assignments are made:

cpMvpListLX[1][0] = cpMvpLY[0] (8-384)

cpMvpListLX[1][1] = cpMvpLY[1] (8-385)

cpMvpListLX[1][2] = cpMvpLY[2] (8-386)

numCpMvpCandLX = 2 (8-387)

– The variable availableFlagLX is set equal to 0.

5. When numCpMvpCandLX is less than 2, the following applies

– The derivation process for constructed affine control point motion vector prediction candidate as specified in clause 8.3.3.6 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX, with X being 0 or 1 as inputs, and the availability flag availableConsFlagLX and cpMvpLX[cpIdx] with X being 0 or 1, cpIdx = 0..numCpMv − 1 as outputs.

– If availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[0][0] = cpMvpLX[0] (8-388)

cpMvpListLX[0][1] = cpMvpLX[1] (8-389)

cpMvpListLX[0][2] = cpMvpLX[2] (8-390)

numCpMvpCandLX = 1 (8-391)

– Otherwise, if availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 1, and one of following conditions is true,

– numCpMv is equal to 3 and one of the following conditions is true:

– cpMvpListLX[0][0] is not equal to cpMvpLX[0]

– cpMvpListLX[0][1] is not equal to cpMvpLX[1]

– cpMvpListLX[0][2] is not equal to cpMvpLX[2]

– numCpMv is equal to 2 and one of the following conditions is true:

– cpMvpListLX[0][0] is not equal to cpMvpLY[0]

– cpMvpListLX[0][1] is not equal to cpMvpLY[1]

the following assignments are made:

cpMvpListLX[1][0] = cpMvpLX[0] (8-392)

cpMvpListLX[1][1] = cpMvpLX[1] (8-393)

cpMvpListLX[1][2] = cpMvpLX[2] (8-394)

numCpMvpCandLX = 2 (8-395)

6. When numCpMvpCandLX is less than 2, the derivation process for luma motion vector predictor candidate list as specified in clause 8.3.2.9 is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth and the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1 as output. The cpMvpListLX is appended with candidates from mvpListLX as follows:

```
i = 0
while ( numCpMvpCandLX < 2 ) {
    cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvpListLX[ i ]
    cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvpListLX[ i ]
    cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvpListLX[ i ]
    numCpMvpCandLX++
    i++
}
```
(8-396)

The affine control point motion vector predictor cpMvpLX with X being 0 is derived as follows:

cpMvpLX = cpMvpListLX[mvp_lX_flag[xCb][yCb]] (8-397)

8.3.3.5 Derivation process for neighbouring affine blocks partition information

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

– the availability flags availableFlag$_k$ with k = 0..4,

– the luma locations (xNb_k, yNb_k) of the top-left sample of the neighbouring coding blocks relative to the top-left luma sample of the current picture with k = 0..4,

– the widths nbW_k and the heights nbH_k of the neighbouring coding blocks with k = 0..4.

The neighbouring luma locations (xN_k, yN_k) with k = 0..4 inside the neighbouring luma coding block are set as follows:

(xN_0, yN_0) = (xCb − 1, yCb + cbHeight − 1) (8-398)

(xN_1, yN_1) = (xCb + cbWidth − 1, yCb − 1) (8-399)

(xN_2, yN_2) = (xCb + cbWidth − 1, yCb) (8-400)

(xN_3, yN_3) = (xCb − 1, yCb + cbHeight) (8-401)

(xN_4, yN_4) = (xCb − 1, yCb − 1) (8-402)

The variables availableFlag$_k$, nbW_k and nbH_k with k = 0..4 are derived as follows:

– The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, and the luma location (xNb_k, yNb_k) as inputs, and the output is assigned to the coding block availability flag availableFlag$_k$.

– When availableFlag$_k$ is equal to 1, the following applies:

  – When MotionModelIdc[xNb_k][yNb_k] is equal to 0, availableFlag$_k$ is set equal 0.

  – When availableFlag$_k$ is equal to 1, xNb_k, yNb_k, nbW_k and nbH_k are set equal to CbPosX[xN_k][yN_k], CbPosY[xN_k][yN_k], CbWidth[xN_k][yN_k] and CbHeight[xN_k][yN_k], respectively.

8.3.3.6 Derivation process for constructed affine control point motion vector prediction candidates

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,

– the reference index of the current prediction unit partition refIdxLX, with X being 0 or 1,

Output of this process are:

– the availability flag of the constructed affine control point motion vector prediction candidiates availableConsFlagLX with X being 0 or 1,

– the constructed affine control point motion vector prediction candidiates cpMvLX[cpIdx] with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLX[0] and the availability flag availableFlagLX[0] are derived in the following ordered steps:

1. The sample locations ($xNbB_2$, $yNbB_2$), ($xNbB_3$, $yNbB_3$) and ($xNbA_2$, $yNbA_2$) are set equal to (xCb − 1, yCb − 1), (xCb , yCb − 1) and (xCb − 1, yCb), respectively.
2. The availability flag availableFlagLX[0] is set equal to 0 and both components of cpMvLX[0] are set equal to 0.
3. The following applies for (xNbTL, yNbTL) withTL being replaced by B_2, B_3, and A_2:
   – The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNbY, yNbY) set equal to (xNbTL, yNbTL) as inputs, and the output is assigned to the coding block availability flag availableTL.
   – When availableTL is equal to TRUE and availableFlagLX[0] is equal to 0, the following applies:
     – If PredFlagLX[xNbTL][yNbTL] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNbTL][yNbTL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[0] is set equal to 1 and the following assignments are made:

       cpMvLX[0] = MvLX[xNbTL][yNbTL] (8-403)

     – Otherwise, when PredFlagLY[xNbTL][yNbTL] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNbTL][yNbTL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[0] is set equal to 1 and the following assignments are made:

       cpMvLX[0] = MvLY[xNbTL][yNbTL] (8-404)

     – When availableFlagLX[0] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[0], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[0] as output.

The second (top-right) control point motion vector cpMvLX[1] and the availability flag availableFlagLX[1] are derived in the following ordered steps:

1. The sample locations ($xNbB_1$, $yNbB_1$) and ($xNbB_0$, $yNbB_0$) are set equal to (xCb + cbWidth − 1, yCb − 1) and (xCb + cbWidth, yCb − 1), respectively.
2. The availability flag availableFlagLX[1] is set equal to 0 and both components of cpMvLX[1] are set equal to 0.
3. The following applies for (xNbTR, yNbTR) withTR being replaced by B_1 and B_0:
   – The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNbY, yNbY) set equal to (xNbTR, yNbTR) as inputs, and the output is assigned to the coding block availability flag availableTR.
   – When availableTR is equal to TRUE and availableFlagLX[0] is equal to 0, the following applies:
     – If PredFlagLX[xNbTR][yNbTR] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNbTR][yNbTR]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[1] is set equal to 1 and the following assignments are made:

cpMvLX[1] = MvLX[xNbTR][yNbTR] (8-405)

– Otherwise, when PredFlagLY[xNbTR][yNbTR] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNbTR][yNbTR]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[1] is set equal to 1 and the following assignments are made:

cpMvLX[1] = MvLY[xNbTR][yNbTR] (8-406)

– When availableFlagLX[1] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[1], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[1] as output.

The third (bottom-left) control point motion vector cpMvLX[2] and the availability flag availableFlagLX[2] are derived in the following ordered steps:

1. The sample locations ($xNbA_1$, $yNbA_1$) and ($xNbA_0$, $yNbA_0$) are set equal to (xCb − 1, yCb + cbHeight − 1) and (xCb − 1, yCb + cbHeight), respectively.
2. The availability flag availableFlagLX[2] is set equal to 0 and both components of cpMvLX[2] are set equal to 0.
3. The following applies for (xNbBL, yNbBL) with BL being replaced by A_1 and A_0:
   – The availability derivation process for a coding block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location (xCb, yCb), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location (xNbY, yNbY) set equal to (xNbBL, yNbBL) as inputs, and the output is assigned to the coding block availability flag availableBL.
   – When availableBL is equal to TRUE and availableFlagLX[2] is equal to 0, the following applies:
     – If PredFlagLX[xNbBL][yNbBL] is equal to 1, and DiffPicOrderCnt(RefPicListX[RefIdxLX[xNbBL][yNbBL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[2] is set equal to 1 and the following assignments are made:

       cpMvLX[2] = MvLX[xNbBL][yNbBL] (8-407)

     – Otherwise, when PredFlagLY[xNbBL][yNbBL] (with Y = !X) is equal to 1 and DiffPicOrderCnt(RefPicListY[RefIdxLY[xNbBL][yNbBL]], RefPicListX[refIdxLX]) is equal to 0, availableFlagLX[2] is set equal to 1 and the following assignments are made:

       cpMvLX[2] = MvLY[xNbBL][yNbBL] (8-408)

     – When availableFlagLX[2] is equal to 1, the rounding process for motion vectors as specified in clause 8.3.2.14 is invoked with mvX set equal to cpMvLX[2], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[2] as output.

The variables availableConsFlagLX, cpMvLX[cpIdx] are derived as follows:

– The variables log2CbW and log2CbH are derived as follows:

log2CbW = Log2(cbWidth) (8-409)

log2CbH = Log2(cbHeight) (8-410)

– If availableFlagLX[0] is equal to 1 and availableFlagLX[1] is equal to 1 and availableFlagLX[2] is equal to 1, availableConsFlagLX is set equal to 1

– Otherwise, if availableFlagLX[0] is equal to 1, and availableFlagLX[1] is equal to 1, and MotionModelIdc[xCb][yCb] is equal to 2, availableConsFlagLX is set equal to 1 and the following applies:

  – The third control point motion vector cpMvLX[2] is derived as follows:

cpMvLX[2][0] = (cpMvLX[0][0] << 7) − (8-411)
((cpMvLX[1][1] − cpMvLX[0][1]) << (7 + log2CbH − log2CbW))

cpMvLX[2][1] = (cpMvLX[0][1] << 7) + (8-412)
((cpMvLX[1][0] − cpMvLX[0][0]) << (7 + log2CbH − log2CbW))

  – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[2], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[2] as output, with X being 0 or 1.

  – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[2], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[2] as output, with X being 0 or 1.

– Otherwise, if availableFlagLX[0] is equal to 1 and availableFlagLX[2] is equal to 1, availableConsFlagLX is set equal to 1, and the following applies:
  – The second control point motion vector cpMvLX[1] is derived as follow:

$$cpMvLX[\,1\,][\,0\,] = (cpMvLX[\,0\,][\,0\,] \ll 7) + ((cpMvLX[\,2\,][\,1\,] - cpMvLX[\,0\,][\,1\,]) \ll (7 + log2CbH - log2CbW)) \quad (8\text{-}413)$$

$$cpMvLX[\,1\,][\,1\,] = (cpMvLX[\,0\,][\,1\,] \ll 7) - ((cpMvLX[\,2\,][\,0\,] - cpMvLX[\,0\,][\,0\,]) \ll (7 + log2CbH - log2CbW)) \quad (8\text{-}414)$$

  – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[1], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[1] as output, with X being 0 or 1.
  – The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to cpMvLX[1], rightShift set equal to 2, and leftShift set equal to 2 as inputs and the rounded cpMvLX[1] as output, with X being 0 or 1.
– Otherwise, availableConsFlagLX is set equal to 0.

8.3.3.7 Derivation process for motion vector arrays from affine control point motion vectors

Inputs to this process are:

– a luma location (xCb, yCb) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
– two variables cbWidth and cbHeight specifying the width and the height of the luma coding block,
– the number of control point motion vectors numCpMv,
– the control point motion vectors cpMvL0[cpIdx] and cpMvL1[cpIdx], with cpIdx = 0..numCpMv − 1,
– the prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are:

– the luma subblock motion vector array mvLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1,
– the chroma subblock motion vector array mvCLX[xSbIdx][ySbIdx] with xSbIdx = 0..(cbWidth >> 2) − 1, ySbIdx = 0 .. (cbHeight >> 2) − 1, X being 0 and 1.

When predFlagLX is equal to 1, the following applies for X being 0 and 1:

– The variables log2CbW and log2CbH are derived as follows:

$$log2CbW = Log2(cbWidth) \quad (8\text{-}415)$$

$$log2CbH = Log2(cbHeight) \quad (8\text{-}416)$$

– The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

$$mvScaleHor = cpMvLX[\,0\,][\,0\,] \ll 7 \quad (8\text{-}417)$$

$$mvScaleVer = cpMvLX[\,0\,][\,1\,] \ll 7 \quad (8\text{-}418)$$

$$dHorX = (cpMvLX[\,1\,][\,0\,] - cpMvLX[\,0\,][\,0\,]) \ll (7 - log2CbW) \quad (8\text{-}419)$$

$$dVerX = (cpMvLX[\,1\,][\,1\,] - cpMvLX[\,0\,][\,1\,]) \ll (7 - log2CbW) \quad (8\text{-}420)$$

– If numCpMv is equal to 3, the variable dHorY and dVerY are derived as follow:

$$dHorY = (cpMvLX[\,2\,][\,0\,] - cpMvLX[\,0\,][\,0\,]) \ll (7 - log2CbH) \quad (8\text{-}421)$$

$$dVerY = (cpMvLX[\,2\,][\,1\,] - cpMvLX[\,0\,][\,1\,]) \ll (7 - log2CbH) \quad (8\text{-}422)$$

– Otherwise (numCpMv is equal to 2), the variable dHorY and dVerY are derived as follows:

$$dHorY = - dVerX \quad (8\text{-}423)$$

$$dVerY = dHorX \quad (8\text{-}424)$$

– For xSbIdx = 0..(cbWidth >> 2) − 1 and ySbIdx = 0..(cbHeight >> 2) − 1, the following applies:

– The luma motion vector mvLX[xSbIdx][ySbIdx] is derived as follows :

$$xPosCb = 2 + (xSbIdx << 2) \quad (8\text{-}425)$$

$$yPosCb = 2 + (ySbIdx << 2) \quad (8\text{-}426)$$

$$mvLX[xSbIdx][ySbIdx][0] = (mvScaleHor + dHorX * xPosCb + dHorY * yPosCb) \quad (8\text{-}427)$$

$$mvLX[xSbIdx][ySbIdx][1] = (mvScaleVer + dVerX * xPosCb + dVerY * yPosCb) \quad (8\text{-}428)$$

– The rounding process for motion vectors as specified in clause 8.3.2.14 is invoked the with mvX set equal to mvLX[xSbIdx][ySbIdx], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded mvLX[xSbIdx][ySbIdx] as output.

– The derivation process for chroma motion vectors in clause 8.3.2.13 is invoked with mvLX[xSbIdx][ySbIdx] as input, and the chroma motion vector mvCLX[xSbIdx][ySbIdx] as output.

8.3.4 Decoding process for inter blocks

8.3.4.1 General

This process is invoked when decoding coding unit whose CuPredMode[xCb][yCb] is not equal to MODE_INTRA.

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,

– the luma motion vectors in 1/16 fractional-sample accuracy mvL0[xSbIdx][ySbIdx] and mvL1[xSbIdx][ySbIdx] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,

– the chroma motion vectors mvCL0 and mvCL1,

– the reference indices refIdxL0 and refIdxL1,

– the prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are:

– an (cbWidth)x(cbHeight) array $predSamples_L$ of luma prediction samples,

– an (cbWidth / 2)x(cbHeight / 2) array $predSamples_{Cb}$ of chroma prediction samples for the component Cb,

– an (cbWidth / 2)x(cbHeight / 2) array $predSamples_{Cr}$ of chroma prediction samples for the component Cr.

Let $predSamplesL0_L$ and $predSamplesL1_L$ be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, $predSamplesL0_{Cb}$, $predSamplesL1_{Cb}$, $predSamplesL0_{Cr}$ and $predSamplesL1_{Cr}$ be (cbWidth / 2)x(cbHeight / 2) arrays of predicted chroma sample values.

The width and the height of the current luma coding sublock subCbWidth, subCbHeight are derived as follows:

$$sbWidth = cbWidth / numSbX \quad (8\text{-}429)$$

$$sbHeight = cbHeight / numSbY \quad (8\text{-}430)$$

For each coding subblock at subblock index (xSbIdx, ySbIdx) with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1, the following applies:

– The luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture is derived as follows:

$$(xSb, ySb) = (xCb + xSbIdx * sbWidth, yCb + ySbIdx * sbHeight) \quad (8\text{-}431)$$

– For X being each of 0 and 1, when predFlagLX is equal to 1, the following applies:

- The reference picture consisting of an ordered two-dimensional array $refPicLX_L$ of luma samples and two ordered two-dimensional arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$ of chroma samples is derived by invoking the process specified in clause 8.3.4.2 with refIdxLX as input.
- The array $predSamplesLX_L$ and the arrays $predSamplesLX_{Cb}$ and $predSamplesLX_{Cr}$ are derived by invoking the fractional sample interpolation process specified in clause 8.3.4.3 with the luma location (xCb, yCb), the luma coding subblock width sbWidth, the luma coding subblock height sbHeight, the motion vectors mvLX[xSb][ySb], mvCLX[xSb][ySb], and the reference arrays $refPicLX_L$, $refPicLX_{Cb}$, and $refPicLX_{Cr}$ as inputs.

– The prediction samples inside the current luma coding subblock, $predSamples_L[x_L + xSb][y_L + ySb]$ with $x_L = 0..sbWidth - 1$ and $y_L = 0..sbHeight - 1$, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays $predSamplesL0_L$ and $predSamplesL1_L$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx set equal to 0 as inputs.

– The prediction samples inside the current chroma component Cb coding block, $predSamples_{Cb}[x_C + xCb / 2][y_C + yCb / 2]$ with $x_C = 0..cbWidth / 2 - 1$ and $y_C = 0..cbHeight / 2 - 1$, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays $predSamplesL0_{Cb}$ and $predSamplesL1_{Cb}$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx set equal to 1 as inputs.

– The prediction samples inside the current chroma component Cr coding block, $predSamples_{Cr}[x_C + xCb / 2][y_C + yCb / 2]$ with $x_C = 0..cbWidth / 2 - 1$ and $y_C = 0..cbHeight / 2 - 1$, are derived by invoking the weighted sample prediction process specified in clause 8.3.4.4 with the coding block width nCbW set equal to cbWidth / 2, the coding block height nCbH set equal to cbHeight / 2, the sample arrays $predSamplesL0_{Cr}$ and $predSamplesL1_{Cr}$, and the variables predFlagL0, predFlagL1, refIdxL0, refIdxL1 and cIdx set equal to 2 as inputs.

– When MotionModelIdc[xCb][yCb] is equal to 0, the following assignments are made for $x = 0..sbWidth - 1$ and $y = 0..sbHeight - 1$:

$$MvL0[xSb + x][ySb + y] = mvL0[xSbIdx][ySbIdx] \quad (8\text{-}432)$$

$$MvL1[xSb + x][ySb + y] = mvL1[xSbIdx][ySbIdx] \quad (8\text{-}433)$$

$$RefIdxL0[xSb + x][ySb + y] = refIdxL0 \quad (8\text{-}434)$$

$$RefIdxL1[xSb + x][ySb + y] = refIdxL1 \quad (8\text{-}435)$$

$$PredFlagL0[xSb + x][ySb + y] = predFlagL0 \quad (8\text{-}436)$$

$$PredFlagL1[xSb + x][ySb + y] = predFlagL1 \quad (8\text{-}437)$$

8.3.4.2 Reference picture selection process

Input to this process is a reference index refIdxLX.

Output of this process is a reference picture consisting of a two-dimensional array of luma samples $refPicLX_L$ and two two-dimensional arrays of chroma samples $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

The output reference picture RefPicListX[refIdxLX] consists of a pic_width_in_luma_samples by pic_height_in_luma_samples array of luma samples $refPicLX_L$ and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

The reference picture sample arrays $refPicLX_L$, $refPicLX_{Cb}$ and $refPicLX_{Cr}$ correspond to decoded sample arrays S_L, S_{Cb} and S_{Cr} derived in clause 8.5 for a previously-decoded picture.

8.3.4.3 Fractional sample interpolation process

8.3.4.3.1 General

Inputs to this process are:

– a luma location (xSb, ySb) specifying the top-left sample of the current coding subblock relative to the top-left luma sample of the current picture,

– a variable sbWidth specifying the width of the current coding subblock in luma samples,

– a variable sbHeight specifying the height of the current coding subblock in luma samples,

– a luma motion vector mvLX given in 1/16-luma-sample units,

– a chroma motion vector mvCLX given in 1/32-chroma-sample units,

– the selected reference picture sample array $refPicLX_L$ and the arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

Outputs of this process are:

– an (sbWidth)x(sbHeight) array $predSamplesLX_L$ of prediction luma sample values,

– two (sbWidth / 2)x(sbHeight / 2) arrays $predSamplesLX_{Cb}$ and $predSamplesLX_{Cr}$ of prediction chroma sample values.

Let ($xInt_L$, $yInt_L$) be a luma location given in full-sample units and ($xFrac_L$, $yFrac_L$) be an offset given in 1/16-sample units. These variables are used only in this clause for specifying fractional-sample locations inside the reference sample arrays $refPicLX_L$, $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

For each luma sample location ($x_L = 0..sbWidth - 1$, $y_L = 0..sbHeight - 1$) inside the prediction luma sample array $predSamplesLX_L$, the corresponding prediction luma sample value $predSamplesLX_L[x_L][y_L]$ is derived as follows:

– The variables $xInt_L$, $yInt_L$, $xFrac_L$ and $yFrac_L$ are derived as follows:

$$xInt_L = xSb + (mvLX[0] >> 4) + x_L \quad (8\text{-}438)$$

$$yInt_L = ySb + (mvLX[1] >> 4) + y_L \quad (8\text{-}439)$$

$$xFrac_L = mvLX[0] \& 15 \quad (8\text{-}440)$$

$$yFrac_L = mvLX[1] \& 15 \quad (8\text{-}441)$$

– The prediction luma sample value $predSamplesLX_L[x_L][y_L]$ is derived by invoking the process specified in clause 8.3.4.3.2 with ($xInt_L$, $yInt_L$), ($xFrac_L$, $yFrac_L$) and $refPicLX_L$ as inputs.

Let ($xInt_C$, $yInt_C$) be a chroma location given in full-sample units and ($xFrac_C$, $yFrac_C$) be an offset given in 1/32 sample units. These variables are used only in this clause for specifying general fractional-sample locations inside the reference sample arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

For each chroma sample location ($x_C = 0..sbWidth / 2 - 1$, $y_C = 0..sbHeight / 2 - 1$) inside the prediction chroma sample arrays $predSamplesLX_{Cb}$ and $predSamplesLX_{Cr}$, the corresponding prediction chroma sample values $predSamplesLX_{Cb}[x_C][y_C]$ and $predSamplesLX_{Cr}[x_C][y_C]$ are derived as follows:

– The variables $xInt_C$, $yInt_C$, $xFrac_C$ and $yFrac_C$ are derived as follows:

$$xInt_C = (xSb / SubWidthC) + (mvCLX[0] >> 5) + x_C \quad (8\text{-}442)$$

$$yInt_C = (ySb / SubHeightC) + (mvCLX[1] >> 5) + y_C \quad (8\text{-}443)$$

$$xFrac_C = mvCLX[0] \& 31 \quad (8\text{-}444)$$

$$yFrac_C = mvCLX[1] \& 31 \quad (8\text{-}445)$$

– The prediction sample value $predSamplesLX_{Cb}[x_C][y_C]$ is derived by invoking the process specified in clause 8.3.4.3.3 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cb}$ as inputs.

– The prediction sample value $predSamplesLX_{Cr}[x_C][y_C]$ is derived by invoking the process specified in clause 8.3.4.3.3 with ($xInt_C$, $yInt_C$), ($xFrac_C$, $yFrac_C$) and $refPicLX_{Cr}$ as inputs.

8.3.4.3.2 Luma sample interpolation process

Inputs to this process are:

– a luma location in full-sample units ($xInt_L$, $yInt_L$),

– a luma location in fractional-sample units ($xFrac_L$, $yFrac_L$),

– the luma reference sample array $refPicLX_L$.

Output of this process is a predicted luma sample value $predSampleLX_L$

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min(4, $BitDepth_Y$ − 8), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max(2, 14 − $BitDepth_Y$).

– The variable picW is set equal to pic_width_in_luma_samples and the variable picH is set equal to pic_height_in_luma_samples.

The luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p equal to $xFrac_L$ or $yFrac_L$ are specified in Table 8-8.

The luma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..7:

$$xInt_i = Clip3(0, picW − 1, xInt_L + i − 3) \quad (8\text{-}446)$$

$$yInt_i = Clip3(0, picH − 1, yInt_L + i − 3) \quad (8\text{-}447)$$

The predicted luma sample value $predSampleLX_L$ is derived as follows:

– If both $xFrac_L$and $yFrac_L$ are equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = refPicLX_L[xInt_3][yInt_3] << shift3 \quad (8\text{-}448)$$

– Otherwise, if $xFrac_L$ is not equal to 0 and $yFrac_L$ is equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_3]) >> shift1 \quad (8\text{-}449)$$

– Otherwise, if $xFrac_L$ is equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (\sum_{i=0}^{7} f_L[yFrac_L][i] * refPicLX_L[xInt_3][yInt_i]) >> shift1 \quad (8\text{-}450)$$

– Otherwise, if $xFrac_L$ is not equal to 0 and $yFrac_L$ is not equal to 0, the value of $predSampleLX_L$ is derived as follows:

  – The sample array temp[n] with n = 0..7, is derived as follows:

$$temp[n] = (\sum_{i=0}^{7} f_L[xFrac_L][i] * refPicLX_L[xInt_i][yInt_n]) >> shift1 \quad (8\text{-}451)$$

  – The predicted luma sample value $predSampleLX_L$ is derived as follows:

$$predSampleLX_L = (\sum_{i=0}^{7} f_L[yFrac_L][i] * temp[i]) >> shift2 \quad (8\text{-}452)$$

Table 8-8 – Specification of the luma interpolation filter coefficients $f_L[p]$ for each 1/16 fractional sample position p.

Fractional sample position p	interpolation filter coefficients							
	$f_L[p][0]$	$f_L[p][1]$	$f_L[p][2]$	$f_L[p][3]$	$f_L[p][4]$	$f_L[p][5]$	$f_L[p][6]$	$f_L[p][7]$
1	0	1	−3	63	4	−2	1	0
2	−1	2	−5	62	8	−3	1	0
3	−1	3	−8	60	13	−4	1	0
4	−1	4	−10	58	17	−5	1	0
5	−1	4	−11	52	26	−8	3	−1
6	−1	3	−9	47	31	−10	4	−1
7	−1	4	−11	45	34	−10	4	−1
8	−1	4	−11	40	40	−11	4	−1
9	−1	4	−10	34	45	−11	4	−1
10	−1	4	−10	31	47	−9	3	−1
11	−1	3	−8	26	52	−11	4	−1
12	0	1	−5	17	58	−10	4	−1
13	0	1	−4	13	60	−8	3	−1
14	0	1	−3	8	62	−5	2	−1
15	0	1	−2	4	63	−3	1	0

8.3.4.3.3 Chroma sample interpolation process

Inputs to this process are:

– a chroma location in full-sample units ($xInt_C$, $yInt_C$),

– a chroma location in 1/32 fractional-sample units ($xFrac_C$, $yFrac_C$),

– the chroma reference sample array $refPicLX_C$.

Output of this process is a predicted chroma sample value $predSampleLX_C$

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min(4, $BitDepth_C$ − 8), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max(2, 14 − $BitDepth_C$).

– The variable $picW_C$ is set equal to pic_width_in_luma_samples / SubWidthC and the variable $picH_C$ is set equal to pic_height_in_luma_samples / SubHeightC.

The luma interpolation filter coefficients $f_C[p]$ for each 1/32 fractional sample position p equal to $xFrac_C$ or $yFrac_C$ are specified in Table 8-9.

The chroma locations in full-sample units ($xInt_i$, $yInt_i$) are derived as follows for i = 0..3:

$$xInt_i = Clip3(0, picW_C - 1, xInt_C + i - 1) \quad (8\text{-}453)$$

$$yInt_i = Clip3(0, picH_C - 1, yInt_C + i - 1) \quad (8\text{-}454)$$

The predicted chroma sample value $predSampleLX_C$ is derived as follows:

– If both $xFrac_C$ and $yFrac_C$ are equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = refPicLX_C[xInt_1][yInt_1] << shift3 \quad (8\text{-}455)$$

– Otherwise if $xFrac_C$ is not equal to 0 and $yFrac_C$ is equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = (\sum_{i=0}^{3} f_C[xFrac_C][i] * refPicLX_C[xInt_i][yInt_1]) >> shift1 \quad (8\text{-}456)$$

– Otherwise if $xFrac_C$ is equal to 0 and $yFrac_C$ is not equal to 0, the value of $predSampleLX_C$ is derived as follows:

$$predSampleLX_C = (\sum_{i=0}^{3} f_C[yFrac_C][i] * refPicLX_C[xInt_1][yInt_i]) >> shift1 \quad (8\text{-}457)$$

– Otherwise if $xFrac_C$ is not equal to 0 and $yFrac_C$ is not equal to 0, the value of $predSampleLX_C$ is derived as follows:

  – The sample array temp[n] with n = 0..3, is derived as follows:

$$temp[n] = (\sum_{i=0}^{3} f_C[xFrac_C][i] * refPicLX_C[xInt_i][yInt_n]) >> shift1 \quad (8\text{-}458)$$

  – The predicted chroma sample value $predSampleLX_C$ is derived as follows:

$$\begin{aligned} predSampleLX_C = (& f_C[yFrac_C][0] * temp[0] + \\ & f_C[yFrac_C][1] * temp[1] + \\ & f_C[yFrac_C][2] * temp[2] + \\ & f_C[yFrac_C][3] * temp[3]) >> shift2 \end{aligned} \quad (8\text{-}459)$$

Table 8-9 – Specification of the chroma interpolation filter coefficients $f_C[p]$ for each 1/32 fractional sample position p.

Fractional sample position p	interpolation filter coefficients			
	$f_C[p][0]$	$f_C[p][1]$	$f_C[p][2]$	$f_C[p][3]$
1	−1	63	2	0
2	−2	62	4	0
3	-2	60	7	−1
4	−2	58	10	−2
5	−3	57	12	−2
6	−4	56	14	−2
7	−4	55	15	−2
8	−4	54	16	−2
9	−5	53	18	−2
10	−6	52	20	−2
11	−6	49	24	−3
12	−6	46	28	−4
13	−5	44	29	−4
14	−4	42	30	−4
15	−4	39	33	−4
16	−4	36	36	−4
17	−4	33	39	−4
18	−4	30	42	−4
19	−4	29	44	−5
20	−4	28	46	−6
21	−3	24	49	−6
22	−2	20	52	−6
23	−2	18	53	−5
24	−2	16	54	−4
25	−2	15	55	−4
26	−2	14	56	−4
27	−2	12	57	−3
28	−2	10	58	−2
29	−1	7	60	−2
30	0	4	62	−2
31	0	2	63	−1

8.3.4.4 Weighted sample prediction process

Inputs to this process are:

– two variables nCbW and nCbH specifying the width and the height of the current coding block,

– two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,

– the prediction list utilization flags predFlagL0 and predFlagL1,

– the reference indices refIdxL0 and refIdxL1,

– a variable cIdx specifying colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to $BitDepth_Y$.

– Otherwise, bitDepth is set equal to $BitDepth_C$.

Variables shift1, shift2, offset1 and offset2 are derived as follows:

– The variable shift1 is set equal to Max(2, 14 − bitDepth) and the variable shift2 is set equal to Max(3, 15 − bitDepth).

– The variable offset1 is set equal to 1 << (shift1 − 1).

– The variable offset2 is set equal to 1 << (shift2 − 1).

Depending on the values of predFlagL0 and predFlagL1, the prediction samples pbSamples[x][y] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

– If predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction sample values are derived as follows:

pbSamples[x][y] = Clip3(0, (1 << bitDepth) − 1, (predSamplesL0[x][y] + offset1) >> shift1) (8-460)

– Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

pbSamples[x][y] = Clip3(0, (1 << bitDepth) − 1, (predSamplesL1[x][y] + offset1) >> shift1) (8-461)

– Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

pbSamples[x][y] = Clip3(0, (1 << bitDepth) − 1,
(predSamplesL0[x][y] + predSamplesL1[x][y] + offset2) >> shift2) (8-462)

8.3.5 Decoding process for the residual signal of coding blocks coded in inter prediction mode

Inputs to this process are:

– a sample location (xTb0, yTb0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture,

– a variable nTbW specifying the width of the current transform block,

– a variable nTbH specifying the height of the current transform block,

– a variable cIdx specifying the colour component of the current block.

Output of this process is an (nTbW)x(nTbH) array resSamples.

The maximum transform block size maxTbSize is derived as follows:

maxTbSize = (cIdx == 0) ? MaxTbSizeY : MaxTbSizeY / 2 (8-463)

The luma sample location is derived as follows:

(xTbY, yTbY) = (cIdx == 0) ? (xTb0, yTb0) : (xTb0 / 2, yTb0 / 2) (8-464)

Depending on maxTbSize, the following applies:

– If nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables newTbW and newTbH are derived as follows:

newTbW = (nTbW > maxTbSize) ? (nTbW / 2) : nTbW (8-465)

newTbH = (nTbH > maxTbSize) ? (nTbH / 2) : nTbH (8-466)

2. The decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

3. When nTbW is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture .

4. When nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0, yTb0 + newTbH), the transform block width nTbW set equal to newTbW and the height nTbH set equal

to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

5. Wwhen nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0 + newTbH), the transform block width nTbW set equal to newTbW and height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

– Otherwise, the scaling and transformation process as specified in clause 8.4.2 is invoked with the luma location (xTbY, yTbY), the variable cIdx, the transform width nTbW and the transform height nTbH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.

8.4 Scaling, transformation and array construction process

8.4.1 Derivation process for quantization parameters

Inputs to this process are:

– a luma location (xCb, yCb) specifying the top-left luma sample of the current coding block relative to the top-left luma sample of the current picture,

– a variable cbWidth specifying the width of the current coding block in luma samples,

– a variable cbHeight specifying the height of the current coding block in luma samples,

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed.

In this process, the luma quantization parameter Qp'_Y and the chroma quantization parameters Qp'_{Cb} and Qp'_{Cr} are derived.

The luma location (xQg, yQg), specifies the top-left luma sample of the current quantization group relative to the top left luma sample of the current picture. The horizontal and vertical positions xQg and yQg are set equal to CuQgTopLeftX and CuQgTopLeftY, respectively.

NOTE – : The currect quantization group is a rectangluar region inside a coding tree block that shares the same qP_{Y_PRED}. Its width and height are equal to the width and height of the coding tree node of which the top-left luma sample position is assigned to the variables CuQgTopLeftX and CuQgTopLeftY.

When treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the predicted luma quantization parameter qP_{Y_PRED} is derived by the following ordered steps:

1. The variable qP_{Y_PREV} is derived as follows:
   – If one or more of the following conditions are true, qP_{Y_PREV} is set equal to $SliceQp_Y$:
     – The current quantization group is the first quantization group in a slice.
     – The current quantization group is the first quantization group in a tile.
   – Otherwise, qP_{Y_PREV} is set equal to the luma quantization parameter Qp_Y of the last luma coding unit in the previous quantization group in decoding order.
2. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring location (xNbY, yNbY) set equal to (xQg − 1, yQg) as inputs, and the output is assigned to availableA. The variable qP_{Y_A} is derived as follows:
   – If one or more of the following conditions are true, qP_{Y_A} is set equal to qP_{Y_PREV}:
     – availableA is equal to FALSE.
     – the CTB address ctbAddrA of the CTB containing the luma coding block covering the luma location (xQg − 1, yQg) is not equal to CtbAddrInTs, where ctbAddrA is derived as follows:

```
xTmp = ( xQg − 1 ) >> MinTbLog2SizeY
yTmp = yQg >> MinTbLog2SizeY
minTbAddrA = MinTbAddrZs[ xTmp ][ yTmp ]
ctbAddrA = minTbAddrA >> ( 2 * ( CtbLog2SizeY − MinTbLog2SizeY ) )        (8-467)
```

– Otherwise, qP_{Y_A} is set equal to the luma quantization parameter Qp_Y of the coding unit containing the luma coding block covering (xQg − 1, yQg).

3. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location (xCurr, yCurr) set equal to (xCb, yCb) and the neighbouring location (xNbY, yNbY) set equal to (xQg, yQg − 1) as inputs, and the output is assigned to availableB. The variable qP_{Y_B} is derived as follows:

   – If one or more of the following conditions are true, qP_{Y_B} is set equal to qP_{Y_PREV}:

      – availableB is equal to FALSE.

      – the CTB address ctbAddrB of the CTB containing the luma coding block covering the luma location (xQg, yQg − 1) is not equal to CtbAddrInTs, where ctbAddrB is derived as follows:

```
xTmp = xQg >> MinTbLog2SizeY
yTmp = ( yQg − 1 ) >> MinTbLog2SizeY
minTbAddrB = MinTbAddrZs[ xTmp ][ yTmp ]
ctbAddrB = minTbAddrB >> ( 2 * ( CtbLog2SizeY − MinTbLog2SizeY ) )
```
(8-468)

   – Otherwise, qP_{Y_B} is set equal to the luma quantization parameter Qp_Y of the coding unit containing the luma coding block covering (xQg, yQg − 1).

4. The predicted luma quantization parameter qP_{Y_PRED} is derived as follows:

$$qP_{Y\_PRED} = (qP_{Y\_A} + qP_{Y\_B} + 1) >> 1 \quad (8\text{-}469)$$

The variable Qp_Y is derived as follows:

$$Qp_Y = ((qP_{Y\_PRED} + CuQpDeltaVal + 64 + 2 * QpBdOffset_Y)\%(64 + QpBdOffset_Y)) − QpBdOffset_Y \quad (8\text{-}470)$$

The luma quantization parameter Qp'_Y is derived as follows:

$$Qp'_Y = Qp_Y + QpBdOffset_Y \quad (8\text{-}471)$$

[Ed. (BB): Modify highlighted sections when tiles without slices are incorporated]

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the following applies:

– When treeType is equal to DUAL_TREE_CHROMA, the variable Qp_Y is set equal to the luma quantization parameter Qp_Y of the luma coding unit that covers the luma location (xCb + cbWidth / 2, yCb + cbHeight / 2).

– The variables qP_{Cb} and qP_{Cr} are derived as follows:

$$qPi_{Cb} = Clip3(−QpBdOffset_C, 69, Qp_Y + pps\_cb\_qp\_offset + slice\_cb\_qp\_offset) \quad (8\text{-}472)$$

$$qPi_{Cr} = Clip3(−QpBdOffset_C, 69, Qp_Y + pps\_cr\_qp\_offset + slice\_cr\_qp\_offset) \quad (8\text{-}473)$$

– If ChromaArrayType is equal to 1, the variables qP_{Cb} and qP_{Cr} are set equal to the value of Qp_C as specified in Table 8-10 based on the index qPi equal to qPi_{Cb} and qPi_{Cr}, respectively.

– Otherwise, the variables qP_{Cb} and qP_{Cr} are set equal to Min(qPi, 63), based on the index qPi equal to qPi_{Cb} and qPi_{Cr}, respectively.

– The chroma quantization parameters for the Cb and Cr components, Qp'_{Cb} and Qp'_{Cr}, are derived as follows:

$$Qp'_{Cb} = qP_{Cb} + QpBdOffset_C \quad (8\text{-}474)$$

$$Qp'_{Cr} = qP_{Cr} + QpBdOffset_C \quad (8\text{-}475)$$

Table 8-10 – Specification of Qp_C as a function of qPi for ChromaArrayType equal to 1

qPi	< 30	30	31	32	33	34	35	36	37	38	39	40	41	42	43	> 43
Qp_C	= qPi	29	30	31	32	33	33	34	34	35	35	36	36	37	37	= qPi − 6

8.4.2 Scaling and transformation process

Inputs to this process are:

– a luma location (xTbY, yTbY) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the current picture,

– a variable cIdx specifying the colour component of the current block,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height.

Output of this process is the (nTbW)x(nTbH) array of residual samples resSamples[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables bitDepth, bdShift and tsShift are derived as follows:

$$bitDepth = (cIdx == 0) ? BitDepth_Y : BitDepth_C \quad (8\text{-}476)$$

$$bdShift = Max(20 - bitDepth, 0) \quad (8\text{-}477)$$

$$tsShift = 5 + ((Log2(nTbW) + Log2(nTbH)) / 2) \quad (8\text{-}478)$$

The (nTbW)x(nTbH) array of residual samples resSamples is derived as follows:

1. The scaling process for transform coefficients as specified in clause 8.4.3 is invoked with the transform block location (xTbY, yTbY), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx and the bit depth of the current colour component bitDepth as inputs, and the output is an (nTbW)x(nTbH) array of scaled transform coefficients d.
2. The (nTbW)x(nTbH) array of residual samples r is derived as follows:
   – If transform_skip_flag[xTbY][yTbY][cIdx] is equal to 1, the residual sample array values r[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

$$r[x][y] = d[x][y] << tsShift \quad (8\text{-}479)$$

   – Otherwise (transform_skip_flag[xTbY][yTbY][cIdx] is equal to 0), the transformation process for scaled transform coefficients as specified in clause 8.4.4.1 is invoked with the transform block location (xTbY, yTbY), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx and the (nTbW)x(nTbH) array of scaled transform coefficients d as inputs, and the output is an (nTbW)x(nTbH) array of residual samples r.
3. The residual samples resSamples[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

$$resSamples[x][y] = (r[x][y] + (1 << (bdShift - 1))) >> bdShift \quad (8\text{-}480)$$

8.4.3 Scaling process for transform coefficients

Inputs to this process are:

– a luma location (xTbY, yTbY) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the current picture,

– a variable nTbW specifying the transform block width,

– a variable nTbH specifying the transform block height,

– a variable cIdx specifying the colour component of the current block,

– a variable bitDepth specifying the bit depth of the current colour component.

Output of this process is the (nTbW)x(nTbH) array d of scaled transform coefficients with elements d[x][y].

The quantization parameter qP is derived as follows:

– If cIdx is equal to 0, the following applies:

$$qP = Qp'_Y \quad (8\text{-}481)$$

– Otherwise, if cIdx is equal to 1, the following applies:

$$qP = Qp'_{Cb} \quad (8\text{-}482)$$

– Otherwise (cIdx is equal to 2), the following applies:

$$qP = Qp'_{Cr} \quad (8\text{-}483)$$

The variables bdShift, rectNorm and bdOffset are derived as follows:

$$bdShift = bitDepth + (((Log2(nTbW) + Log2(nTbH)) \& 1) * 8 + (Log2(nTbW) + Log2(nTbH)) / 2) - 5 + dep\_quant\_enabled\_flag \quad (8\text{-}484)$$

$$rectNorm = ((Log2(nTbW) + Log2(nTbH)) \& 1) == 1 \ ? \ 181 : 1 \quad (8\text{-}485)$$

$$bdOffset = (1 << bdShift) >> 1 \quad (8\text{-}486)$$

The list levelScale[ ] is specified as levelScale[k] = { 40, 45, 51, 57, 64, 72 } with k = 0..5.

For the derivation of the scaled transform coefficients d[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1, the following applies:

– The intermediate scaling factor m[x][y] is set equal to 16.

– The scaling factor ls[x][y] is derived as follows:

  – If dep_quant_enabled_flag is equal to 1, the following applies:

$$ls[x][y] = (m[x][y] * levelScale[(qP + 1) \% 6]) << ((qP + 1) / 6) \quad (8\text{-}487)$$

  – Otherwise (dep_quant_enabled_flag is equal to 0), the following applies:

$$ls[x][y] = (m[x][y] * levelScale[qP \% 6]) << (qP / 6) \quad (8\text{-}488)$$

– The value dnc[x][y] is derived as follows:

$$dnc[x][y] = (TransCoeffLevel[xTbY][yTbY][cIdx][x][y] * ls[x][y] * rectNorm + bdOffset) >> bdShift \quad (8\text{-}489)$$

– The scaled transform coefficient d[x][y] is derived as follows:

$$d[x][y] = Clip3(CoeffMin, CoeffMax, dnc[x][y]) \quad (8\text{-}490)$$

8.4.4 Transformation process for scaled transform coefficients

8.4.4.1 General

Inputs to this process are:

– a luma location (xTbY, yTbY) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the current picture,

– a variable nTbW specifying the width of the current transform block,

– a variable nTbH specifying the height of the current transform block,

– a variable cIdx specifying the colour component of the current block,

– an (nTbW)x(nTbH) array d[x][y] of scaled transform coefficients with x = 0..nTbW − 1, y = 0..nTbH − 1.

Output of this process is the (nTbW)x(nTbH) array r[x][y] of residual samples with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable trTypeHor specifying the horizontal transform kernel and the variable trTypeVer specifying the vertical transform kernel are derived in Table 8-11 depending on mts_idx[xTbY][yTbY][cIdx].

The variables nonZeroW and nonZeroH are derived as follows:

$$nonZeroW = Min(nTbW, 32) \quad (8\text{-}491)$$

$$nonZeroH = Min(nTbH, 32) \quad (8\text{-}492)$$

The (nTbW)x(nTbH) array r of residual samples is derived as follows:

1. Each (vertical) column of scaled transform coefficients d[x][y] with x = 0..nonZeroW − 1, y = 0..nonZeroH − 1 is transformed to e[x][y] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.4.4.2 for each column x = 0..nonZeroW − 1 with the height of the transform block nTbH, the non-zero height of the scaled transform coefficients nonZeroH, the list d[x][y] with y = 0..nonZeroH − 1 and the transform type variable trType set equal to trTypeVer as inputs, and the output is the list e[x][y] with y = 0..nTbH − 1.
2. The intermediate sample values g[x][y] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 are derived as follows:

$$g[x][y] = Clip3(CoeffMin, CoeffMax, (e[x][y] + 64) >> 7) \quad (8\text{-}493)$$

3. Each (horizontal) row of the resulting array g[x][y] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 is transformed to r[x][y] with x = 0..nTbW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.4.4.2 for each row y = 0..nTbH − 1 with the width of the transform block nTbW, the non-zero width of the resulting array g[x][y] nonZeroW, the list g[x][y] with x = 0..nonZeroW − 1 and the transform type variable trType set equal to trTypeHor as inputs, and the output is the list r[x][y] with x = 0..nTbW − 1.

Table 8-11 – Specification of trTypeHor and trTypeVer depending on mts_idx[x][y][cIdx]

mts_idx[xTbY][yTbY][cIdx]	trTypeHor	trTypeVer
−1	0	0
0	1	1
1	2	1
2	1	2
3	2	2

8.4.4.2 Transformation process

Inputs to this process are:

– a variable nTbS specifying the horizontal sample size of transformed samples,

– a variable nonZeroS specifying the horizontal sample size of non-zero scaled transform coefficients,

– a list of scaled transform coefficients x[j] with j = 0..nonZeroS − 1,

– a transform kernel type variable trType.

Output of this process is the list of transformed samples y[i] with i = 0..nTbS − 1.

The transformation matrix derivation process as specified in clause 8.4.4.3 in invoked with the transform size nTbS and the transform kernel Type trType as inputs, and the transformation maxtrix transMatrix as output.

Depending on the value of trType, the following applies:, the list of transformed samples y[i] with i = 0..nTbS − 1 is derived as follows:

– If trType is equal to 0, the following transform matrix multiplication applies:

$$y[i]= \sum_{j=0}^{nonZeroS-1} transMatrix[i][j * 2^{6-Log2(nTbS)}] * x[j] \text{ with } i = 0..nTbS - 1 \quad (8\text{-}494)$$

– Otherwise (trType is equal to 1 or trType is equal to 2), the following transform matrix multiplication applies:

$$y[i]= \sum_{j=0}^{nonZeroS-1} transMatrix[i][j] * x[j] \text{ with } i = 0..nTbS - 1 \quad (8\text{-}495)$$

8.4.4.3 Transformation matrix derivation process

Inputs to this process are:

– a variable nTbS specifying the horizontal sample size of scaled transform coefficients,

– the transformation kernel type trType.

Output of this process is the transformation matrix transMatrix.

The transformation matrix transMatrix is derived based on trType and nTbs as follows:

– If trType is equal to 0, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..63 (8-496)

transMatrixCol0to15 = (8-497)

```
{
{  64   64   64   64   64   64   64   64   64   64   64   64   64   64   64   64 }
{  91   90   90   90   88   87   86   84   83   81   79   77   73   71   69   65 }
{  90   90   88   85   82   78   73   67   61   54   46   38   31   22   13    4 }
{  90   88   84   79   71   62   52   41   28   15    2  -11  -24  -37  -48  -59 }
{  90   87   80   70   57   43   25    9   -9  -25  -43  -57  -70  -80  -87  -90 }
{  90   84   73   59   41   20   -2  -24  -44  -62  -77  -86  -90  -90  -83  -71 }
{  90   82   67   46   22   -4  -31  -54  -73  -85  -90  -88  -78  -61  -38  -13 }
{  90   79   59   33    2  -28  -56  -77  -88  -90  -81  -62  -37   -7   24   52 }
{  89   75   50   18  -18  -50  -75  -89  -89  -75  -50  -18   18   50   75   89 }
{  88   71   41    2  -37  -69  -87  -90  -73  -44   -7   33   65   86   90   77 }
{  88   67   31  -13  -54  -82  -90  -78  -46   -4   38   73   90   85   61   22 }
{  87   62   20  -28  -69  -90  -84  -56  -11   37   73   90   81   48    2  -44 }
{  87   57    9  -43  -80  -90  -70  -25   25   70   90   80   43   -9  -57  -87 }
{  86   52   -2  -56  -87  -84  -48    7   59   88   83   44  -11  -62  -90  -81 }
{  85   46  -13  -67  -90  -73  -22   38   82   88   54   -4  -61  -90  -78  -31 }
{  84   41  -24  -77  -90  -56    7   65   91   69   11  -52  -88  -79  -28   37 }
{  83   36  -36  -83  -83  -36   36   83   83   36  -36  -83  -83  -36   36   83 }
{  83   28  -44  -88  -73  -11   59   91   62   -7  -71  -90  -48   24   81   84 }
{  82   22  -54  -90  -61   13   78   85   31  -46  -90  -67    4   73   88   38 }
{  81   15  -62  -90  -44   37   88   69   -7  -77  -84  -24   56   91   52  -28 }
{  80    9  -70  -87  -25   57   90   43  -43  -90  -57   25   87   70   -9  -80 }
{  79    2  -77  -81   -7   73   83   11  -71  -84  -15   69   86   20  -65  -87 }
{  78   -4  -82  -73   13   85   67  -22  -88  -61   31   90   54  -38  -90  -46 }
{  77  -11  -86  -62   33   90   44  -52  -90  -24   69   83    2  -81  -71   20 }
{  75  -18  -89  -50   50   89   18  -75  -75   18   89   50  -50  -89  -18   75 }
{  73  -24  -90  -37   65   81  -11  -88  -48   56   86    2  -84  -59   44   90 }
{  73  -31  -90  -22   78   67  -38  -90  -13   82   61  -46  -88   -4   85   54 }
{  71  -37  -90   -7   86   48  -62  -79   24   91   20  -81  -59   52   84  -11 }
{  70  -43  -87    9   90   25  -80  -57   57   80  -25  -90   -9   87   43  -70 }
{  69  -48  -83   24   90    2  -90  -28   81   52  -65  -71   44   84  -20  -90 }
{  67  -54  -78   38   85  -22  -90    4   90   13  -88  -31   82   46  -73  -61 }
{  65  -59  -71   52   77  -44  -81   37   84  -28  -87   20   90  -11  -90    2 }
{  64  -64  -64   64   64  -64  -64   64   64  -64  -64   64   64  -64  -64   64 }
{  62  -69  -56   73   48  -79  -41   83   33  -86  -24   88   15  -90   -7   91 }
{  61  -73  -46   82   31  -88  -13   90   -4  -90   22   85  -38  -78   54   67 }
{  59  -77  -37   87   11  -91   15   86  -41  -73   62   56  -79  -33   88    7 }
{  57  -80  -25   90   -9  -87   43   70  -70  -43   87    9  -90   25   80  -57 }
{  56  -83  -15   90  -28  -77   65   44  -87   -2   88  -41  -69   73   33  -90 }
{  54  -85   -4   88  -46  -61   82   13  -90   38   67  -78  -22   90  -31  -73 }
{  52  -87    7   83  -62  -41   90  -20  -77   71   28  -91   33   69  -79  -15 }
{  50  -89   18   75  -75  -18   89  -50  -50   89  -18  -75   75   18  -89   50 }
{  48  -90   28   65  -84    7   79  -73  -15   87  -59  -37   91  -41  -56   88 }
{  46  -90   38   54  -90   31   61  -88   22   67  -85   13   73  -82    4   78 }
{  44  -91   48   41  -90   52   37  -90   56   33  -90   59   28  -88   62   24 }
{  43  -90   57   25  -87   70    9  -80   80   -9  -70   87  -25  -57   90  -43 }
{  41  -90   65   11  -79   83  -20  -59   90  -48  -33   87  -71   -2   73  -86 }
{  38  -88   73   -4  -67   90  -46  -31   85  -78   13   61  -90   54   22  -82 }
{  37  -86   79  -20  -52   90  -69    2   65  -90   56   15  -77   87  -41  -33 }
{  36  -83   83  -36  -36   83  -83   36   36  -83   83  -36  -36   83  -83   36 }
{  33  -81   87  -48  -15   71  -90   62   -2  -59   90  -73   20   44  -86   83 }
{  31  -78   90  -61    4   54  -88   82  -38  -22   73  -90   67  -13  -46   85 }
{  28  -73   91  -71   24   33  -77   90  -69   20   37  -79   90  -65   15   41 }
{  25  -70   90  -80   43    9  -57   87  -87   57   -9  -43   80  -90   70  -25 }
{  24  -65   88  -86   59  -15  -33   71  -90   83  -52    7   41  -77   91  -79 }
{  22  -61   85  -90   73  -38   -4   46  -78   90  -82   54  -13  -31   67  -88 }
{  20  -56   81  -91   83  -59   24   15  -52   79  -90   84  -62   28   11  -48 }
{  18  -50   75  -89   89  -75   50  -18  -18   50  -75   89  -89   75  -50   18 }
{  15  -44   69  -84   91  -86   71  -48   20   11  -41   65  -83   90  -87   73 }
{  13  -38   61  -78   88  -90   85  -73   54  -31    4   22  -46   67  -82   90 }
{  11  -33   52  -69   81  -88   91  -87   79  -65   48  -28    7   15  -37   56 }
{   9  -25   43  -57   70  -80   87  -90   90  -87   80  -70   57  -43   25   -9 }
{   7  -20   33  -44   56  -65   73  -81   86  -90   91  -90   87  -83   77  -69 }
{   4  -13   22  -31   38  -46   54  -61   67  -73   78  -82   85  -88   90  -90 }
{   2   -7   11  -15   20  -24   28  -33   37  -41   44  -48   52  -56   59  -62 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..63 (8-498)

transMatrixCol16to31 = (8-499)

```
{
{  64   64   64   64   64   64   64   64   64   64   64   64   64   64   64   64 }
{  62   59   56   52   48   44   41   37   33   28   24   20   15   11    7    2 }
{  -4  -13  -22  -31  -38  -46  -54  -61  -67  -73  -78  -82  -85  -88  -90  -90 }
{ -69  -77  -83  -87  -90  -91  -90  -86  -81  -73  -65  -56  -44  -33  -20   -7 }
{ -90  -87  -80  -70  -57  -43  -25   -9    9   25   43   57   70   80   87   90 }
{ -56  -37  -15    7   28   48   65   79   87   91   88   81   69   52   33   11 }
{  13   38   61   78   88   90   85   73   54   31    4  -22  -46  -67  -82  -90 }
{  73   87   90   83   65   41   11  -20  -48  -71  -86  -91  -84  -69  -44  -15 }
{  89   75   50   18  -18  -50  -75  -89  -89  -75  -50  -18   18   50   75   89 }
{  48   11  -28  -62  -84  -90  -79  -52  -15   24   59   83   91   81   56   20 }
{ -22  -61  -85  -90  -73  -38    4   46   78   90   82   54   13  -31  -67  -88 }
{ -79  -91  -77  -41    7   52   83   90   71   33  -15  -59  -86  -88  -65  -24 }
{ -87  -57   -9   43   80   90   70   25  -25  -70  -90  -80  -43    9   57   87 }
{ -41   15   65   90   79   37  -20  -69  -90  -77  -33   24   71   91   73   28 }
{  31   78   90   61    4  -54  -88  -82  -38   22   73   90   67   13  -46  -85 }
{  83   86   44  -20  -73  -90  -59    2   62   90   71   15  -48  -87  -81  -33 }
{  83   36  -36  -83  -83  -36   36   83   83   36  -36  -83  -83  -36   36   83 }
{  33  -41  -87  -77  -15   56   90   65   -2  -69  -90  -52   20   79   86   37 }
{ -38  -88  -73   -4   67   90   46  -31  -85  -78  -13   61   90   54  -22  -82 }
{ -86  -73   -2   71   87   33  -48  -90  -59   20   83   79   11  -65  -90  -41 }
{ -80   -9   70   87   25  -57  -90  -43   43   90   57  -25  -87  -70    9   80 }
{ -24   62   88   28  -59  -90  -33   56   90   37  -52  -90  -41   48   91   44 }
{  46   90   38  -54  -90  -31   61   88   22  -67  -85  -13   73   82    4  -78 }
{  88   56  -41  -91  -37   59   87   15  -73  -79    7   84   65  -28  -90  -48 }
{  75  -18  -89  -50   50   89   18  -75  -75   18   89   50  -50  -89  -18   75 }
{  15  -79  -69   33   91   28  -71  -77   20   90   41  -62  -83    7   87   52 }
{ -54  -85    4   88   46  -61  -82   13   90   38  -67  -78   22   90   31  -73 }
{ -90  -33   73   69  -41  -88   -2   87   44  -65  -77   28   90   15  -83  -56 }
{ -70   43   87   -9  -90  -25   80   57  -57  -80   25   90    9  -87  -43   70 }
{  -7   88   33  -79  -56   62   73  -41  -86   15   91   11  -87  -37   77   59 }
{  61   73  -46  -82   31   88  -13  -90   -4   90   22  -85  -38   78   54  -67 }
{  91    7  -90  -15   88   24  -86  -33   83   41  -79  -48   73   56  -69  -62 }
{  64  -64  -64   64   64  -64  -64   64   64  -64  -64   64   64  -64  -64   64 }
{  -2  -90   11   90  -20  -87   28   84  -37  -81   44   77  -52  -71   59   65 }
{ -67  -54   78   38  -85  -22   90    4  -90   13   88  -31  -82   46   73  -61 }
{ -90   20   84  -44  -71   65   52  -81  -28   90    2  -90   24   83  -48  -69 }
{ -57   80   25  -90    9   87  -43  -70   70   43  -87   -9   90  -25  -80   57 }
{  11   84  -52  -59   81   20  -91   24   79  -62  -48   86    7  -90   37   71 }
{  73   31  -90   22   78  -67  -38   90  -13  -82   61   46  -88    4   85  -54 }
{  90  -44  -59   84    2  -86   56   48  -88   11   81  -65  -37   90  -24  -73 }
{  50  -89   18   75  -75  -18   89  -50  -50   89  -18  -75   75   18  -89   50 }
{ -20  -71   81    2  -83   69   24  -90   52   44  -90   33   62  -86   11   77 }
{ -78   -4   82  -73  -13   85  -67  -22   88  -61  -31   90  -54  -38   90  -46 }
{ -87   65   20  -86   69   15  -84   71   11  -83   73    7  -81   77    2  -79 }
{ -43   90  -57  -25   87  -70   -9   80  -80    9   70  -87   25   57  -90   43 }
{  28   52  -91   56   24  -84   77   -7  -69   88  -37  -44   90  -62  -15   81 }
{  82  -22  -54   90  -61  -13   78  -85   31   46  -90   67    4  -73   88  -38 }
{  84  -81   24   48  -90   71   -7  -62   91  -59  -11   73  -88   44   28  -83 }
{  36  -83   83  -36  -36   83  -83   36   36  -83   83  -36  -36   83  -83   36 }
{ -37  -28   79  -88   52   11  -69   91  -65    7   56  -90   77  -24  -41   84 }
{ -85   46   13  -67   90  -73   22   38  -82   88  -54   -4   61  -90   78  -31 }
{ -81   90  -62   11   44  -83   88  -59    7   48  -84   87  -56    2   52  -86 }
{ -25   70  -90   80  -43   -9   57  -87   87  -57    9   43  -80   90  -70   25 }
{  44    2  -48   81  -90   73  -37  -11   56  -84   90  -69   28   20  -62   87 }
{  88  -67   31   13  -54   82  -90   78  -46    4   38  -73   90  -85   61  -22 }
{  77  -90   86  -65   33    7  -44   73  -90   87  -69   37    2  -41   71  -88 }
{  18  -50   75  -89   89  -75   50  -18  -18   50  -75   89  -89   75  -50   18 }
{ -52   24    7  -37   62  -81   90  -88   77  -56   28    2  -33   59  -79   90 }
{ -90   82  -67   46  -22   -4   31  -54   73  -85   90  -88   78  -61   38  -13 }
{ -71   83  -90   90  -86   77  -62   44  -24    2   20  -41   59  -73   84  -90 }
{  -9   25  -43   57  -70   80  -87   90  -90   87  -80   70  -57   43  -25    9 }
{  59  -48   37  -24   11    2  -15   28  -41   52  -62   71  -79   84  -88   90 }
{  90  -90   88  -85   82  -78   73  -67   61  -54   46  -38   31  -22   13   -4 }
{  65  -69   71  -73   77  -79   81  -83   84  -86   87  -88   90  -90   90  -91 }
},
```

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol16to31[47 −m][n] (8-500)
with m =32..47, n = 0..63

transMatrix[m][n] = (n & 1 ? −1 : 1) * transMatrixCol0to15[63 −m][n] (8-501)
with m =48..63, n = 0..63

– Otherwise, if trType is equal to 1 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (8-502)

```
{
{  29   55   74   84 }
{  74   74    0  -74 }
{  84  -29  -74   55 }
{  55  -84   74  -29 }
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 8, the following applies:

transMatrix[m][n] = (8-503)

```
{
{  17   32   46   60   71   78   85   86 }
{  46   78   86   71   32  -17  -60  -85 }
{  71   85   32  -46  -86  -60   17   78 }
{  85   46  -60  -78   17   86   32  -71 }
{  86  -17  -85   32   78  -46  -71   60 }
{  78  -71  -17   85  -60  -32   86  -46 }
{  60  -86   71  -17  -46   85  -78   32 }
{  32  -60   78  -86   85  -71   46  -17 }
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-504)

```
{
{   9   17   25   33   41   49   56   62   66   72   77   81   83   87   89   90 }
{  25   49   66   81   89   89   81   66   49   25    0  -25  -49  -66  -81  -89 }
{  41   72   89   83   62   25  -17  -56  -81  -90  -77  -49   -9   33   66   87 }
{  56   87   81   41  -17  -66  -90  -72  -25   33   77   89   62    9  -49  -83 }
{  66   89   49  -25  -81  -81  -25   49   89   66    0  -66  -89  -49   25   81 }
{  77   77    0  -77  -77    0   77   77    0  -77  -77    0   77   77    0  -77 }
{  83   56  -49  -87   -9   81   62  -41  -89  -17   77   66  -33  -90  -25   72 }
{  89   25  -81  -49   66   66  -49  -81   25   89    0  -89  -25   81   49  -66 }
{  90   -9  -89   17   87  -25  -83   33   81  -41  -77   49   72  -56  -66   62 }
{  87  -41  -66   72   33  -89    9   83  -49  -62   77   25  -90   17   81  -56 }
{  81  -66  -25   89  -49  -49   89  -25  -66   81    0  -81   66   25  -89   49 }
{  72  -83   25   56  -90   49   33  -87   66    9  -77   81  -17  -62   89  -41 }
{  62  -90   66   -9  -56   89  -72   17   49  -87   77  -25  -41   83  -81   33 }
{  49  -81   89  -66   25   25  -66   89  -81   49    0  -49   81  -89   66  -25 }
{  33  -62   81  -90   83  -66   41   -9  -25   56  -77   89  -87   72  -49   17 }
{  17  -33   49  -62   72  -81   87  -90   89  -83   77  -66   56  -41   25   -9 }
},
```

– Otherwise, if trType is equal to 1 and nTbs is equal to 32, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..31 (8-505)

transMatrixCol0to15 = (8-506)

```
{
{   4    9   13   17   21   26   30   34   38   42   45   50   53   56   60   63 }
{  13   26   38   50   60   68   77   82   86   89   90   88   85   80   74   66 }
{  21   42   60   74   84   89   89   84   74   60   42   21    0  -21  -42  -60 }
{  30   56   77   88   89   80   63   38    9  -21  -50  -72  -85  -90  -84  -68 }
{  38   68   86   88   74   45    9  -30  -63  -84  -90  -78  -53  -17   21   56 }
{  45   78   90   77   42   -4  -50  -80  -90  -74  -38    9   53   82   89   72 }
{  53   85   85   53    0  -53  -85  -85  -53    0   53   85   85   53    0  -53 }
{  60   89   74   21  -42  -84  -84  -42   21   74   89   60    0  -60  -89  -74 }
{  66   90   56  -13  -74  -88  -45   26   80   84   34  -38  -85  -78  -21   50 }
{  72   86   34  -45  -89  -63   13   78   82   21  -56  -90  -53   26   84   77 }
{  77   80    9  -72  -84  -17   66   86   26  -60  -88  -34   53   90   42  -45 }
{  80   72  -17  -86  -60   34   90   45  -50  -89  -30   63   85   13  -74  -78 }
{  84   60  -42  -89  -21   74   74  -21  -89  -42   60   84    0  -84  -60   42 }
{  86   45  -63  -78   21   90   26  -77  -66   42   88    4  -85  -50   60   80 }
{  88   30  -78  -56   60   77  -34  -88    4   89   26  -80  -53   63   74  -38 }
{  90   13  -88  -26   84   38  -78  -50   72   60  -63  -68   53   77  -42  -82 }
{  90   -4  -90    9   89  -13  -88   17   88  -21  -86   26   85  -30  -84   34 }
{  89  -21  -84   42   74  -60  -60   74   42  -84  -21   89    0  -89   21   84 }
{  88  -38  -72   68   42  -86   -4   88  -34  -74   66   45  -85   -9   89  -30 }
{  85  -53  -53   85    0  -85   53   53  -85    0   85  -53  -53   85    0  -85 }
{  82  -66  -30   90  -42  -56   86  -13  -77   74   17  -88   53   45  -89   26 }
{  78  -77   -4   80  -74   -9   82  -72  -13   84  -68  -17   85  -66  -21   86 }
{  74  -84   21   60  -89   42   42  -89   60   21  -84   74    0  -74   84  -21 }
{  68  -88   45   30  -84   78  -17  -56   90  -60  -13   77  -85   34   42  -88 }
{  63  -90   66   -4  -60   90  -68    9   56  -89   72  -13  -53   88  -74   17 }
{  56  -88   80  -38  -21   72  -90   68  -17  -42   82  -86   53    4  -60   88 }
{  50  -82   88  -66   21   30  -72   90  -78   42    9  -56   85  -86   60  -13 }
{  42  -74   89  -84   60  -21  -21   60  -84   89  -74   42    0  -42   74  -89 }
{  34  -63   82  -90   84  -66   38   -4  -30   60  -80   90  -85   68  -42    9 }
{  26  -50   68  -82   89  -88   80  -66   45  -21   -4   30  -53   72  -84   90 }
{  17  -34   50  -63   74  -82   88  -90   88  -84   77  -66   53  -38   21   -4 }
{   9  -17   26  -34   42  -50   56  -63   68  -74   78  -82   85  -88   89  -90 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-507)

transMatrixCol16to31 = (8-508)

```
{
{  66   68   72   74   77   78   80   82   84   85   86   88   88   89   90   90 }
{  56   45   34   21    9   -4  -17  -30  -42  -53  -63  -72  -78  -84  -88  -90 }
{ -74  -84  -89  -89  -84  -74  -60  -42  -21    0   21   42   60   74   84   89 }
{ -45  -17   13   42   66   82   90   86   74   53   26   -4  -34  -60  -78  -88 }
{  80   90   82   60   26  -13  -50  -77  -89  -85  -66  -34    4   42   72   88 }
{  34  -13  -56  -84  -88  -68  -30   17   60   85   88   66   26  -21  -63  -86 }
{ -85  -85  -53    0   53   85   85   53    0  -53  -85  -85  -53    0   53   85 }
{ -21   42   84   84   42  -21  -74  -89  -60    0   60   89   74   21  -42  -84 }
{  88   72    9  -60  -90  -63    4   68   89   53  -17  -77  -86  -42   30   82 }
{   9  -66  -88  -42   38   88   68   -4  -74  -85  -30   50   90   60  -17  -80 }
{ -90  -50   38   89   56  -30  -88  -63   21   85   68  -13  -82  -74    4   78 }
{   4   82   68  -21  -88  -56   38   90   42  -53  -88  -26   66   84    9  -77 }
{  89   21  -74  -74   21   89   42  -60  -84    0   84   60  -42  -89  -21   74 }
{ -17  -90  -30   74   68  -38  -88   -9   84   53  -56  -82   13   89   34  -72 }
{ -86    9   90   21  -82  -50   66   72  -42  -85   13   90   17  -84  -45   68 }
{  30   86  -17  -89    4   90    9  -88  -21   85   34  -80  -45   74   56  -66 }
{  82  -38  -80   42   78  -45  -77   50   74  -53  -72   56   68  -60  -66   63 }
{ -42  -74   60   60  -74  -42   84   21  -89    0   89  -21  -84   42   74  -60 }
{ -77   63   50  -84  -13   90  -26  -78   60   53  -82  -17   90  -21  -80   56 }
{  53   53  -85    0   85  -53  -53   85    0  -85   53   53  -85    0   85  -53 }
{  68  -80   -4   84  -63  -34   90  -38  -60   85   -9  -78   72   21  -88   50 }
{ -63  -26   88  -60  -30   88  -56  -34   89  -53  -38   90  -50  -42   90  -45 }
{ -60   89  -42  -42   89  -60  -21   84  -74    0   74  -84   21   60  -89   42 }
{  72   -4  -66   89  -50  -26   82  -80   21   53  -90   63    9  -74   86  -38 }
{  50  -88   77  -21  -45   86  -78   26   42  -85   80  -30  -38   84  -82   34 }
{ -78   34   26  -74   90  -66   13   45  -84   85  -50   -9   63  -89   77  -30 }
{ -38   77  -90   74  -34  -17   63  -88   84  -53    4   45  -80   89  -68   26 }
{  84  -60   21   21  -60   84  -89   74  -42    0   42  -74   89  -84   60  -21 }
{  26  -56   78  -89   86  -72   45  -13  -21   53  -77   88  -88   74  -50   17 }
{ -88   78  -63   42  -17   -9   34  -56   74  -85   90  -86   77  -60   38  -13 }
{ -13   30  -45   60  -72   80  -86   90  -89   85  -78   68  -56   42  -26    9 }
{  90  -88   86  -84   80  -77   72  -66   60  -53   45  -38   30  -21   13   -4 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 4, the following applies:

transMatrix[m][n] = (8-509)

```
{
{  84   74   55   29 }
{  74    0  -74  -74 }
{  55  -74  -29   84 }
{  29  -74   84  -55 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 8, the following applies:

transMatrix[m][n] = (8-510)

```
{
{  86   85   78   71   60   46   32   17 }
{  85   60   17  -32  -71  -86  -78  -46 }
{  78   17  -60  -86  -46   32   85   71 }
{  71  -32  -86  -17   78   60  -46  -85 }
{  60  -71  -46   78   32  -85  -17   86 }
{  46  -86   32   60  -85   17   71  -78 }
{  32  -78   85  -46  -17   71  -86   60 }
{  17  -46   71  -85   86  -78   60  -32 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 16, the following applies:

transMatrix[m][n] = (8-511)

```
{
{  90   89   87   83   81   77   72   66   62   56   49   41   33   25   17    9 }
{  89   81   66   49   25    0  -25  -49  -66  -81  -89  -89  -81  -66  -49  -25 }
{  87   66   33   -9  -49  -77  -90  -81  -56  -17   25   62   83   89   72   41 }
{  83   49   -9  -62  -89  -77  -33   25   72   90   66   17  -41  -81  -87  -56 }
{  81   25  -49  -89  -66    0   66   89   49  -25  -81  -81  -25   49   89   66 }
{  77    0  -77  -77    0   77   77    0  -77  -77    0   77   77    0  -77  -77 }
{  72  -25  -90  -33   66   77  -17  -89  -41   62   81   -9  -87  -49   56   83 }
{  66  -49  -81   25   89    0  -89  -25   81   49  -66  -66   49   81  -25  -89 }
{  62  -66  -56   72   49  -77  -41   81   33  -83  -25   87   17  -89   -9   90 }
{  56  -81  -17   90  -25  -77   62   49  -83   -9   89  -33  -72   66   41  -87 }
{  49  -89   25   66  -81    0   81  -66  -25   89  -49  -49   89  -25  -66   81 }
{  41  -89   62   17  -81   77   -9  -66   87  -33  -49   90  -56  -25   83  -72 }
{  33  -81   83  -41  -25   77  -87   49   17  -72   89  -56   -9   66  -90   62 }
{  25  -66   89  -81   49    0  -49   81  -89   66  -25  -25   66  -89   81  -49 }
{  17  -49   72  -87   89  -77   56  -25   -9   41  -66   83  -90   81  -62   33 }
{   9  -25   41  -56   66  -77   83  -89   90  -87   81  -72   62  -49   33  -17 }
},
```

– Otherwise, if trType is equal to 2 and nTbs is equal to 32, the following applies:

transMatrix[m][n] = transMatrixCol0to15[m][n] with m = 0..15, n = 0..31 (8-512)

transMatrixCol0to15 = (8-513)

```
{
{  90   90   89   88   88   86   85   84   82   80   78   77   74   72   68   66 }
{  90   88   84   78   72   63   53   42   30   17    4   -9  -21  -34  -45  -56 }
{  89   84   74   60   42   21    0  -21  -42  -60  -74  -84  -89  -89  -84  -74 }
{  88   78   60   34    4  -26  -53  -74  -86  -90  -82  -66  -42  -13   17   45 }
{  88   72   42    4  -34  -66  -85  -89  -77  -50  -13   26   60   82   90   80 }
{  86   63   21  -26  -66  -88  -85  -60  -17   30   68   88   84   56   13  -34 }
{  85   53    0  -53  -85  -85  -53    0   53   85   85   53    0  -53  -85  -85 }
{  84   42  -21  -74  -89  -60    0   60   89   74   21  -42  -84  -84  -42   21 }
{  82   30  -42  -86  -77  -17   53   89   68    4  -63  -90  -60    9   72   88 }
{  80   17  -60  -90  -50   30   85   74    4  -68  -88  -38   42   88   66   -9 }
{  78    4  -74  -82  -13   68   85   21  -63  -88  -30   56   89   38  -50  -90 }
{  77   -9  -84  -66   26   88   53  -42  -90  -38   56   88   21  -68  -82   -4 }
{  74  -21  -89  -42   60   84    0  -84  -60   42   89   21  -74  -74   21   89 }
{  72  -34  -89  -13   82   56  -53  -84    9   88   38  -68  -74   30   90   17 }
{  68  -45  -84   17   90   13  -85  -42   72   66  -50  -82   21   90    9  -86 }
{  66  -56  -74   45   80  -34  -85   21   88   -9  -90   -4   89   17  -86  -30 }
{  63  -66  -60   68   56  -72  -53   74   50  -77  -45   78   42  -80  -38   82 }
{  60  -74  -42   84   21  -89    0   89  -21  -84   42   74  -60  -60   74   42 }
{  56  -80  -21   90  -17  -82   53   60  -78  -26   90  -13  -84   50   63  -77 }
{  53  -85    0   85  -53  -53   85    0  -85   53   53  -85    0   85  -53  -53 }
{  50  -88   21   72  -78   -9   85  -60  -38   90  -34  -63   84   -4  -80   68 }
{  45  -90   42   50  -90   38   53  -89   34   56  -88   30   60  -88   26   63 }
{  42  -89   60   21  -84   74    0  -74   84  -21  -60   89  -42  -42   89  -60 }
{  38  -86   74   -9  -63   90  -53  -21   80  -82   26   50  -89   66    4  -72 }
{  34  -82   84  -38  -30   80  -85   42   26  -78   86  -45  -21   77  -88   50 }
{  30  -77   89  -63    9   50  -85   84  -45  -13   66  -90   74  -26  -34   78 }
{  26  -68   89  -80   45    4  -53   84  -88   63  -17  -34   74  -90   77  -38 }
{  21  -60   84  -89   74  -42    0   42  -74   89  -84   60  -21  -21   60  -84 }
{  17  -50   74  -88   88  -77   53  -21  -13   45  -72   86  -89   78  -56   26 }
{  13  -38   60  -77   86  -90   85  -74   56  -34    9   17  -42   63  -78   88 }
{   9  -26   42  -56   68  -78   85  -89   90  -86   80  -72   60  -45   30  -13 }
{   4  -13   21  -30   38  -45   53  -60   66  -72   77  -80   84  -86   88  -90 }
},
```

transMatrix[m][n] = transMatrixCol16to31[m − 16][n] with m = 16..31, n = 0..31 (8-514)

transMatrixCol16to31 = (8-515)

```
{
{  63   60   56   53   50   45   42   38   34   30   26   21   17   13    9    4 }
{ -66  -74  -80  -85  -88  -90  -89  -86  -82  -77  -68  -60  -50  -38  -26  -13 }
{ -60  -42  -21    0   21   42   60   74   84   89   89   84   74   60   42   21 }
{  68   84   90   85   72   50   21   -9  -38  -63  -80  -89  -88  -77  -56  -30 }
{  56   21  -17  -53  -78  -90  -84  -63  -30    9   45   74   88   86   68   38 }
{ -72  -89  -82  -53   -9   38   74   90   80   50    4  -42  -77  -90  -78  -45 }
{ -53    0   53   85   85   53    0  -53  -85  -85  -53    0   53   85   85   53 }
{  74   89   60    0  -60  -89  -74  -21   42   84   84   42  -21  -74  -89  -60 }
{  50  -21  -78  -85  -38   34   84   80   26  -45  -88  -74  -13   56   90   66 }
{ -77  -84  -26   53   90   56  -21  -82  -78  -13   63   89   45  -34  -86  -72 }
{ -45   42   90   53  -34  -88  -60   26   86   66  -17  -84  -72    9   80   77 }
{  78   74  -13  -85  -63   30   89   50  -45  -90  -34   60   86   17  -72  -80 }
{  42  -60  -84    0   84   60  -42  -89  -21   74   74  -21  -89  -42   60   84 }
{ -80  -60   50   85   -4  -88  -42   66   77  -26  -90  -21   78   63  -45  -86 }
{ -38   74   63  -53  -80   26   89    4  -88  -34   77   60  -56  -78   30   88 }
{  82   42  -77  -53   68   63  -60  -72   50   78  -38  -84   26   88  -13  -90 }
{  34  -84  -30   85   26  -86  -21   88   17  -88  -13   89    9  -90   -4   90 }
{ -84  -21   89    0  -89   21   84  -42  -74   60   60  -74  -42   84   21  -89 }
{ -30   89   -9  -85   45   66  -74  -34   88   -4  -86   42   68  -72  -38   88 }
{  85    0  -85   53   53  -85    0   85  -53  -53   85    0  -85   53   53  -85 }
{  26  -89   45   53  -88   17   74  -77  -13   86  -56  -42   90  -30  -66   82 }
{ -86   21   66  -85   17   68  -84   13   72  -82    9   74  -80    4   77  -78 }
{ -21   84  -74    0   74  -84   21   60  -89   42   42  -89   60   21  -84   74 }
{  88  -42  -34   85  -77   13   60  -90   56   17  -78   84  -30  -45   88  -68 }
{  17  -74   88  -53  -13   72  -89   56    9  -68   90  -60   -4   66  -90   63 }
{ -88   60   -4  -53   86  -82   42   17  -68   90  -72   21   38  -80   88  -56 }
{ -13   60  -86   85  -56    9   42  -78   90  -72   30   21  -66   88  -82   50 }
{  89  -74   42    0  -42   74  -89   84  -60   21   21  -60   84  -89   74  -42 }
{   9  -42   68  -85   90  -80   60  -30   -4   38  -66   84  -90   82  -63   34 }
{ -90   84  -72   53  -30    4   21  -45   66  -80   88  -89   82  -68   50  -26 }
{  -4   21  -38   53  -66   77  -84   88  -90   88  -82   74  -63   50  -34   17 }
{  90  -89   88  -85   82  -78   74  -68   63  -56   50  -42   34  -26   17   -9 }
},
```

8.4.5 Picture reconstruction process

[Ed. (BB): tbd]

8.5 In-loop Filter Process

8.5.1 General

The three in-loop filters, namely deblocking filter, sample adaptive offset and adaptive loop filter, are applied as specified by the following ordered steps:

1. For the deblocking filter, the following applies:
   - The deblocking filter process as specified in clause 8.5.2.1 is invoked with the reconstructed picture sample array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} after deblocking as outputs.
   - The array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} are assigned to the array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.
2. When sps_sao_enabled_flag is equal to 1, the following applies:
   - The sample adaptive offset process as specified in clause 8.5.3.1 is invoked with the reconstructed picture sample array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} after sample adaptive offset as outputs.
   - The array S'_L and, when ChromaArrayType is not equal to 0, the arrays S'_{Cb} and S'_{Cr} are assigned to the array S_L and, when ChromaArrayType is not equal to 0, the arrays S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.
3. When sps_alf_enabled_flag is equal to 1, the following applies:
   - The adaptive loop filter process as specified in clause 8.5.4.1 is invoked with the reconstructed picture sample arrays S_L, S_{Cb} and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L, S'_{Cb} and S'_{Cr} after sample adaptive offset as outputs.
   - The arrays S'_L, S'_{Cb} and S'_{Cr} are assigned to the arrays S_L, S_{Cb} and S_{Cr} (which represent the decoded picture), respectively.

8.5.2 Deblocking filter process

8.5.2.1 General

Inputs to this process are the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

Outputs of this process are the modified reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

The vertical edges in a picture are filtered first. Then the horizontal edges in a picture are filtered with samples modified by the vertical edge filtering process as input. The vertical and horizontal edges in the CTBs of each CTU are processed separately on a coding unit basis. The vertical edges of the coding blocks in a coding unit are filtered starting with the edge on the left-hand side of the coding blocks proceeding through the edges towards the right-hand side of the coding blocks in their geometrical order. The horizontal edges of the coding blocks in a coding unit are filtered starting with the edge on the top of the coding blocks proceeding through the edges towards the bottom of the coding blocks in their geometrical order.

NOTE – Although the filtering process is specified on a picture basis in this Specification, the filtering process can be implemented on a coding unit basis with an equivalent result, provided the decoder properly accounts for the processing dependency order so as to produce the same output values.

The deblocking filter process is applied to all coding subblock edges and transform block edges of a picture, except the following types of edges:

– Edges that are at the boundary of the picture,

– Edges that coincide with tile boundaries when loop_filter_across_tiles_enabled_flag is equal to 0,

– Edges that coincide with upper or left boundaries of slices with slice_loop_filter_across_slices_enabled_flag equal to 0 or slice_deblocking_filter_disabled_flag equal to 1,
– Edges within slices with slice_deblocking_filter_disabled_flag equal to 1,
– Edges that do not correspond to 8x8 sample grid boundaries of the considered component,
– Edges within chroma components for which both sides of the edge use inter prediction,
– Edges of chroma transform blocks that are not edges of the associated transform unit.

[Ed. (BB): Adapt syntax once tiles are integrated.]

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 8-12.

Table 8-12 – Name of association to edgeType

edgeType	Name of edgeType
0 (vertical edge)	EDGE_VER
1 (horizontal edge)	EDGE_HOR

When slice_deblocking_filter_disabled_flag of the current slice is equal to 0, the following applies:

– The variable treeType is derived as follows:
  – If slice_type is equal to I and qtbtt_dual_tree_intra_flag is equal to 1, treeType is set equal to DUAL_TREE_LUMA.
  – Otherwise, treeType is set equal to SINGLE_TREE.
– The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.
– The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the modifed reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 or treeType is equal to SINGLE_TREE, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.
– When slice_type is equal to I and qtbtt_dual_tree_intra_flag is equal to 1, the following applies:
  – The variable treeType is set equal to DUAL_TREE_CHROMA
  – The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.
  – The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.5.2.2 with the variable treeType, the modifed reconstructed picture after deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, and the variable edgeType set equal to EDGE_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$ as outputs.

8.5.2.2 Deblocking filter process for one direction

Inputs to this process are:

– the variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,
– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

For each coding unit with coding block width log2CbW, coding block height log2CbH and location of top-left sample of the coding block (xCb, yCb), when edgeType is equal to EDGE_VER and xCb % 8 is equal 0 or when edgeType is equal to EDGE_HOR and yCb % 8 is equal to 0, the edges are filtered by the following ordered steps:

1. The coding block width nCbW is set equal to 1 << log2CbW and the coding block height nCbH is set equal to 1 << log2CbH
2. The variable filterEdgeFlag is derived as follows:
   – If edgeType is equal to EDGE_VER and one or more of the following conditions are true, filterEdgeFlag is set equal to 0:
     – The left boundary of the current coding block is the left boundary of the picture.
     – The left boundary of the current coding block is the left boundary of the tile and loop_filter_across_tiles_enabled_flag is equal to 0.
     – The left boundary of the current coding block is the left boundary of the slice and slice_loop_filter_across_slices_enabled_flag is equal to 0.
   – Otherwise if edgeType is equal to EDGE_HOR and one or more of the following conditions are true, the variable filterEdgeFlag is set equal to 0:
     – The top boundary of the current luma coding block is the top boundary of the picture.
     – The top boundary of the current coding block is the top boundary of the tile and loop_filter_across_tiles_enabled_flag is equal to 0.
     – The top boundary of the current coding block is the top boundary of the slice and slice_loop_filter_across_slices_enabled_flag is equal to 0.
   – Otherwise, filterEdgeFlag is set equal to 1.

   [Ed. (BB): Adapt syntax once tiles are integrated.]
3. All elements of the two-dimensional (nCbW)x(nCbH) array edgeFlags are initialized to be equal to zero.
4. The derivation process of transform block boundary specified in clause 8.5.2.3 is invoked with the location (xB0, yB0) set equal to (0, 0), the block width nTbW set equal to nCbW, the block height nTbH set equal to nCbH, the variable treeType, the variable filterEdgeFlag, the array edgeFlags, and the variable edgeType as inputs, and the modified array edgeFlags as output.
5. The derivation process of coding subblock boundary specified in clause 8.5.2.4 is invoked with the location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the array edgeFlags, and the variable edgeType as inputs, and the modified array edgeFlags as output.
6. The picture sample array recPicture is derived as follows:
   – If treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, recPicture is set equal to the reconstructed luma picture sample array prior to deblocking $recPicture_L$.
   – Otherwise (treeType is equal to DUAL_TREE_CHROMA), recPicture is set equal to the reconstructed chroma picture sample array prior to deblocking $recPicture_{Cb}$.
7. The derivation process of the boundary filtering strength specified in clause 8.5.2.5 is invoked with the picture sample array recPicture, the luma location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, the variable edgeType, and the array edgeFlags as inputs, and an (nCbW)x(nCbH) array verBs as output.
8. The edge filtering process is invoked as follows:
   – If edgeType is equal to EDGE_VER, the vertical edge filtering process for a coding unit as specified in clause 8.5.2.6.1 is invoked with the variable treeType, the reconstructed picture prior to deblocking, i.e.,

when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, the location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, and the array verBs as inputs, and the modified reconstructed picture, i.e., when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, as output.

– Otherwise if edgeType is equal to EDGE_HOR, the horizontal edge filtering process for a coding unit as specified in clause 8.5.2.6.2 is invoked with the variable treeType, the modified reconstructed picture prior to deblocking, i.e., when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, the location (xCb, yCb), the coding block width nCbW, the coding block height nCbH, and the array horBs as inputs and the modified reconstructed picture, i.e., when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$ and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$, as output.

8.5.2.3 Derivation process of transform block boundary

Inputs to this process are:

– a location (xB0, yB0) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable nTbW specifying the width of the current block,

– a variable nTbH specifying the height of the current block,

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– a variable filterEdgeFlag,

– a two-dimensional (nCbW)x(nCbH) array edgeFlags,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbW)x(nCbH) array edgeFlags.

The maximum transform block size maxTbSize is derived as follows:

$$maxTbSize = (treeType == DUAL\_TREE\_CHROMA) ? MaxTbSizeY / 2 : MaxTbSizeY \quad (8\text{-}516)$$

Depending on maxTbSize, the following applies:

– If nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables newTbW and newTbH are derived as follows:

$$newTbW = (nTbW > maxTbSize) ? (nTbW / 2) : nTbW \quad (8\text{-}517)$$

$$newTbH = (nTbH > maxTbSize) ? (nTbH / 2) : nTbH \quad (8\text{-}518)$$

2. The derivation process of transform block boundary as specified in this clause is invoked with the location (xB0, yB0), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags, and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

3. If nTbW is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

4. If nTbH is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location (xTb0, yTb0) set equal to (xTb0, yTb0 + newTbH), the variables nTbW set

equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

5. If nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the derivation process of transform block boundary as specified in this clause is invoked with the luma location (xTb0, yTb0) set equal to (xTb0 + newTbW, yTb0 + newTbH), the variables nTbW set equal to newTbW and nTbH set equal to newTbH, the variable filterEdgeFlag, the array edgeFlags and the variable edgeType as inputs, and the output is the modified version of array edgeFlags.

– Otherwise, the following applies:
  – If edgeType is equal to EDGE_VER, the value of edgeFlags[xB0][yB0 + k] for k = 0..nTbH − 1 is derived as follows:
    – If xB0 is equal to 0, edgeFlags[xB0][yB0 + k] is set equal to filterEdgeFlag.
    – Otherwise, edgeFlags[xB0][yB0 + k] is set equal to 1.
  – Otherwise (edgeType is equal to EDGE_HOR), the value of edgeFlags[xB0 + k][yB0] for k = 0..nTbW − 1 is derived as follows:
    – If yB0 is equal to 0, edgeFlags[xB0 + k][yB0] is set equal to filterEdgeFlag.
    – Otherwise, edgeFlags[xB0 + k][yB0] is set equal to 1.

8.5.2.4 Derivation process of coding subblock boundary

Inputs to this process are:

– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
– a variable nCbW specifying the width of the current coding block,
– a variable nCbH specifying the height of the current coding block,
– a two-dimensional (nCbW)x(nCbH) array edgeFlags,
– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbW)x(nCbH) array edgeFlags.

The number of coding subblock in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

– If CuPredMode[xCb][yCb] == MODE_INTRA, numSbX and numSbY are both set equal to 1.
– Otherwise, numSbX and numSbY are set equal to NumSbX[xCb][yCb] and NumSbY[xCb][yCb], respectively.

Depending on the value of edgeType the following applies:

– If edgeType is equal to EDGE_VER and numSbX is greater than 1, the following applies for i = 1..min((nCbW / 8) − 1, numSbX − 1), k = 0..nCbH − 1:

$$edgeFlags[i * Max(8, nCbW / numSbX)][k] = 1 \quad (8\text{-}519)$$

– Otherwise if edgeType is equal to EDGE_HOR and numSbY is greater than 1, the following applies for j = 1..min((nCbH / 8) − 1, numSbY − 1), k = 0..nCbW − 1:

$$edgeFlags[k][j * Max(8, nCbH / numSbY)] = 1 \quad (8\text{-}520)$$

8.5.2.5 Derivation process of boundary filtering strength

Inputs to this process are:

– a picture sample array recPicture,
– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
– a variable nCbW specifying the width of the current coding block,

– a variable nCbH specifying the height of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– a two-dimensional (nCbW)x(nCbH) array edgeFlags.

Output of this process is a two-dimensional (nCbW)x(nCbH) array bS specifying the boundary filtering strength.

The variables xD_i, yD_j, xN and yN are derived as follows:

– If edgeType is equal to EDGE_VER, xD_i is set equal to (i << 3), yD_j is set equal to (j << 2), xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to (nCbH / 4) − 1.

– Otherwise (edgeType is equal to EDGE_HOR), xD_i is set equal to (i << 2), yD_j is set equal to (j << 3), xN is set equal to (nCbW / 4) − 1 and yN is set equal to Max(0, (nCbH / 8) − 1).

For xD_i with i = 0..xN and yD_j with j = 0..yN, the following applies:

– If edgeFlags[xD_i][yD_j] is equal to 0, the variable bS[xD_i][yD_j] is set equal to 0.

– Otherwise, the following applies:

  – The sample values p_0 and q_0 are derived as follows:

    – If edgeType is equal to EDGE_VER, p_0 is set equal to recPicture [xCb + xD_i − 1][yCb + yD_j] and q_0 is set equal to recPicture [xCb + xD_i][yCb + yD_j].

    – Otherwise (edgeType is equal to EDGE_HOR), p_0 is set equal to recPicture [xCb + xD_i][yCb + yD_j − 1] and q_0 is set equal to recPicture [xCb + xD_i][yCb + yD_j].

  – The variable bS[xD_i][yD_j] is derived as follows:

    – If the sample p_0 or q_0 is in the coding block of a coding unit coded with intra prediction mode, bS[xD_i][yD_j] is set equal to 2.

    – Otherwise, if the block edge is also a transform block edge and the sample p_0 or q_0 is in a transform block which contains one or more non-zero transform coefficient levels, bS[xD_i][yD_j] is set equal to 1.

    – Otherwise, if one or more of the following conditions are true, bS[xD_i][yD_j] is set equal to 1:

      – For the prediction of the coding subblock containing the sample p_0 different reference pictures or a different number of motion vectors are used than for the prediction of the coding subblock containing the sample q_0.

        NOTE 1 – The determination of whether the reference pictures used for the two coding sublocks are the same or different is based only on which pictures are referenced, without regard to whether a prediction is formed using an index into reference picture list 0 or an index into reference picture list 1, and also without regard to whether the index position within a reference picture list is different.

        NOTE 2 – The number of motion vectors that are used for the prediction of a coding subblock with top-left sample covering (xSb, ySb), is equal to PredFlagL0[xSb][ySb] + PredFlagL1[xSb][ySb].

      – One motion vector is used to predict the coding subblock containing the sample p_0 and one motion vector is used to predict the coding subblock containing the sample q_0, and the absolute difference between the horizontal or vertical component of the motion vectors used is greater than or equal to 4 in units of quarter luma samples.

      – Two motion vectors and two different reference pictures are used to predict the coding subblock containing the sample p_0, two motion vectors for the same two reference pictures are used to predict the coding subblock containing the sample q_0 and the absolute difference between the horizontal or vertical component of the two motion vectors used in the prediction of the two coding subblocks for the same reference picture is greater than or equal to 4 in units of quarter luma samples.

      – Two motion vectors for the same reference picture are used to predict the coding subblock containing the sample p_0, two motion vectors for the same reference picture are used to predict the coding subblock containing the sample q_0 and both of the following conditions are true:

        – The absolute difference between the horizontal or vertical component of list 0 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in quarter luma samples, or the absolute difference between the horizontal or vertical component of the

list 1 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in units of quarter luma samples.

– The absolute difference between the horizontal or vertical component of list 0 motion vector used in the prediction of the coding subblock containing the sample p_0 and the list 1 motion vector used in the prediction of the coding subblock containing the sample q_0 is greater than or equal to 4 in units of quarter luma samples, or the absolute difference between the horizontal or vertical component of the list 1 motion vector used in the prediction of the coding subblock containing the sample p_0 and list 0 motion vector used in the prediction of the coding subblock containing the sample q_0 is greater than or equal to 4 in units of quarter luma samples.

– Otherwise, the variable bS[xD_i][yD_j] is set equal to 0.

8.5.2.6 Edge filtering process

8.5.2.6.1 Vertical edge filtering process

Inputs to this process are:

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$,

– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,

– a variable nCbW specifying the width of the current coding block,

– a variable nCbH specifying the height of the current coding block.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

When treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to (nCbH / 4) − 1.
2. For xD_k equal to k << 3 with k = 0..nN and yD_m equal to m << 2 with m = 0..yN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
     a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER,the boundary filtering strength bS[xD_k][yD_m], and the bit depth bD set equal to $BitDepth_Y$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.
     b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified luma picture sample array $recPicture_L$ as output.

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE the filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to Max(0, (nCbH / 8) − 1).

2. The variable edgeSpacing is set equal to 8 / SubWidthC.
3. The variable edgeSections is set equal to yN * (2 / SubHeightC).
4. For xD_k equal to k * edgeSpacing with k = 0..xN and yD_m equal to m << 2 with m = 0..edgeSections, the following applies:
   - When bS[xD_k * SubWidthC][yD_m * SubHeightC] is equal to 2 and (((xCb / SubWidthC + xD_k) >> 3) << 3) is equal to xCb / SubWidthC + xD_k, the following ordered steps apply:
     a. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER and a variable cQpPicOffset set equal to pps_cb_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
     b. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER and a variable cQpPicOffset set equal to pps_cr_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

When treeType is equal to DUAL_TREE_CHROMA, the filtering process for edges in the two chroma coding blocks of the current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to (nCbH / 4) − 1.
2. For xD_k equal to k << 3 with k = 0..xN and yD_m equal to m << 2 with m = 0..yN, the following applies:
   - When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
     a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the location of the chroma block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the boundary filtering strength bS[xD_k][yD_m], and the bit depth bD set equal to $BitDepth_C$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.
     b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
     c. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_VER, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

8.5.2.6.2 Horizontal edge filtering process

Inputs to this process are:

- a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,
- when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the reconstructed picture prior to deblocking, i.e., the array $recPicture_L$,
- when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$,
- a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
- a variable nCbW specifying the width of the current coding block,

– a variable nCbH specifying the height of the current coding block.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

– when treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the array $recPicture_L$,

– when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE or DUAL_TREE_CHROMA, the arrays $recPicture_{Cb}$ and $recPicture_{Cr}$.

When treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable yN is set equal to Max(0, (nCbH / 8) − 1) and xN is set equal to (nCbW / 4) − 1.
2. For yD_m equal to m << 3 with m = 0..yN and xD_k equal to k << 2 with k = 0..xN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
     a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the boundary filtering strength bS[xD_k][yD_m], and the bit depth bD set equal to $BitDepth_Y$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.
     b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the luma picture sample array $recPicture_L$, the location of the luma coding block (xCb, yCb), the luma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the decisions dEp, dEp and dEq, and the variable t_C as inputs, and the modified luma picture sample array $recPicture_L$ as output.

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE_TREE the filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable xN is set equal to Max(0, (nCbW / 8) − 1) and yN is set equal to Max(0, (nCbH / 8) − 1).
2. The variable edgeSpacing is set equal to 8 / SubHeightC.
3. The variable edgeSections is set equal to xN * (2 / SubWidthC).
4. For yD_m equal to m * edgeSpacing with m = 0..yN and xD_k equal to k << 2 with k = 0..edgeSections, the following applies:
   – When bS[xD_k * SubWidthC][yD_m * SubHeightC] is equal to 2 and (((yCb / SubHeightC + yD_m) >> 3) << 3) is equal to yCb / SubHeightC + yD_m, the following ordered steps apply:
     a. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR and a variable cQpPicOffset set equal to pps_cb_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
     b. The filtering process for chroma block edges as specified in clause 8.5.2.6.5 is invoked with the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb / SubWidthC, yCb / SubHeightC), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR and a variable cQpPicOffset set equal to pps_cr_qp_offset as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

When treeType is equal to DUAL_TREE_CHROMA, the filtering process for edges in the two chroma coding blocks of the current coding unit consists of the following ordered steps:

1. The variable yN is set equal to Max(0, (nCbH / 8) − 1) and xN is set equal to (nCbW / 4) − 1.
2. For yD_m equal to m << 3 with m = 0..yN and xD_k equal to k << 2 with k = 0..xN, the following applies:
   – When bS[xD_k][yD_m] is greater than 0, the following ordered steps apply:
     a. The decision process for block edges as specified in clause 8.5.2.6.3 is invoked with treeType, the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the location of the chroma block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the boundary filtering strength bS[xD_k][yD_m], and

the bit depth bD set equal to $BitDepth_C$ as inputs, and the decisions dE, dEp and dEq, and the variable t_C as outputs.

b. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.

c. The filtering process for block edges as specified in clause 8.5.2.6.4 is invoked with the picture sample array recPicture set equal to the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block (xCb, yCb), the chroma location of the block (xD_k, yD_m), a variable edgeType set equal to EDGE_HOR, the decisions dE, dEp and dEq, and the variable t_C as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

8.5.2.6.3 Decision process for block edges

Inputs to this process are:

– a variable treeType specifying whether a single tree (SINGLE_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL_TREE_LUMA) or chroma components (DUAL_TREE_CHROMA) are currently processed,

– a picture sample array recPicture,

– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,

– a location (xBl, yBl) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– a variable bS specifying the boundary filtering strength,

– a variable bD specifying the bit depth of the current component.

Outputs of this process are:

– the variables dE, dEp and dEq containing decisions,

– the variable t_C.

If edgeType is equal to EDGE_VER, the sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0 and 3 are derived as follows:

$$q_{i,k} = recPicture_L[\ xCb + xBl + i\ ][\ yCb + yBl + k\ ] \quad (8\text{-}521)$$

$$p_{i,k} = recPicture_L[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] \quad (8\text{-}522)$$

Otherwise (edgeType is equal to EDGE_HOR), the sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0 and 3 are derived as follows:

$$q_{i,k} = recPicture[\ xCb + xBl + k\ ][\ yCb + yBl + i\ ] \quad (8\text{-}523)$$

$$p_{i,k} = recPicture[\ xCb + xBl + k\ ][\ yCb + yBl - i - 1\ ] \quad (8\text{-}524)$$

The variable qpOffset is derived as follows:

– If sps_ladf_enabled_flag is equal to 1 and treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, the following applies:

  – The variable lumaLevel of the reconstructed luma level is derived as follow:

$$lumaLevel = (\ (\ p0,0 + p0,3 + q0,0 + q0,3\ ) >> 2\ ), \quad (8\text{-}525)$$

  – The variable qpOffset is set equal to sps_ladf_lowest_interval_qp_offset and modified as follows:

```
for( i = 0; i < sps_num_ladf_intervals_minus2 + 1; i++ ) {
    if( lumaLevel > SpsLadfIntervalLowerBound[ i + 1 ] )
```

```
            qpOffset = sps_ladf_qp_offset[ i ]                                    (8-526)
        else
            break
    }
```

– Otherwise (treeType is equal to DUAL_TREE_CHROMA), qpOffset is set equal to 0.

The variables Qp_Q and Qp_P are derived as follows:

– If treeType is equal to SINGLE_TREE or DUAL_TREE_LUMA, Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

– Otherwise (treeType is equal to DUAL_TREE_CHROMA), Qp_Q and Qp_P are set equal to the Qp_C values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

The variable qP is derived as follows:

$$qP = ((Qp_Q + Qp_P + 1) >> 1) + qpOffset \quad (8\text{-}527)$$

The value of the variable β′ is determined as specified in Table 8-13 based on the quantization parameter Q derived as follows:

$$Q = Clip3(0, 63, qP + (slice\_beta\_offset\_div2 << 1)) \quad (8\text{-}528)$$

where slice_beta_offset_div2 is the value of the syntax element slice_beta_offset_div2 for the slice that contains sample $q_{0,0}$.

The variable β is derived as follows:

$$\beta = \beta' * (1 << (bD - 8)) \quad (8\text{-}529)$$

The value of the variable t_C' is determined as specified in Table 8-13 based on the quantization parameter Q derived as follows:

$$Q = Clip3(0, 65, qP + 2 * (bS - 1) + (slice\_tc\_offset\_div2 << 1)) \quad (8\text{-}530)$$

where slice_tc_offset_div2 is the value of the syntax element slice_tc_offset_div2 for the slice that contains sample $q_{0,0}$.

The variable t_C is derived as follows:

$$t_C = t_C' * (1 << (bD - 8)) \quad (8\text{-}531)$$

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE_VER, the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

$$dp0 = Abs(p_{2,0} - 2 * p_{1,0} + p_{0,0}) \quad (8\text{-}532)$$

$$dp3 = Abs(p_{2,3} - 2 * p_{1,3} + p_{0,3}) \quad (8\text{-}533)$$

$$dq0 = Abs(q_{2,0} - 2 * q_{1,0} + q_{0,0}) \quad (8\text{-}534)$$

$$dq3 = Abs(q_{2,3} - 2 * q_{1,3} + q_{0,3}) \quad (8\text{-}535)$$

$$dpq0 = dp0 + dq0 \quad (8\text{-}536)$$

$$dpq3 = dp3 + dq3 \quad (8\text{-}537)$$

$$dp = dp0 + dp3 \quad (8\text{-}538)$$

$$dq = dq0 + dq3 \quad (8\text{-}539)$$

$$d = dpq0 + dpq3 \quad (8\text{-}540)$$

2. The variables dE, dEp and dEq are set equal to 0.

3. When d is less than β, the following ordered steps apply:
   a. The variable dpq is set equal to 2 * dpq0.
   b. For the sample location (xCb + xBl, yCb + yBl), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,0}$, $p_{3,0}$, $q_{0,0}$, and $q_{3,0}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam0.
   c. The variable dpq is set equal to 2 * dpq3.
   d. For the sample location (xCb + xBl, yCb + yBl + 3), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,3}$, $p_{3,3}$, $q_{0,3}$, and $q_{3,3}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam3.
   e. The variable dE is set equal to 1.
   f. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
   g. When dp is less than (β + (β >> 1)) >> 3, the variable dEp is set equal to 1.
   h. When dq is less than (β + (β >> 1)) >> 3, the variable dEq is set equal to 1.

– Otherwise (edgeType is equal to EDGE_HOR), the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

$$dp0 = Abs(p_{2,0} - 2 * p_{1,0} + p_{0,0}) \quad (8\text{-}541)$$

$$dp3 = Abs(p_{2,3} - 2 * p_{1,3} + p_{0,3}) \quad (8\text{-}542)$$

$$dq0 = Abs(q_{2,0} - 2 * q_{1,0} + q_{0,0}) \quad (8\text{-}543)$$

$$dq3 = Abs(q_{2,3} - 2 * q_{1,3} + q_{0,3}) \quad (8\text{-}544)$$

$$dpq0 = dp0 + dq0 \quad (8\text{-}545)$$

$$dpq3 = dp3 + dq3 \quad (8\text{-}546)$$

$$dp = dp0 + dp3 \quad (8\text{-}547)$$

$$dq = dq0 + dq3 \quad (8\text{-}548)$$

$$d = dpq0 + dpq3 \quad (8\text{-}549)$$

2. The variables dE, dEp and dEq are set equal to 0.
3. When d is less than β, the following ordered steps apply:
   a. The variable dpq is set equal to 2 * dpq0.
   b. For the sample location (xCb + xBl, yCb + yBl), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,0}$, $p_{3,0}$, $q_{0,0}$ and $q_{3,0}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam0.
   c. The variable dpq is set equal to 2 * dpq3.
   d. For the sample location (xCb + xBl + 3, yCb + yBl), the decision process for a sample as specified in clause 8.5.2.6.6 is invoked with sample values $p_{0,3}$, $p_{3,3}$, $q_{0,3}$ and $q_{3,3}$, the variables dpq, β and t_C as inputs, and the output is assigned to the decision dSam3.
   e. The variable dE is set equal to 1.
   f. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
   g. When dp is less than (β + (β >> 1)) >> 3, the variable dEp is set equal to 1.
   h. When dq is less than (β + (β >> 1)) >> 3, the variable dEq is set equal to 1.

Table 8-13 – Derivation of threshold variables β′ and t_C′ from input Q

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
β′	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
t_C′	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Q	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
β′	7	8	9	10	11	12	13	14	15	16	17	18	20	22	24	26	28
t_C′	0	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3
Q	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
β′	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
t_C′	3	4	4	4	5	5	6	6	7	8	9	10	11	13	14	16	18
Q	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65		
β′	64	66	68	70	72	74	76	78	80	82	84	86	88	-	-		
t_C′	20	22	25	28	31	35	39	44	50	56	63	70	79	88	99		

8.5.2.6.4 Filtering process for block edges

Inputs to this process are:

– a picture sample array recPicture,

– a location (xCb, yCb) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,

– a location (xBl, yBl) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– the variables dE, dEp and dEq containing decisions,

– the variable t_C.

Output of this process is the modified picture sample array recPicture.

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE_VER, the following ordered steps apply:

1. The sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0..3 are derived as follows:

$$q_{i,k} = recPicture_L[\ xCb + xBl + i\ ][\ yCb + yBl + k\ ] \quad (8\text{-}550)$$

$$p_{i,k} = recPicture_L[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] \quad (8\text{-}551)$$

2. When dE is not equal to 0, for each sample location (xCb + xBl, yCb + yBl + k), k = 0..3, the following ordered steps apply:

   a. The filtering process for a sample as specified in clause 8.5.2.6.7 is invoked with the sample values $p_{i,k}$, $q_{i,k}$ with i = 0..3, the locations (xP_i, yP_i) set equal to (xCb + xBl − i − 1, yCb + yBl + k) and (xQ_i, yQ_i) set equal to (xCb + xBl + i, yCb + yBl + k) with i = 0..2, the decision dE, the variables dEp and dEq and the variable t_C as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values p_i' and q_j' as outputs.

   b. When nDp is greater than 0, the filtered sample values p_i' with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[\ xCb + xBl - i - 1\ ][\ yCb + yBl + k\ ] = p_i' \quad (8\text{-}552)$$

   c. When nDq is greater than 0, the filtered sample values q_j' with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[\ xCb + xBl + j\ ][\ yCb + yBl + k\ ] = q_j' \quad (8\text{-}553)$$

– Otherwise (edgeType is equal to EDGE_HOR), the following ordered steps apply:

1. The sample values $p_{i,k}$ and $q_{i,k}$ with i = 0..3 and k = 0..3 are derived as follows:

$$q_{i,k} = recPicture_L[xCb + xBl + k][yCb + yBl + i] \quad (8\text{-}554)$$

$$p_{i,k} = recPicture_L[xCb + xBl + k][yCb + yBl − i − 1] \quad (8\text{-}555)$$

2. When dE is not equal to 0, for each sample location (xCb + xBl + k, yCb + yBl), k = 0..3, the following ordered steps apply:

   a. The filtering process for a sample as specified in clause 8.5.2.6.7 is invoked with the sample values $p_{i,k}$, $q_{i,k}$ with i = 0..3, the locations (xP_i, yP_i) set equal to (xCb + xBl + k, yCb + yBl − i − 1) and (xQ_i, yQ_i) set equal to (xCb + xBl + k, yCb + yBl + i) with i = 0..2, the decision dE, the variables dEp and dEq, and the variable t_C as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values p_i' and q_j' as outputs.

   b. When nDp is greater than 0, the filtered sample values p_i' with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[xCb + xBl + k][yCb + yBl − i − 1] = p_i' \quad (8\text{-}556)$$

   c. When nDq is greater than 0, the filtered sample values q_j' with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPicture as follows:

$$recPicture[xCb + xBl + k][yCb + yBl + j] = q_j' \quad (8\text{-}557)$$

8.5.2.6.5 Filtering process for chroma block edges

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

– a chroma picture sample array s′,

– a chroma location (xCb, yCb) specifying the top-left sample of the current chroma coding block relative to the top-left chroma sample of the current picture,

– a chroma location (xBl, yBl) specifying the top-left sample of the current chroma block relative to the top-left sample of the current chroma coding block,

– a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,

– a variable cQpPicOffset specifying the picture-level chroma quantization parameter offset.

Output of this process is the modified chroma picture sample array s′.

If edgeType is equal to EDGE_VER, the values p_i and q_i with i = 0..1 and k = 0..3 are derived as follows:

$$q_{i,k} = s'[xCb + xBl + i][yCb + yBl + k] \quad (8\text{-}558)$$

$$p_{i,k} = s'[xCb + xBl − i − 1][yCb + yBl + k] \quad (8\text{-}559)$$

Otherwise (edgeType is equal to EDGE_HOR), the sample values p_i and q_i with i = 0..1 and k = 0..3 are derived as follows:

$$q_{i,k} = s'[xCb + xBl + k][yCb + yBl + i] \quad (8\text{-}560)$$

$$p_{i,k} = s'[xCb + xBl + k][yCb + yBl − i − 1] \quad (8\text{-}561)$$

The variables Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

If ChromaArrayType is equal to 1, the variable Qp_C is determined as specified in Table 8-10 based on the index qPi derived as follows:

$$qPi = ((Qp_Q + Qp_P + 1) >> 1) + cQpPicOffset \quad (8\text{-}562)$$

Otherwise (ChromaArrayType is greater than 1), the variable Qp_C is set equal to Min(qPi, 63).

NOTE – The variable cQpPicOffset provides an adjustment for the value of pps_cb_qp_offset or pps_cr_qp_offset, according to whether the filtered chroma component is the Cb or Cr component. However, to avoid the need to vary the amount of the adjustment within the picture, the filtering process does not include an adjustment for the value of slice_cb_qp_offset or slice_cr_qp_offset.

The value of the variable t_C' is determined as specified in Table 8-13 based on the chroma quantization parameter Q derived as follows:

$$Q = Clip3(0, 65, Qp_C + 2 + (slice\_tc\_offset\_div2 << 1)) \quad (8\text{-}563)$$

where slice_tc_offset_div2 is the value of the syntax element slice_tc_offset_div2 for the slice that contains sample $q_{0,0}$.

The variable t_C is derived as follows:

$$t_C = t_C' * (1 << (BitDepth_C - 8)) \quad (8\text{-}564)$$

Depending on the value of edgeType, the following applies:

– If edgeType is equal to EDGE_VER, for each sample location (xCb + xBl, yCb + yBl + k), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.5.2.6.8 is invoked with the sample values $p_{i,k}$, $q_{i,k}$, with i = 0..1, the locations (xCb + xBl − 1, yCb + yBl + k) and (xCb + xBl, yCb + yBl + k) and the variable t_C as inputs, and the filtered sample values p_0' and q_0' as outputs.
2. The filtered sample values p_0' and q_0' replace the corresponding samples inside the sample array s′ as follows:

$$s'[xCb + xBl][yCb + yBl + k] = q_0' \quad (8\text{-}565)$$

$$s'[xCb + xBl - 1][yCb + yBl + k] = p_0' \quad (8\text{-}566)$$

– Otherwise (edgeType is equal to EDGE_HOR), for each sample location (xCb + xBl + k, yCb + yBl), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.5.2.6.8 is invoked with the sample values $p_{i,k}$, $q_{i,k}$, with i = 0..1, the locations (xCb + xBl + k, yCb + yBl − 1) and (xCb + xBl + k, yCb + yBl), and the variable t_C as inputs, and the filtered sample values p_0' and q_0' as outputs.
2. The filtered sample values p_0' and q_0' replace the corresponding samples inside the sample array s′ as follows:

$$s'[xCb + xBl + k][yCb + yBl] = q_0' \quad (8\text{-}567)$$

$$s'[xCb + xBl + k][yCb + yBl - 1] = p_0' \quad (8\text{-}568)$$

8.5.2.6.6 Decision process for a sample

Inputs to this process are:

– the sample values p_0, p_3, q_0 and q_3,

– the variables dpq, β and t_C.

Output of this process is the variable dSam containing a decision.

The variable dSam is specified as follows:

– If dpq is less than (β >> 2), Abs($p_3 - p_0$) + Abs($q_0 - q_3$) is less than (β >> 3) and Abs($p_0 - q_0$) is less than (5 * t_C + 1) >> 1, dSam is set equal to 1.

– Otherwise, dSam is set equal to 0.

8.5.2.6.7 Filtering process for a sample

Inputs to this process are:

– the sample values p_i and q_i with i = 0..3,

– the locations of p_i and q_i, (xP_i, yP_i) and (xQ_i, yQ_i) with i = 0..2,

– a variable dE,

– the variables dEp and dEq containing decisions to filter samples p1 and q1, respectively,

– a variable t_C.

Outputs of this process are:

– the number of filtered samples nDp and nDq,

– the filtered sample values p_i' and q_j' with i = 0..nDp − 1, j = 0..nDq − 1.

Depending on the value of dE, the following applies:

– If the variable dE is equal to 2, nDp and nDq are both set equal to 3 and the following strong filtering applies:

$$p_0' = \text{Clip3}(p_0 - 2 * t_C, p_0 + 2 * t_C, (p_2 + 2 * p_1 + 2 * p_0 + 2 * q_0 + q_1 + 4) >> 3) \quad (8\text{-}569)$$

$$p_1' = \text{Clip3}(p_1 - 2 * t_C, p_1 + 2 * t_C, (p_2 + p_1 + p_0 + q_0 + 2) >> 2) \quad (8\text{-}570)$$

$$p_2' = \text{Clip3}(p_2 - 2 * t_C, p_2 + 2*t_C, (2 * p_3 + 3 * p_2 + p_1 + p_0 + q_0 + 4) >> 3) \quad (8\text{-}571)$$

$$q_0' = \text{Clip3}(q_0 - 2 * t_C, q_0 + 2 * t_C, (p_1 + 2 * p_0 + 2 * q_0 + 2 * q_1 + q_2 + 4) >> 3) \quad (8\text{-}572)$$

$$q_1' = \text{Clip3}(q_1 - 2 * t_C, q_1 + 2 * t_C, (p_0 + q_0 + q_1 + q_2 + 2) >> 2) \quad (8\text{-}573)$$

$$q_2' = \text{Clip3}(q_2 - 2 * t_C, q_2 + 2 * t_C, (p_0 + q_0 + q_1 + 3 * q_2 + 2 * q_3 + 4) >> 3) \quad (8\text{-}574)$$

– Otherwise, nDp and nDq are set both equal to 0 and the following weak filtering applies:

  – The following applies:

$$\Delta = (9 * (q_0 - p_0) - 3 * (q_1 - p_1) + 8) >> 4 \quad (8\text{-}575)$$

  – When Abs(Δ) is less than $t_C * 10$, the following ordered steps apply:

    1. The filtered sample values p_0' and q_0' are specified as follows:

$$\Delta = \text{Clip3}(-t_C, t_C, \Delta) \quad (8\text{-}576)$$

$$p_0' = \text{Clip1}_Y(p_0 + \Delta) \quad (8\text{-}577)$$

$$q_0' = \text{Clip1}_Y(q_0 - \Delta) \quad (8\text{-}578)$$

    2. When dEp is equal to 1, the filtered sample value p_1' is specified as follows:

$$\Delta p = \text{Clip3}(-(t_C >> 1), t_C >> 1, (((p_2 + p_0 + 1) >> 1) - p_1 + \Delta) >> 1) \quad (8\text{-}579)$$

$$p_1' = \text{Clip1}_Y(p_1 + \Delta p) \quad (8\text{-}580)$$

    3. When dEq is equal to 1, the filtered sample value q_1' is specified as follows:

$$\Delta q = \text{Clip3}(-(t_C >> 1), t_C >> 1, (((q_2 + q_0 + 1) >> 1) - q_1 - \Delta) >> 1) \quad (8\text{-}581)$$

$$q_1' = \text{Clip1}_Y(q_1 + \Delta q) \quad (8\text{-}582)$$

    4. nDp is set equal to dEp + 1 and nDq is set equal to dEq + 1.

When nDp is greater than 0 and one or more of the following conditions are true, nDp is set equal to 0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xP_0][yP_0] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample p_0 is equal to 1.

When nDq is greater than 0 and one or more of the following conditions are true, nDq is set equal to 0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xQ_0][yQ_0] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample q_0 is equal to 1.

8.5.2.6.8 Filtering process for a chroma sample

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

– the chroma sample values p_i and q_i with i = 0..1,

– the chroma locations of p_0 and q_0, (xP_0, yP_0) and (xQ_0, yQ_0),

– a variable t_C.

Outputs of this process are the filtered sample values p_0' and q_0'.

The filtered sample values p_0' and q_0' are derived as follows:

$$\Delta = \text{Clip3}(-t_C, t_C, ((((q_0 - p_0) << 2) + p_1 - q_1 + 4) >> 3)) \quad (8\text{-}583)$$

$$p_0' = \text{Clip1}_C(p_0 + \Delta) \quad (8\text{-}584)$$

$$q_0' = \text{Clip1}_C(q_0 - \Delta) \quad (8\text{-}585)$$

When one or more of the following conditions are true, the filtered sample value, p_0' is substituted by the corresponding input sample value p_0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xP_0 * SubWidthC][yP_0 * SubHeightC] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample p_0 is equal to 1.

When one or more of the following conditions are true, the filtered sample value, q_0' is substituted by the corresponding input sample value q_0:

– pcm_loop_filter_disabled_flag is equal to 1 and pcm_flag[xQ_0 * SubWidthC][yQ_0 * SubHeightC] is equal to 1.

– cu_transquant_bypass_flag of the coding unit that includes the coding block containing the sample q_0 is equal to 1.

8.5.3 Sample adaptive offset process

8.5.3.1 General

Inputs to this process are the reconstructed picture sample array prior to sample adaptive offset recPicture_L and, when ChromaArrayType is not equal to 0, the arrays recPicture_{Cb} and recPicture_{Cr}.

Outputs of this process are the modified reconstructed picture sample array after sample adaptive offset saoPicture_L and, when ChromaArrayType is not equal to 0, the arrays saoPicture_{Cb} and saoPicture_{Cr}.

This process is performed on a CTB basis after the completion of the deblocking filter process for the decoded picture.

The sample values in the modified reconstructed picture sample array saoPicture_L and, when ChromaArrayType is not equal to 0, the arrays saoPicture_{Cb} and saoPicture_{Cr} are initially set equal to the sample values in the reconstructed picture sample array recPicture_L and, when ChromaArrayType is not equal to 0, the arrays recPicture_{Cb} and recPicture_{Cr}, respectively.

For every CTU with CTB location (rx, ry), where rx = 0..PicWidthInCtbsY − 1 and ry = 0..PicHeightInCtbsY − 1, the following applies:

– When slice_sao_luma_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.5.3.2 is invoked with recPicture set equal to recPicture_L, cIdx set equal to 0, (rx, ry), and both nCtbSw and nCtbSh set equal to CtbSizeY as inputs, and the modified luma picture sample array saoPicture_L as output.

– When ChromaArrayType is not equal to 0 and slice_sao_chroma_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.5.3.2 is invoked with recPicture set equal to recPicture_{Cb}, cIdx set equal to 1, (rx, ry), nCtbSw set equal to (1 << CtbLog2SizeY) / SubWidthC and nCtbSh set equal to (1 << CtbLog2SizeY) / SubHeightC as inputs, and the modified chroma picture sample array saoPicture_{Cb} as output.

– When ChromaArrayType is not equal to 0 and slice_sao_chroma_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.5.3.2 is invoked with recPicture set equal to recPicture_{Cr}, cIdx set equal to 2, (rx, ry), nCtbSw set equal to (1 << CtbLog2SizeY) / SubWidthC and nCtbSh set equal to (1 << CtbLog2SizeY) / SubHeightC as inputs, and the modified chroma picture sample array saoPicture_{Cr} as output.

8.5.3.2 CTB modification process

Inputs to this process are:

– the picture sample array recPicture for the colour component cIdx,

– a variable cIdx specifying the colour component index,

– a pair of variables (rx, ry) specifying the CTB location,

– the CTB width nCtbSw and height nCtbSh.

Output of this process is a modified picture sample array saoPicture for the colour component cIdx.

The variable bitDepth is derived as follows:

– If cIdx is equal to 0, bitDepth is set equal to $BitDepth_Y$.

– Otherwise, bitDepth is set equal to $BitDepth_C$.

The location (xCtb, yCtb), specifying the top-left sample of the current CTB for the colour component cIdx relative to the top-left sample of the current picture component cIdx, is derived as follows:

$$(xCtb, yCtb) = (rx * nCtbSw, ry * nCtbSh) \quad (8\text{-}586)$$

The sample locations inside the current CTB are derived as follows:

$$(xS_i, yS_j) = (xCtb + i, yCtb + j) \quad (8\text{-}587)$$

$$(xY_i, yY_j) = (cIdx == 0) ? (xS_i, yS_j) : (xS_i * SubWidthC, yS_j * SubHeightC) \quad (8\text{-}588)$$

For all sample locations (xS_i, yS_j) and (xY_i, yY_j) with i = 0..nCtbSw − 1 and j = 0..nCtbSh − 1, depending on the values of pcm_loop_filter_disabled_flag, pcm_flag[xY_i][yY_j] and cu_transquant_bypass_flag of the coding unit which includes the coding block covering recPicture[xS_i][yS_j], the following applies:

– If one or more of the following conditions are true, saoPicture[xS_i][yS_j] is not modified:

  – pcm_loop_filter_disabled_flag and pcm_flag[xY_i][yY_j] are both equal to 1.

  – cu_transquant_bypass_flag is equal to 1.

  – SaoTypeIdx[cIdx][rx][ry] is equal to 0.

[Ed. (BB): Modify highlighted sections prending on future decision transform/quantizaion bypass.]

– Otherwise, if SaoTypeIdx[cIdx][rx][ry] is equal to 2, the following ordered steps apply:

  1. The values of hPos[k] and vPos[k] for k = 0..1 are specified in Table 8-14 based on SaoEoClass[cIdx][rx][ry].

  2. The variable edgeIdx is derived as follows:

     – The modified sample locations (xS_{ik}', yS_{jk}') and (xY_{ik}', yY_{jk}') are derived as follows:

$$(xS_{ik}', yS_{jk}') = (xS_i + hPos[k], yS_j + vPos[k]) \quad (8\text{-}589)$$

$$(xY_{ik}', yY_{jk}') = (cIdx == 0) ? (xS_{ik}', yS_{jk}') : (xS_{ik}' * SubWidthC, yS_{jk}' * SubHeightC) \quad (8\text{-}590)$$

     – If one or more of the following conditions for all sample locations (xS_{ik}', yS_{jk}') and (xY_{ik}', yY_{jk}') with k = 0..1 are true, edgeIdx is set equal to 0:

       – The sample at location (xS_{ik}', yS_{jk}') is outside the picture boundaries.

       – The sample at location (xS_{ik}', yS_{jk}') belongs to a different slice and one of the following two conditions is true:

         – MinTbAddrZs[xY_{ik}' >> MinTbLog2SizeY][yY_{jk}' >> MinTbLog2SizeY] is less than MinTbAddrZs[xY_i >> MinTbLog2SizeY][yY_j >> MinTbLog2SizeY] and slice_loop_filter_across_slices_enabled_flag in the slice which the sample recPicture[xS_i][yS_j] belongs to is equal to 0.

         – MinTbAddrZs[xY_i >> MinTbLog2SizeY][yY_j >> MinTbLog2SizeY] is less than MinTbAddrZs[xY_{ik}' >> MinTbLog2SizeY][yY_{jk}' >> MinTbLog2SizeY] and slice_loop_filter_across_slices_enabled_flag in the slice which the sample recPicture[xS_{ik}'][yS_{jk}'] belongs to is equal to 0.

– loop_filter_across_tiles_enabled_flag is equal to 0 and the sample at location (xS_{ik}', yS_{jk}') belongs to a different tile.

[Ed. (BB): Modify highlighted sections when tiles without slices are incorporated]

– Otherwise, edgeIdx is derived as follows:

– The following applies:

$$edgeIdx = 2 + Sign(recPicture[xS_i][yS_j] - recPicture[xS_i + hPos[0]][yS_j + vPos[0]]) + Sign(recPicture[xS_i][yS_j] - recPicture[xS_i + hPos[1]][yS_j + vPos[1]]) \quad (8\text{-}591)$$

– When edgeIdx is equal to 0, 1, or 2, edgeIdx is modified as follows:

$$edgeIdx = (edgeIdx == 2) ? 0 : (edgeIdx + 1) \quad (8\text{-}592)$$

3. The modified picture sample array saoPicture[xS_i][yS_j] is derived as follows:

$$saoPicture[xS_i][yS_j] = Clip3(0, (1 << bitDepth) - 1, recPicture[xS_i][yS_j] + SaoOffsetVal[cIdx][rx][ry][edgeIdx]) \quad (8\text{-}593)$$

– Otherwise (SaoTypeIdx[cIdx][rx][ry] is equal to 1), the following ordered steps apply:

1. The variable bandShift is set equal to bitDepth − 5.
2. The variable saoLeftClass is set equal to sao_band_position[cIdx][rx][ry].
3. The list bandTable is defined with 32 elements and all elements are initially set equal to 0. Then, four of its elements (indicating the starting position of bands for explicit offsets) are modified as follows:

```
for( k = 0; k < 4; k++ )
    bandTable[ ( k + saoLeftClass ) & 31 ] = k + 1
```
(8-594)

4. The variable bandIdx is set equal to bandTable[recPicture[xS_i][yS_j] >> bandShift].
5. The modified picture sample array saoPicture[xS_i][yS_j] is derived as follows:

$$saoPicture[xS_i][yS_j] = Clip3(0, (1 << bitDepth) - 1, recPicture[xS_i][yS_j] + SaoOffsetVal[cIdx][rx][ry][bandIdx]) \quad (8\text{-}595)$$

Table 8-14 – Specification of hPos and vPos according to the sample adaptive offset class

SaoEoClass[cIdx][rx][ry]	0	1	2	3
hPos[0]	−1	0	−1	1
hPos[1]	1	0	1	−1
vPos[0]	0	−1	−1	−1
vPos[1]	0	1	1	1

8.5.4 Adaptive loop filter process

8.5.4.1 General

Inputs of this process are the reconstructed picture sample arrays prior to adaptive loop filter $recPicture_L$, $recPicture_{Cb}$ and $recPicture_{Cr}$.

Outputs of this process are the modified reconstructed picture sample arrays after adaptive loop filter $alfPicture_L$, $alfPicture_{Cb}$ and $alfPicture_{Cr}$.

The sample values in the modified reconstructed picture sample arrays after adaptive loop filter $alfPicture_L$, $alfPicture_{Cb}$ and $alfPicture_{Cr}$ are initially set equal to the sample values in the reconstructed picture sample arrays prior to adaptive loop filter $recPicture_L$, $recPicture_{Cb}$ and $recPicture_{Cr}$, respectively.

When slice_alf_enabled_flag is equal to 1, for every coding tree unit with luma coding tree block location (rx, ry), where rx = 0..PicWidthInCtbs − 1 and ry = 0..PicHeightInCtbs − 1, the following applies:

– When alf_ctb_flag[0][rx][ry] is equal to 1, the coding tree block filtering process for luma samples as specified in clause 8.5.4.2 is invoked with $recPicture_L$, $alfPicture_L$, and the luma coding tree block location (xCtb, yCtb) set equal to (rx << CtbLog2SizeY, ry << CtbLog2SizeY) as inputs, and the output is the modified filtered picture $alfPicture_L$.

– When alf_ctb_flag[1][rx][ry] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.5.4.4 is invoked with recPicture set equal to $recPicture_{Cb}$, alfPicture set equal to $alfPicture_{Cb}$, and the chroma coding tree block location (xCtbC, yCtbC) set equal to (rx << (CtbLog2SizeY − 1), ry << (CtbLog2SizeY − 1)) as inputs, and the output is the modified filtered picture $alfPicture_{Cb}$.

– When alf_ctb_flag[2][rx][ry] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.5.4.4 is invoked with recPicture set equal to $recPicture_{Cr}$, alfPicture set equal to $alfPicture_{Cr}$, and the chroma coding tree block location (xCtbC, yCtbC) set equal to (rx << (CtbLog2SizeY − 1), ry << (CtbLog2SizeY − 1)) as inputs, and the output is the modified filtered picture $alfPicture_{Cr}$.

8.5.4.2 Coding tree block filtering process for luma samples

Inputs of this process are:

– a reconstructed luma picture sample array $recPicture_L$ prior to the adaptive loop filtering process,

– a filtered reconstructed luma picture sample array $alfPicture_L$,

– a luma location (xCtb, yCtb) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture.

Output of this process is the modified filtered reconstructed luma picture sample array $alfPicture_L$.

The derivation process for filter index clause 8.5.4.3 is invoked with the location (xCtb, yCtb) and the reconstructed luma picture sample array $recPicture_L$ as inputs, and filtIdx[x][y] and transposeIdx[x][y] with x, y = 0..CtbSizeY − 1 as outputs.

For the derivation of the filtered reconstructed luma samples $alfPicture_L$[x][y], each reconstructed luma sample inside the current luma coding tree block $recPicture_L$[x][y] is filtered as follows with x, y = 0..CtbSizeY − 1:

– The array of luma filter coefficients f[j] corresponding to the filter specified by filtIdx[x][y] is derived as follows with j = 0..12:

$$f[j] = AlfCoeff_L[filtIdx[x][y]][j] \quad (8\text{-}596)$$

– The luma filter coefficients filterCoeff are derived depending on transposeIdx[x][y] as follows:

  – If transposeIndex[x][y] = = 1,

$$filterCoeff[ ] = \{ f[9], f[4], f[10], f[8], f[1], f[5], f[11], f[7], f[3], f[0], f[2], f[6], f[12] \} \quad (8\text{-}597)$$

  – Otherwise, if transposeIndex[x][y] = = 2,

$$filterCoeff[ ] = \{ f[0], f[3], f[2], f[1], f[8], f[7], f[6], f[5], f[4], f[9], f[10], f[11], f[12] \} \quad (8\text{-}598)$$

  – Otherwise, if transposeIndex[x][y] = = 3,

$$filterCoeff[ ] = \{ f[9], f[8], f[10], f[4], f[3], f[7], f[11], f[5], f[1], f[0], f[2], f[6], f[12] \} \quad (8\text{-}599)$$

  – Otherwise,

$$filterCoeff[ ] = \{ f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10], f[11], f[12] \} \quad (8\text{-}600)$$

– The locations (h_x, v_y) for each of the corresponding luma samples (x, y) inside the given array recPicture of luma samples are derived as follows:

$$h_x = Clip3(0, pic\_width\_in\_luma\_samples − 1, xCtb + x) \quad (8\text{-}601)$$

$$v_y = Clip3(0, pic\_height\_in\_luma\_samples − 1, yCtb + y) \quad (8\text{-}602)$$

– The variable sum is derived as follows:

$$\begin{aligned}sum = {} & filterCoeff[\,0\,] * (recPicture_L[\,h_x, v_{y+3}\,] + recPicture_L[\,h_x, v_{y-3}\,]) + \\ & filterCoeff[\,1\,] * (recPicture_L[\,h_{x+1}, v_{y+2}\,] + recPicture_L[\,h_{x-1}, v_{y-2}\,]) + \\ & filterCoeff[\,2\,] * (recPicture_L[\,h_x, v_{y+2}\,] + recPicture_L[\,h_x, v_{y-2}\,]) + \\ & filterCoeff[\,3\,] * (recPicture_L[\,h_{x-1}, v_{y+2}\,] + recPicture_L[\,h_{x+1}, v_{y-2}\,]) + \\ & filterCoeff[\,4\,] * (recPicture_L[\,h_{x+2}, v_{y+1}\,] + recPicture_L[\,h_{x-2}, v_{y-1}\,]) + \\ & filterCoeff[\,5\,] * (recPicture_L[\,h_{x+1}, v_{y+1}\,] + recPicture_L[\,h_{x-1}, v_{y-1}\,]) + \\ & filterCoeff[\,6\,] * (recPicture_L[\,h_x, v_{y+1}\,] + recPicture_L[\,h_x, v_{y-1}\,]) + \\ & filterCoeff[\,7\,] * (recPicture_L[\,h_{x-1}, v_{y+1}\,] + recPicture_L[\,h_{x+1}, v_{y-1}\,]) + \\ & filterCoeff[\,8\,] * (recPicture_L[\,h_{x-2}, v_{y+1}\,] + recPicture_L[\,h_{x+2}, v_{y-1}\,]) + \\ & filterCoeff[\,9\,] * (recPicture_L[\,h_{x+3}, v_y\,] + recPicture_L[\,h_{x-3}, v_y\,]) + \\ & filterCoeff[\,10\,] * (recPicture_L[\,h_{x+2}, v_y\,] + recPicture_L[\,h_{x-2}, v_y\,]) + \\ & filterCoeff[\,11\,] * (recPicture_L[\,h_{x+1}, v_y\,] + recPicture_L[\,h_{x-1}, v_y\,]) + \\ & filterCoeff[\,12\,] * \; recPicture_L[\,h_x, v_y\,]\end{aligned} \tag{8-603}$$

$$sum = (sum + 64) >> 7 \tag{8-604}$$

– The modified filtered reconstructed luma picture sample $alfPicture_L[\,xCtb + x\,][\,yCtb + y\,]$ is derived as follows:

$$alfPicture_L[\,xCtb + x\,][\,yCtb + y\,] = Clip3(0, (1 << BitDepth_Y) - 1, sum) \tag{8-605}$$

8.5.4.3 Derivation process for ALF transpose and filter index for luma samples

Inputs of this process are:

– a luma location (xCtb, yCtb) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture,

– a reconstructed luma picture sample array $recPicture_L$ prior to the adaptive loop filtering process.

Outputs of this process are

– the classification filter index array $filtIdx[\,x\,][\,y\,]$ with x, y = 0..CtbSizeY − 1,

– the transpose index array $transposeIdx[\,x\,][\,y\,]$ with x, y = 0..CtbSizeY − 1.

The locations (h_x, v_y) for each of the corresponding luma samples (x, y) inside the given array recPicture of luma samples are derived as follows:

$$h_x = Clip3(0, pic\_width\_in\_luma\_samples - 1, x) \tag{8-606}$$

$$v_y = Clip3(0, pic\_height\_in\_luma\_samples - 1, y) \tag{8-607}$$

The classification filter index array filtIdx and the transpose index array transposeIdx are derived by the following ordered steps:

1. The variables $filtH[\,x\,][\,y\,]$, $filtV[\,x\,][\,y\,]$, $filtD0[\,x\,][\,y\,]$ and $filtD1[\,x\,][\,y\,]$ with x, y = − 2..CtbSizeY + 1 are derived as follows:

   – If both x and y are even numbers or both x and y are uneven numbers, the following applies:

$$filtH[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x-1}, v_{yCtb+y}\,] - recPicture[\,h_{xCtb+x+1}, v_{yCtb+y}\,]) \tag{8-608}$$

$$filtV[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x}, v_{yCtb+y-1}\,] - recPicture[\,h_{xCtb+x}, v_{yCtb+y+1}\,]) \tag{8-609}$$

$$filtD0[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x-1}, v_{yCtb+y-1}\,] - recPicture[\,h_{xCtb+x+1}, v_{yCtb+y+1}\,]) \tag{8-610}$$

$$filtD1[\,x\,][\,y\,] = Abs((recPicture[\,h_{xCtb+x}, v_{yCtb+y}\,] << 1) - recPicture[\,h_{xCtb+x+1}, v_{yCtb+y-1}\,] - recPicture[\,h_{xCtb+x-1}, v_{yCtb+y+1}\,]) \tag{8-611}$$

   – Otherwise, $filtH[\,x\,][\,y\,]$, $filtV[\,x\,][\,y\,]$, $filtD0[\,x\,][\,y\,]$ and $filtD1[\,x\,][\,y\,]$ are set equal to 0.

2. The variables varTempH1[x][y], varTempV1[x][y], varTempD01[x][y], varTempD11[x][y] and varTemp[x][y] with x, y = 0..(CtbSizeY − 1) >> 2 are derived as follows:

$$sumH[x][y] = \Sigma_i\Sigma_j\ filtH[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}612)$$

$$sumV[x][y] = \Sigma_i\Sigma_j\ filtV[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}613)$$

$$sumD0[x][y] = \Sigma_i\Sigma_j\ filtD0[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}614)$$

$$sumD1[x][y] = \Sigma_i\Sigma_j\ filtD1[(x << 2) + i][(y << 2) + j] \text{ with } i, j = -2..5 \quad (8\text{-}615)$$

$$sumOfHV[x][y] = sumH[x][y] + sumV[x][y] \quad (8\text{-}616)$$

3. The variables dir1[x][y], dir2[x][y] and dirS[x][y] with x, y = 0..CtbSizeY − 1 are derived as follows:
   - The variables hv1, hv0 and dirHV are derived as follows:
     - If sumV[x >> 2][y >> 2] is greater than sumH[x >> 2][y >> 2], the following applies:

$$hv1 = sumV[x >> 2][y >> 2] \quad (8\text{-}617)$$

$$hv0 = sumH[x >> 2][y >> 2] \quad 8\text{-}618)$$

$$dirHV = 1 \quad (8\text{-}619)$$

     - Otherwise, the following applies:

$$hv1 = sumH[x >> 2][y >> 2] \quad (8\text{-}620)$$

$$hv0 = sumV[x >> 2][y >> 2] \quad (8\text{-}621)$$

$$dirHV = 3 \quad (8\text{-}622)$$

   - The variables d1, d0 and dirD are derived as follows:
     - If sumD0[x >> 2][y >> 2] is greater than sumD1[x >> 2][y >> 2], the following applies:

$$d1 = sumD0[x >> 2][y >> 2] \quad (8\text{-}623)$$

$$d0 = sumD1[x >> 2][y >> 2] \quad (8\text{-}624)$$

$$dirD = 0 \quad (8\text{-}625)$$

     - Otherwise, the following applies:

$$d1 = sumD1[x >> 2][y >> 2] \quad (8\text{-}626)$$

$$d0 = sumD0[x >> 2][y >> 2] \quad (8\text{-}627)$$

$$dirD = 2 \quad (8\text{-}628)$$

   - The variables hvd1, hvd0, are derived as follows:

$$hvd1 = (d1 * hv0 > hv1 * d0)\ ?\ d1\ :\ hv1 \quad (8\text{-}629)$$

$$hvd0 = (d1 * hv0 > hv1 * d0)\ ?\ d0\ :\ hv0 \quad (8\text{-}630)$$

   - The variables dirS[x][y], dir1[x][y] and dir2[x][y] derived as follows:

$$dir1[x][y] = (d1 * hv0 > hv1 * d0)\ ?\ dirD\ :\ dirHV \quad (8\text{-}631)$$

$$dir2[x][y] = (d1 * hv0 > hv1 * d0)\ ?\ dirHV\ :\ dirD \quad (8\text{-}632)$$

$$dirS[x][y] = (hvd1 > 2 * hvd0)\ ?\ 1\ :\ (\ (hvd1 * 2 > 9 * hvd0)\ ?\ 2\ :\ 0\ ) \quad (8\text{-}633)$$

4. The variable avgVar[x][y] with x, y = 0..CtbSizeY − 1 is derived as follows:

$$varTab[\ ] = \{\ 0, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4\ \} \quad (8\text{-}634)$$

$$avgVar[x][y] = varTab[Clip3(0, 15, (sumOfHV[x >> 2][y >> 2] * 64) >> (3 + BitDepth_Y))] \quad (8\text{-}635)$$

5. The classification filter index array filtIdx[x][y] and the transpose index array transposeIdx[x][y] with x = y = 0..CtbSizeY − 1 are derived as follows:

   transposeTable[ ] = { 0, 1, 0, 2, 2, 3, 1, 3 }

   transposeIdx[x][y] = transposeTable[dir1[x][y] * 2 + (dir2[x][y] >> 1)]

   filtIdx[x][y] = avgVar[x][y]

   When dirS[x][y] is not equal 0, filtIdx[x][y] is modified as follows:

   filtIdx[x][y] += (((dir1[x][y] & 0x1) << 1) + dirS[x][y]) * 5 (8-636)

8.5.4.4 Coding tree block filtering process for chroma samples

Inputs of this process are:

– a reconstructed chroma picture sample array recPicture prior to the adaptive loop filtering process,

– a filtered reconstructed chroma picture sample array alfPicture,

– a chroma location (xCtbC, yCtbC) specifying the top-left sample of the current chroma coding tree block relative to the top left sample of the current picture.

Output of this process is the modified filtered reconstructed chroma picture sample array alfPicture.

The size of the current chroma coding tree block ctbSizeC is derived as follows:

ctbSizeC = CtbSizeY / SubWidthC (8-637)

For the derivation of the filtered reconstructed chroma samples alfPicture[x][y], each reconstructed chroma sample inside the current chroma coding tree block recPicture[x][y] is filtered as follows with x, y = 0..ctbSizeC − 1:

– The locations (h_x, v_y) for each of the corresponding chroma samples (x, y) inside the given array recPicture of chroma samples are derived as follows:

$$h_x = Clip3(0, pic\_width\_in\_luma\_samples / SubWidthC - 1, xCtbC + x) \quad (8\text{-}638)$$

$$v_y = Clip3(0, pic\_height\_in\_luma\_samples / SubHeightC - 1, yCtbC + y) \quad (8\text{-}639)$$

– The variable sum is derived as follows:

$$\begin{aligned} sum = {} & AlfCoeff_C[0] * (recPicture[h_x, v_{y+2}] + recPicture[h_x, v_{y-2}]) + \\ & AlfCoeff_C[1] * (recPicture[h_{x+1}, v_{y+1}] + recPicture[h_{x-1}, v_{y-1}]) + \\ & AlfCoeff_C[2] * (recPicture[h_x, v_{y+1}] + recPicture[h_x, v_{y-1}]) + \\ & AlfCoeff_C[3] * (recPicture[h_{x-1}, v_{y+1}] + recPicture[h_{x+1}, v_{y-1}]) + \\ & AlfCoeff_C[4] * (recPicture[h_{x+2}, v_y] + recPicture[h_{x-2}, v_y]) + \\ & AlfCoeff_C[5] * (recPicture[h_{x+1}, v_y] + recPicture[h_{x-1}, v_y]) + \\ & AlfCoeff_C[6] * recPicture[h_x, v_y] \end{aligned} \quad (8\text{-}640)$$

$$sum = (sum + 64) >> 7 \quad (8\text{-}641)$$

– The modified filtered reconstructed chroma picture sample alfPicture[xCtbC + x][yCtbC + y] is derived as follows:

$$alfPicture[xCtbC + x][yCtbC + y] = Clip3(0, (1 << BitDepth_C) - 1, sum) \quad (8\text{-}642)$$

9 Parsing process

9.1 General

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), se(v), uek(v) (see clause 9.2), tu(v) (see clause 9.3), tb(v) (see clause 9.4), or ae(v) (see clause 9.5).

9.2 Parsing process for k-th order Exp-Golomb codes

9.2.1 General

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), uek(v) or se(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v) or se(v) are Exp-Golomb-coded with order k equal to 0 and syntax elements coded as uek(v) are Exp-Golomb-coded with order k. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

```
leadingZeroBits = −1
for( b = 0; !b; leadingZeroBits++ )                                    (9-1)
    b = read_bits( 1 )
```

The variable codeNum is then assigned as follows:

$$codeNum = (2^{leadingZeroBits} - 1) * 2^{k} + read\_bits(leadingZeroBits + k) \quad (9\text{-}2)$$

where the value returned from read_bits(leadingZeroBits) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

Table 9-1 illustrates the structure of the 0-th order Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed as specified above for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 9-1. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as x_i in Table 9-1, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each x_i is equal to either 0 or 1.

Table 9-1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative)

Bit string form	Range of codeNum
1	0
0 1 x_0	1..2
0 0 1 x_1 x_0	3..6
0 0 0 1 x_2 x_1 x_0	7..14
0 0 0 0 1 x_3 x_2 x_1 x_0	15..30
0 0 0 0 0 1 x_4 x_3 x_2 x_1 x_0	31..62
...	...

Table 9-2 illustrates explicitly the assignment of bit strings to codeNum values.

Table 9-2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative)

Bit string	codeNum
1	0
0 1 0	1
0 1 1	2
0 0 1 0 0	3
0 0 1 0 1	4
0 0 1 1 0	5
0 0 1 1 1	6
0 0 0 1 0 0 0	7
0 0 0 1 0 0 1	8
0 0 0 1 0 1 0	9
...	...

Depending on the descriptor, the value of a syntax element is derived as follows:

– If the syntax element is coded as ue(v), the value of the syntax element is equal to codeNum.

– Otherwise (the syntax element is coded as se(v)), the value of the syntax element is derived by invoking the mapping process for signed Exp-Golomb codes as specified in clause 9.2.2 with codeNum as input.

9.2.2 Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in clause 9.2.

Output of this process is a value of a syntax element coded as se(v).

The syntax element is assigned to the codeNum by ordering the syntax element by its absolute value in increasing order and representing the positive value for a given absolute value with the lower codeNum. Table 9-3 provides the assignment rule.

Table 9-3 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v)

codeNum	syntax element value
0	0
1	1
2	−1
3	2
4	−2
5	3
6	−3
k	$(-1)^{k+1}$ Ceil(k ÷ 2)

9.3 Parsing process for truncated unary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tu(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tu(v) are truncated unary coded. The range of possible values for the syntax element is determined first. The range of this syntax element is 0 to maxVal inclusive, with maxVal being greater than or equal to 1. codeNum which is equal to the value of the syntax element is given by a process specified as follows:

```
codeNum = 0
keepGoing = 1
for(i = 0; i < maxVal && keepGoing; i++){
      keepGoing = read_bits( 1 )                                    (9-3)
      if( keepGoing )
            codeNum ++
}
```

9.4 Parsing process for truncated binary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tb(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tb(v) are truncated binary coded. The range of possible values for the syntax element is determined first. The range of this syntax element is 0 to maxValinclusive, with maxVal being greater than or equal to 1. synVal which is equal to the value of the syntax element is given by a process specified as follows:

```
thVal = 1 << 8
th = 8
while( thVal <= maxVal ) {
      th++
      thVal <<= 1
}
th− −
val = 1 << th                                                       (9-4)
b = n − val
synVal = read_bits( th )
if( synVal >= val − b ) {
      synVal <<= 1
      synVal += read_bits( 1 )
      synVal −= val − b
}
```

where the value returned from read_bits(th) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

9.5 CABAC parsing process for slice segment data

9.5.1 General

9.5.2 Initialization process

9.5.3 Binarization process

9.5.3.1 General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 9-4 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the truncated binary (TB) binarization process, the k-th order Exp-Golomb (EGk) binarization process and the fixed-length (FL) binarization process are given in clauses 9.5.3.3 through 9.5.3.6, respectively.

Table 9-4 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
coding_quadtree( )	qt_split_cu_flag[ ][ ]	FL	cMax = 1
multi_type_tree( )	mtt_split_cu_flag	FL	cMax = 1
	mtt_split_cu_vertical_flag	FL	cMax = 1
	mtt_split_cu_binary_flag	FL	cMax = 1
coding_tree_unit( )	alf_ctb_flag[ ][ ][ ]	FL	cMax = 1
coding_unit( )	cu_skip_flag[ ][ ]	FL	cMax = 1
	pred_mode_flag	FL	cMax = 1
	intra_luma_ref_idx[ ][ ]	TR	cMax = 2, cRiceParam = 0
	intra_luma_mpm_flag[ ][ ]	FL	cMax = 1
	intra_luma_mpm_idx[ ][ ]	TR	cMax = 5, cRiceParam = 0
	intra_luma_mpm_remainder[ ][ ]	TB	cMax = 60
	intra_chroma_pred_mode[ ][ ]	9.5.3.7	-
	merge_affine_flag[ ][ ]	FL	cMax = 1
	merge_flag[ ][ ]	FL	cMax = 1
	merge_idx[ ][ ]	TR	cMax = MaxNumMergeCand − 1, cRiceParam = 0
	inter_pred_idc[x0][y0]	9.5.3.8	-
	inter_affine_flag[ ][ ]	FL	cMax = 1
	cu_affine_type_flag[ ][ ]	FL	cMax = 1
	ref_idx_l0[ ][ ]	TR	cMax = num_ref_idx_l0_active_minus1, cRiceParam = 0
	mvp_l0_flag[ ][ ]	FL	cMax = 1
	ref_idx_l1[ ][ ]	TR	cMax = num_ref_idx_l1_active_minus1, cRiceParam = 0
	mvp_l1_flag[ ][ ]	FL	cMax = 1
	amvr_mode[ ][ ]	TR	cMax = 2, cRiceParam = 0
	cu_cbf	FL	cMax = 1
mvd_coding( )	abs_mvd_greater0_flag[ ]	FL	cMax = 1
	abs_mvd_greater1_flag[ ]	FL	cMax = 1
	abs_mvd_minus2[ ]	EG1	-
	mvd_sign_flag[ ]	FL	cMax = 1

Table 9-4 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
transform_unit( )	tu_cbf_luma[ ][ ][ ]	FL	cMax = 1
	tu_cbf_cb[ ][ ][ ]	FL	cMax = 1
	tu_cbf_cr[ ][ ][ ]	FL	cMax = 1
	cu_qp_delta_abs	9.5.3.9	-
	cu_qp_delta_sign_flag	FL	cMax = 1
	tu_mts_flag[ ][ ]	FL	cMax = 1
residual_coding( )	transform_skip_flag[ ][ ][ ]	FL	cMax = 1
	last_sig_coeff_x_prefix	TR	cMax = (log2TrafoSize << 1) − 1, cRiceParam = 0
	last_sig_coeff_y_prefix	TR	cMax = (log2TrafoSize << 1) − 1, cRiceParam = 0
	last_sig_coeff_x_suffix	FL	cMax = (1 << ((last_sig_coeff_x_prefix >> 1) − 1) − 1)
	last_sig_coeff_y_suffix	FL	cMax = (1 << ((last_sig_coeff_y_prefix >> 1) − 1) − 1)
	coded_sub_block_flag[ ][ ]	FL	cMax = 1
	sig_coeff_flag[ ][ ]	FL	cMax = 1
	par_level_flag[ ]	FL	cMax = 1
	rem_abs_gt1_flag[ ]	FL	cMax = 1
	rem_abs_gt2_flag[ ]	FL	cMax = 1
	abs_remainder[ ]	9.5.3.10	cIdx, x0, y0, xC, yC, log2TbWidth, log2TbHeight
	coeff_sign_flag[ ]	FL	cMax = 1
	mts_idx[ ][ ][ ]	FL	cMax = 3

9.5.3.2 Rice parameter derivation process

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the Rice parameter cRiceParam.

Given the syntax elements sig_coeff_flag[x][y] and the array AbsLevel[x][C] for the transform block with component index cIdx and the top-left luma location (x0, y0), the variable locSumAbs is derived as specified by the following pseudo code:

```
locSumAbs = 0
if( xC < (1 << log2TbWidth) − 1 ) {
    locSumAbs += AbsLevel[ xC + 1 ][ yC ] − sig_coeff_flag[ xC + 1 ][ yC ]
    if( xC < (1 << log2TbWidth) − 2 )
        locSumAbs += AbsLevel[ xC + 2 ][ yC ] − sig_coeff_flag[ xC + 2 ][ yC ]
    if( yC < (1 << log2TbHeight) − 1 )
        locSumAbs += AbsLevel[ xC + 1 ][ yC + 1 ] − sig_coeff_flag[ xC + 1 ][ yC + 1 ]          (9-5)
}
if( yC < (1 << log2TbHeight) − 1 ) {
    locSumAbs += AbsLevel[ xC ][ yC + 1 ] − sig_coeff_flag[ xC ][ yC + 1 ]
```

```
        if( yC < (1 << log2TbHeight) − 2 )
            locSumAbs += AbsLevelPass1 [ xC ][ yC + 2 ] − sig_coeff_flag[ xC ][ yC + 2 ]
    }
```

The Rice parameter cRiceParam is derived as follows:

– If locSumAbs is less than 12, cRiceParam is set equal to 0;

– Otherwise, if locSumAbs is less than 25, cRiceParam is set equal to 1;

– Otherwise (locSumAbs is greater than or equal to 25), cRiceParam is set equal to 2.

9.5.3.3 Truncated Rice binarization process

Input to this process is a request for a truncated Rice (TR) binarization, cMax and cRiceParam.

Output of this process is the TR binarization associating each value symbolVal with a corresponding bin string.

A TR bin string is a concatenation of a prefix bin string and, when present, a suffix bin string.

For the derivation of the prefix bin string, the following applies:

– The prefix value of symbolVal, prefixVal, is derived as follows:

$$prefixVal = symbolVal \gg cRiceParam \tag{9-6}$$

– The prefix of the TR bin string is specified as follows:

  – If prefixVal is less than cMax >> cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0. Table 9-5 illustrates the bin strings of this unary binarization for prefixVal.

  – Otherwise, the bin string is a bit string of length cMax >> cRiceParam with all bins being equal to 1.

Table 9-5 – Bin string of the unary binarization (informative)

prefixVal	Bin string					
0	0					
1	1	0				
2	1	1	0			
3	1	1	1	0		
4	1	1	1	1	0	
5	1	1	1	1	1	0
...						
binIdx	0	1	2	3	4	5

When cMax is greater than symbolVal and cRiceParam is greater than 0, the suffix of the TR bin string is present and it is derived as follows:

– The suffix value suffixVal is derived as follows:

$$suffixVal = symbolVal − ((prefixVal) \ll cRiceParam) \tag{9-7}$$

– The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified in clause 9.5.3.6 for suffixVal with a cMax value equal to (1 << cRiceParam) − 1.

NOTE – For the input parameter cRiceParam = 0, the TR binarization is exactly a truncated unary binarization and it is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

9.5.3.4 Truncated Binary (TB) binarization process

Input to this process is a request for a TB binarization for a syntax element with value synVal and cMax. Output of this process is the TB binarization of the syntax element.The bin string of the TB binarization process of a syntax element synVal is specified as follows:

```
n = cMax + 1
k = Floor( Log2( n ) )                                    (9-8)
u = ( 1 << ( k + 1) ) − n
```

– If synVal is less than u, the TB bin string is derived by invoking the FL binarization process specified in clause 9.5.3.6 for synVal with a cMax value equal to (1 << k) − 1.

– Otherwise (synVal is greater than or equal to u), the TB bin string is derived by invoking the FL binarization process specified in clause 9.5.3.6 for (synVal + u) with a cMax value equal to (1 << (k + 1)) − 1.

9.5.3.5 k-th order Exp-Golomb binarization process

Inputs to this process is a request for a k-th order Exp-Golomb (EGk) binarization.

Output of this process is the EGk binarization associating each value symbolVal with a corresponding bin string.

The bin string of the EGk binarization process for each value symbolVal is specified as follows, where each call of the function put(X), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

```
absV = Abs( symbolVal )
stopLoop = 0
do
    if( absV >= ( 1 << k ) ) {
        put( 1 )
        absV = absV − ( 1 << k )
        k++
    } else {
        put( 0 )                                          (9-9)
        while( k− − )
            put( ( absV >> k ) & 1 )
        stopLoop = 1
    }
while( !stopLoop )
```

NOTE – The specification for the k-th order Exp-Golomb (EGk) code uses 1's and 0's in reverse meaning for the unary part of the Exp-Golomb code of k-th order as specified in clause 9.2.

9.5.3.6 Fixed-length binarization process

Inputs to this process are a request for a fixed-length (FL) binarization and cMax.

Output of this process is the FL binarization associating each value symbolVal with a corresponding bin string.

FL binarization is constructed by using the fixedLength-bit unsigned integer bin string of the symbol value symbolVal, where fixedLength = Ceil(Log2(cMax + 1)). The indexing of bins for the FL binarization is such that the binIdx = 0 relates to the most significant bit with increasing values of binIdx towards the least significant bit.

9.5.3.7 Binarization process for intra_chroma_pred_mode

Input to this process is a request for a binarization for the syntax element intra_chroma_pred_mode.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element intra_chroma_pred_mode is specified in Table 9-6 and Table 9-7.

Table 9-6 – Binarization for intra_chroma_pred_mode when sps_cclm_enabled_flag is equal to 0

Value of intra_chroma_pred_mode	Bin string
4	0
0	100
1	101
2	110
3	111

Table 9-7 – Binarization for intra_chroma_pred_mode when sps_cclm_enabled_flag is equal to 1

Value of intra_chroma_pred_mode	Bin string
7	0
4	10
5	1110
6	1111
0	11000
1	11001
2	11010
3	11011

9.5.3.8 Binarization process for inter_pred_idc

Input to this process is a request for a binarization for the syntax element inter_pred_idc.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element inter_pred_idc is specified in Table 9-8.

Table 9-8 – Binarization for inter_pred_idc

Value of inter_pred_idc	Name of inter_pred_idc	Bin string
0	PRED_L0	00
1	PRED_L1	01
2	PRED_BI	1

9.5.3.9 Binarization process for cu_qp_delta_abs

Input to this process is a request for a binarization for the syntax element cu_qp_delta_abs.

Output of this process is the binarization of the syntax element.

The binarization of the syntax element cu_qp_delta_abs is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

– The prefix value of cu_qp_delta_abs, prefixVal, is derived as follows:

$$prefixVal = Min(cu\_qp\_delta\_abs, 5) \quad (9\text{-}10)$$

– The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with cMax = 5 and cRiceParam = 0.

When prefixVal is greater than 4, the suffix bin string is present and it is derived as follows:

– The suffix value of cu_qp_delta_abs, suffixVal, is derived as follows:

$$suffixVal = cu\_qp\_delta\_abs - 5 \quad (9\text{-}11)$$

– The suffix bin string is specified by invoking the k-th order EGk binarization process as specified in clause 9.5.3.5 for suffixVal with the Exp-Golomb order k set equal to 0.

9.5.3.10 Binarization process for abs_remainder[ ]

Input to this process is a request for a binarization for the syntax element abs_remainder[n], the colour component cIdx, the luma location (x0, y0) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the picture), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the binarization of the syntax element.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process as specified in clause 9.5.3.2 with the colour component index cIdx, the luma location (x0, y0), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as inputs.

The variable cMax is derived from cRiceParam as:

$$cMax = (cRiceParam == 1 \ ? \ 6 \ : \ 7) << cRiceParam \quad (9\text{-}12)$$

The binarization of the syntax element abs_remainder[n] is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

– The prefix value of abs_remainder[n], prefixVal, is derived as follows:

$$prefixVal = Min(cMax, abs\_remainder[n]) \quad (9\text{-}13)$$

– The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with the variables cMax and cRiceParam as inputs.

When the prefix bin string is equal to the bit string of length 4 with all bits equal to 1, the suffix bin string is present and it is derived as follows:

– The suffix value of abs_remainder[n], suffixVal, is derived as follows:

$$suffixVal = abs\_remainder[n] - cMax \quad (9\text{-}14)$$

– The suffix bin string is specified by invoking the k-th order EGk binarization process as specified in clause 9.5.3.5 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1.

9.5.4 Decoding process flow

9.5.4.1 General

9.5.4.2 Derivation process for ctxTable, ctxIdx and bypassFlag

9.5.4.2.1 General

9.5.4.2.2 Derivation process of ctxInc for the syntax element tu_cbf_cr[ ][ ]

Inputs to this process is the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture.

Outputs of this process is the variable ctxInc.

The variable ctxInc is set equal to tu_cbf_cb[x0][y0].

9.5.4.2.3 Derivation process for the variables locNumSig, locSumAbsPass1

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Outputs of this process are the variables locNumSig and locSumAbsPass1.

Given the syntax elements sig_coeff_flag[x][y] and the array AbsLevelPass1[x][C] for the transform block with component index cIdx and the top-left luma location (x0, y0), the variables locNumSig and locSumAbsPass1 are derived as specified by the following pseudo code:

```
locNumSig      = 0
locSumAbsPass1 = 0
if( xC < (1 << log2TbWidth) − 1 ) {
    locNumSig      += sig_coeff_flag[ xC + 1 ][ yC ]
    locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC ]
    if( xC < (1 << log2TbWidth) − 2 ) {
        locNumSig      += sig_coeff_flag[ xC + 2 ][ yC ]
        locSumAbsPass1 += AbsLevelPass1[ xC + 2 ][ yC ]
    }
    if( yC < (1 << log2TbHeight) − 1 ) {
        locNumSig      += sig_coeff_flag[ xC + 1 ][ yC + 1 ]                    (9-15)
        locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC + 1 ]
    }
}
if( yC < (1 << log2TbHeight) − 1 ) {
    locNumSig      += sig_coeff_flag[ xC ][ yC + 1 ]
    locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 1 ]
    if( yC < (1 << log2TbHeight) − 2 ) {
        locNumSig      += sig_coeff_flag[ xC ][ yC + 2 ]
        locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 2 ]
    }
}
```

9.5.4.2.4 Derivation process of ctxInc for the syntax element sig_coeff_flag

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.3 with colour component index cIdx, the luma location (x0, y0), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

– If cIdx is equal to 0, ctxInc is derived as follows:

$$ctxInc = 18 * Max(0, QState − 1) + Min(locSumAbsPass1, 5) + (d < 2 \ ? \ 12 \ : \ (d < 5 \ ? \ 6 \ : \ 0)) \quad (9\text{-}16)$$

– Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

$$ctxInc = 54 + 12 * Max(0, QState − 1) + Min(locSumAbsPass1, 5) + (d < 2 \ ? \ 6 \ : \ 0) \quad (9\text{-}17)$$

9.5.4.2.5 Derivation process of ctxInc for the syntax elements par_level_flag, rem_abs_gt1_flag, and rem_abs_level_gt2_flag

Inputs to this process are the colour component index cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location

(xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variablea locNumSig and locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.3 with colour component index cIdx, the luma location (x0, y0), the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable ctxOffset is set equal to Min(locSumAbsPass1 − locNumSig, 4).

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

– If xC is equal to LastSignificantCoeffX and yC is equal to LastSignificantCoeffY, ctxInc is derived as follows:

$$ctxInc = (cIdx == 0 \ ? \ 0 \ : \ 21) \quad (9\text{-}18)$$

– Otherwise, if cIdx is equal to 0, ctxInc is derived as follows:

$$ctxInc = 1 + ctxOffset + (d == 0 \ ? \ 15 \ : \ (d < 3 \ ? \ 10 \ : \ (d < 10 \ ? \ 5 \ : \ 0))) \quad (9\text{-}19)$$

– Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

$$ctxInc = 22 + ctxOffset + (d == 0 \ ? \ 5 \ : \ 0) \quad (9\text{-}20)$$