(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 May 2020 (14.05.2020)

WIPO | PCT

(10) International Publication Number
WO 2020/095984 A1

(51) International Patent Classification:
H04N 19/70 (2014.01)

(21) International Application Number:
PCT/JP2019/043637

(22) International Filing Date:
07 November 2019 (07.11.2019)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

62/757,054	07 November 2018 (07.11.2018)	US
62/757,669	08 November 2018 (08.11.2018)	US
62/790,763	10 January 2019 (10.01.2019)	US
62/803,239	08 February 2019 (08.02.2019)	US
62/803,734	11 February 2019 (11.02.2019)	US
62/815,980	08 March 2019 (08.03.2019)	US

(71) Applicant: **SHARP KABUSHIKI KAISHA** [JP/JP]; 1, Takumi-cho, Sakai-ku, Sakai City, Osaka, 5908522 (JP).

(72) Inventor: **BOSSEN, Frank**.

(74) Agent: **HARAKENZO WORLD PATENT & TRADEMARK**; Daiwa Minamimorimachi Building, 2-6, Tenjinbashi 2-chome Kita, Kita-ku, Osaka-shi, Osaka, 5300041 (JP).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

(54) Title: SYSTEMS AND METHODS FOR PERFORMING BINARY ARITHMETIC CODING IN VIDEO CODING

(57) Abstract: According to an aspect of an invention, a method comprises determining an initial probability state of bin of a syntax element and entropy encoding the bin of the syntax element using the determined initial probability state.

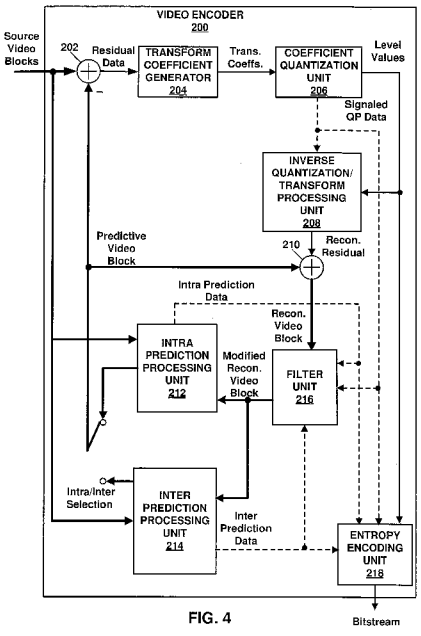

FIG. 4

WO 2020/095984 A1

[Continued on next page]

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

— *with international search report (Art. 21(3))*

Description

Title of Invention: SYSTEMS AND METHODS FOR PERFORMING BINARY ARITHMETIC CODING IN VIDEO CODING

Technical Field

[0001] This disclosure relates to video coding and more particularly to techniques for performing binary arithmetic coding.

Background Art

[0002] Digital video capabilities can be incorporated into a wide range of devices, including digital televisions, laptop or desktop computers, tablet computers, digital recording devices, digital media players, video gaming devices, cellular telephones, including so-called smartphones, medical imaging devices, and the like. Digital video may be coded according to a video coding standard. Video coding standards define the format of a compliant bitstream. A compliant bitstream is data structure that may be received and decoded by a video decoding device to generate reconstructed video data. Video coding standards may incorporate video compression techniques. Examples of video coding standards include ISO/IEC MPEG-4 Visual and ITU-T H.264 (also known as ISO/IEC MPEG-4 AVC) and High-Efficiency Video Coding (HEVC). HEVC is described in High Efficiency Video Coding (HEVC), Rec. ITU-T H.265, December 2016, which is incorporated by reference, and referred to herein as ITU-T H.265. Extensions and improvements for ITU-T H.265 are currently being considered for the development of next generation video coding standards. For example, the ITU-T Video Coding Experts Group (VCEG) and ISO/IEC (Moving Picture Experts Group (MPEG) (collectively referred to as the Joint Video Exploration Team (JVET)) are working to standardized video coding technology with a compression capability that significantly exceeds that of the current HEVC standard. The Joint Exploration Model 7 (JEM 7), Algorithm Description of Joint Exploration Test Model 7 (JEM 7), ISO/IEC JTC1/SC29/WG11 Document: JVET-G1001, July 2017, Torino, IT, which is incorporated by reference herein, describes the coding features that were under coordinated test model study by the JVET as potentially enhancing video coding technology beyond the capabilities of ITU-T H.265. It should be noted that the coding features of JEM 7 are implemented in JEM reference software. As used herein, the term JEM may collectively refer to algorithms included in JEM 7 and implementations of JEM reference software. Further, in response to a "Joint Call for Proposals on Video Compression with Capabilities beyond HEVC," jointly issued by VCEG and MPEG, multiple descriptions of video coding tools were proposed by various groups at the 10th

Meeting of ISO/IEC JTC1/SC29/WG11 16-20 April 2018, San Diego, CA. From the multiple descriptions of video coding tools, a resulting initial draft text of a video coding specification is described in "Versatile Video Coding (Draft 1)," 10th Meeting of ISO/IEC JTC1/SC29/WG11 16-20 April 2018, San Diego, CA, document JVET-J1001-v2, which is incorporated by reference herein, and referred to as JVET-J1001. This current development of the a next generation video coding standard by the VCEG and MPEG is referred to as the Versatile Video Coding (VVC) project. "Versatile Video Coding (Draft 4)," 13th Meeting of ISO/IEC JTC1/SC29/WG11 9-18 January 2019, Marrakech, MA, document JVET-M1001-v1, which is incorporated by reference herein, and referred to as JVET-M1001, represents the current iteration of the draft text of a video coding specification corresponding to the VVC project.

[0003] Video compression techniques enable data requirements for storing and transmitting video data to be reduced. Video compression techniques may reduce data requirements by exploiting the inherent redundancies in a video sequence. Video compression techniques may sub-divide a video sequence into successively smaller portions (i.e., groups of pictures within a video sequence, a picture within a group of pictures, regions within a picture, sub-regions within regions, etc.). Intra prediction coding techniques (e.g., spatial prediction techniques within a picture) and inter prediction techniques (i.e., inter-picture techniques (temporal)) may be used to generate difference values between a unit of video data to be coded and a reference unit of video data. The difference values may be referred to as residual data. Residual data may be coded as quantized transform coefficients. Syntax elements may relate residual data and a reference coding unit (e.g., intra-prediction mode indices, and motion information). Residual data and syntax elements may be entropy coded. Entropy encoded residual data and syntax elements may be included in data structures forming a compliant bitstream.

Summary of Invention

[0004] In one example, a method comprises determining an initial probability state of bin of a syntax element and entropy encoding the bin of the syntax element using the determined initial probability state.

[0005] In one example, a device comprises one or more processors configured to determine an initial probability state of a bin of a syntax element and entropy decode the bin of the syntax element using the determined initial probability state.

Brief Description of Drawings

[0006] [fig.1]FIG. 1 is a conceptual diagram illustrating an example of a group of pictures coded according to a quad tree multi-tree partitioning in accordance with one or more techniques of this disclosure.

[fig.2A]FIG. 2A is conceptual diagram illustrating example of coding a block of video data in accordance with one or more techniques of this disclosure.

[fig.2B]FIG. 2B is conceptual diagram illustrating example of coding a block of video data in accordance with one or more techniques of this disclosure.

[fig.3]FIG. 3 is a block diagram illustrating an example of a system that may be configured to encode and decode video data according to one or more techniques of this disclosure.

[fig.4]FIG. 4 is a block diagram illustrating an example of a video encoder that may be configured to encode video data according to one or more techniques of this disclosure.

[fig.5]FIG. 5 is a block diagram illustrating an example of a entropy encoder that may be configured to encode video data according to one or more techniques of this disclosure.

[fig.6]FIG. 6 is a block diagram illustrating an example of a video decoder that may be configured to decode video data according to one or more techniques of this disclosure.

[fig.7]FIG. 7 is a block diagram illustrating an example of a entropy decoder that may be configured to encode video data according to one or more techniques of this disclosure.

[fig.8]FIG. 8 is a conceptual diagram illustrating an example of a group of pictures coded according to a quad tree binary tree partitioning in accordance with one or more techniques of this disclosure.

Description of Embodiments

[0007] In general, this disclosure describes various techniques for coding video data. In particular, this disclosure describes techniques for performing binary arithmetic coding. It should be noted that although techniques of this disclosure are described with respect to ITU-T H.264, ITU-T H.265, JEM, and JVET-M1001 the techniques of this disclosure are generally applicable to video coding. For example, the coding techniques described herein may be incorporated into video coding systems, (including video coding systems based on future video coding standards) including video block structures, intra prediction techniques, inter prediction techniques, transform techniques, filtering techniques, and/or entropy coding techniques other than those included in ITU-T H.265, JEM, and JVET-M1001. Thus, reference to ITU-T H.264, ITU-T H.265, JEM, and/or JVET-M1001 is for descriptive purposes and should not be construed to limit the scope of the techniques described herein. Further, it should be noted that incorporation by reference of documents herein is for descriptive purposes and should not be construed to limit or create ambiguity with respect to terms used herein. For example, in the case where an incorporated reference provides a different definition of a term than another incorporated reference and/or as the term is used

herein, the term should be interpreted in a manner that broadly includes each respective definition and/or in a manner that includes each of the particular definitions in the alternative.

[0008] In one example, a device comprises one or more processors configured to determine an initial probability state of a bin of a syntax element and entropy encode the bin of the syntax element using the determined initial probability state.

[0009] In one example, a non-transitory computer-readable storage medium comprises instructions stored thereon that, when executed, cause one or more processors of a device to determine an initial probability state of a bin of a syntax element and entropy encode the bin of the syntax element using the determined initial probability state.

[0010] In one example, an apparatus comprises means for determining an initial probability state of a bin of a syntax element and means for entropy encoding the bin of the syntax element using the determined initial probability state.

[0011] In one example, a method comprises determining an initial probability state of a bin of a syntax element and entropy decoding the bin of the syntax element using the determined initial probability state.

[0012] In one example, a non-transitory computer-readable storage medium comprises instructions stored thereon that, when executed, cause one or more processors of a device to determine an initial probability state of a bin of a syntax element and entropy decode the bin of the syntax element using the determined initial probability state.

[0013] In one example, an apparatus comprises means for determining an initial probability state of a bin of a syntax element and means for entropy decoding the bin of the syntax element using the determined initial probability state.

[0014] The details of one or more examples are set forth in the accompanying drawings and the description below. Other features, objects, and advantages will be apparent from the description and drawings, and from the claims.

[0015] Video content typically includes video sequences comprised of a series of frames (or pictures). A series of frames may also be referred to as a group of pictures (GOP). Each video frame or picture may include a plurality of slices or tiles, where a slice or tile includes a plurality of video blocks. As used herein, the term video block may generally refer to an area of a picture or may more specifically refer to the largest array of sample values that may be predictively coded, sub-divisions thereof, and/or corresponding structures. Further, the term current video block may refer to an area of a picture being encoded or decoded. A video block may be defined as an array of sample values that may be predictively coded. It should be noted that in some cases pixel values may be described as including sample values for respective components of video data, which may also be referred to as color components, (e.g., luma (Y) and chroma (Cb and Cr) components or red, green, and blue components). It should be

noted that in some cases, the terms pixel value and sample value are used interchangeably. Video blocks may be ordered within a picture according to a scan pattern (e.g., a raster scan). A video encoder may perform predictive encoding on video blocks and sub-divisions thereof. Video blocks and sub-divisions thereof may be referred to as nodes.

[0016] A video sampling format, which may also be referred to as a chroma format, may define the number of chroma samples included in a CU with respect to the number of luma samples included in a CU. For example, for the 4:2:0 sampling format, the sampling rate for the luma component is twice that of the chroma components for both the horizontal and vertical directions. As a result, for a CU formatted according to the 4:2:0 format, the width and height of an array of samples for the luma component are twice that of each array of samples for the chroma components. As described above, a CU is typically defined according to the number of horizontal and vertical luma samples. For a CU formatted according to the 4:2:2 format, the width of an array of samples for the luma component is twice that of the width of an array of samples for each chroma component, but the height of the array of samples for the luma component is equal to the height of an array of samples for each chroma component. Further, for a CU formatted according to the 4:4:4 format, an array of samples for the luma component has the same width and height as an array of samples for each chroma component.

[0017] ITU-T H.264 specifies a macroblock including 16x16 luma samples. That is, in ITU-T H.264, a picture is segmented into macroblocks. ITU-T H.265 specifies an analogous Coding Tree Unit (CTU) structure (which may be referred to as a largest coding unit (LCU)). In ITU-T H.265, pictures are segmented into CTUs. In ITU-T H.265, for a picture, a CTU size may be set as including 16x16, 32x32, or 64x64 luma samples. In ITU-T H.265, a CTU is composed of respective Coding Tree Blocks (CTB) for each component of video data (e.g., luma (Y) and chroma (Cb and Cr). It should be noted that video having one luma component and the two corresponding chroma components may be described as having two channels, i.e., a luma channel and a chroma channel. Further, in ITU-T H.265, a CTU may be partitioned according to a quadtree (QT) partitioning structure, which results in the CTBs of the CTU being partitioned into Coding Blocks (CB). That is, in ITU-T H.265, a CTU may be partitioned into quadtree leaf nodes. According to ITU-T H.265, one luma CB together with two corresponding chroma CBs and associated syntax elements are referred to as a coding unit (CU). In ITU-T H.265, a minimum allowed size of a CB may be signaled. In ITU-T H.265, the smallest minimum allowed size of a luma CB is 8x8 luma samples. In ITU-T H.265, the decision to code a picture area using intra prediction or inter prediction is made at the CU level.

[0018] In ITU-T H.265, a CU is associated with a prediction unit (PU) structure having its root at the CU. In ITU-T H.265, PU structures allow luma and chroma CBs to be split for purposes of generating corresponding reference samples. That is, in ITU-T H.265, luma and chroma CBs may be split into respective luma and chroma prediction blocks (PBs), where a PB includes a block of sample values for which the same prediction is applied. In ITU-T H.265, a CB may be partitioned into 1, 2, or 4 PBs. ITU-T H.265 supports PB sizes from 64x64 samples down to 4x4 samples. In ITU-T H.265, square PBs are supported for intra prediction, where a CB may form the PB or the CB may be split into four square PBs (i.e., intra prediction PB types include MxM or M/2xM/2, where M is the height and width of the square CB). In ITU-T H.265, in addition to the square PBs, rectangular PBs are supported for inter prediction, where a CB may by halved vertically or horizontally to form PBs (i.e., inter prediction PB types include MxM, M/2xM/2, M/2xM, or MxM/2). Further, it should be noted that in ITU-T H.265, for inter prediction, four asymmetric PB partitions are supported, where the CB is partitioned into two PBs at one quarter of the height (at the top or the bottom) or width (at the left or the right) of the CB (i.e., asymmetric partitions include M/4xM left, M/4xM right, MxM/4 top, and MxM/4 bottom). Intra prediction data (e.g., intra prediction mode syntax elements) or inter prediction data (e.g., motion data syntax elements) corresponding to a PB is used to produce reference and/or predicted sample values for the PB.

[0019] JEM specifies a CTU having a maximum size of 256x256 luma samples. JEM specifies a quadtree plus binary tree (QTBT) block structure. In JEM, the QTBT structure enables quadtree leaf nodes to be further partitioned by a binary tree (BT) structure. That is, in JEM, the binary tree structure enables quadtree leaf nodes to be recursively divided vertically or horizontally. FIG. 8 illustrates an example of a CTU (e.g., a CTU having a size of 256x256 luma samples) being partitioned into quadtree leaf nodes and quadtree leaf nodes being further partitioned according to a binary tree. That is, in FIG. 8 dashed lines indicate additional binary tree partitions in a quadtree. Thus, the binary tree structure in JEM enables square and rectangular leaf nodes, where each leaf node includes a CB. As illustrated in FIG. 8, a picture included in a GOP may include slices, where each slice includes a sequence of CTUs and each CTU may be partitioned according to a QTBT structure. FIG. 8 illustrates an example of QTBT partitioning for one CTU included in a slice. In JEM, CBs are used for prediction without any further partitioning. That is, in JEM, a CB may be a block of sample values on which the same prediction is applied. Thus, a JEM QTBT leaf node may be analogous a PB in ITU-T H.265.

[0020] As described above, each video frame or picture may divided into one or more regions. For example, according to ITU-T H.265, each video frame or picture may be

partitioned to include one or more slices and further partitioned to include one or more tiles, where each slice includes a sequence of CTUs (e.g., in raster scan order) and where a tile is a sequence of CTUs corresponding to a rectangular area of a picture. It should be noted that a slice, in ITU-T H.265, is a sequence of one or more slice segments starting with an independent slice segment and containing all subsequent dependent slice segments (if any) that precede the next independent slice segment (if any) within the same access unit. A slice segment, like a slice, is a sequence of CTUs. Thus, in some cases, the terms slice and slice segment may be used interchangeably to indicate a sequence of CTUs. Further, it should be noted that in ITU-T H.265, a tile may consist of CTUs contained in more than one slice and a slice may consist of CTUs contained in more than one tile. However, ITU-T H.265 provides that one or both of the following conditions shall be fulfilled: (1) All CTUs in a slice belong to the same tile; and (2) All CTUs in a tile belong to the same slice. With respect to JVET-M1001, slices are required to consist of an integer number of complete tiles instead of only being required to consist of an integer number of complete CTUs. As such, a slice including a set of CTUs which do not form a rectangular region of a picture may or may not be supported in some video coding techniques. Further, a slice that is required to consist of an integer number of complete tiles is referred to as a tile group. The techniques described herein may applicable to slices, tiles, and/or tile groups. FIG. 1 is a conceptual diagram illustrating an example of a group of pictures including tile groups. In the example illustrated in FIG. 1, Pic_3 is illustrated as including two tile groups (i.e., Tile $Group_1$ and Tile $Group_2$). It should be noted that in some cases, Tile $Group_1$ and Tile $Group_2$ may be classified as slices and/or tiles.

[0021] JEM specifies a CTU having a maximum size of 256x256 luma samples. JEM specifies a quadtree plus binary tree (QTBT) block structure. In JEM, the QTBT structure enables quadtree leaf nodes to be further partitioned by a binary tree (BT) structure. That is, in JEM, the binary tree structure enables quadtree leaf nodes to be recursively divided vertically or horizontally. In JVET-M1001, CTUs are partitioned according a quadtree plus multi-type tree (QTMT) structure. The QTMT in JVET-M1001 is similar to the QTBT in JEM. However, in JVET-M1001, in addition to indicating binary splits, the multi-type tree may indicate so-called ternary (or triple tree (TT)) splits. A ternary split divides a block vertically or horizontally into three blocks. In the case of a vertical TT split, a block is divided at one quarter of its width from the left edge and at one quarter its width from the right edge and in the case of a horizontal TT split a block is at one quarter of its height from the top edge and at one quarter of its height from the bottom edge. Referring again to FIG. 1, FIG. 1 illustrates an example of a CTU being partitioned into quadtree leaf nodes and quadtree leaf nodes being further partitioned according to a BT split or a TT split. That is, in FIG. 1 dashed

lines indicate additional binary and ternary splits in a quadtree.

[0022] For intra prediction coding, an intra prediction mode may specify the location of reference samples within a picture. In ITU-T H.265, defined possible intra prediction modes include a planar (i.e., surface fitting) prediction mode (predMode: 0), a DC (i.e., flat overall averaging) prediction mode (predMode: 1), and 33 angular prediction modes (predMode: 2-34). In JEM, defined possible intra-prediction modes include a planar prediction mode (predMode: 0), a DC prediction mode (predMode: 1), and 65 angular prediction modes (predMode: 2-66). It should be noted that planar and DC prediction modes may be referred to as non-directional prediction modes and that angular prediction modes may be referred to as directional prediction modes. It should be noted that the techniques described herein may be generally applicable regardless of the number of defined possible prediction modes.

[0023] For inter prediction coding, a motion vector (MV) identifies reference samples in a picture other than the picture of a video block to be coded and thereby exploits temporal redundancy in video. For example, a current video block may be predicted from reference block(s) located in previously coded frame(s) and a motion vector may be used to indicate the location of the reference block. A motion vector and associated data may describe, for example, a horizontal component of the motion vector, a vertical component of the motion vector, a resolution for the motion vector (e.g., one-quarter pixel precision, one-half pixel precision, one-pixel precision, two-pixel precision, four-pixel precision), a prediction direction and/or a reference picture index value. Further, a coding standard, such as, for example ITU-T H.265, may support motion vector prediction. Motion vector prediction enables a motion vector to be specified using motion vectors of neighboring blocks. Examples of motion vector prediction include advanced motion vector prediction (AMVP), temporal motion vector prediction (TMVP), so-called "merge" mode, and "skip" and "direct" motion inference. Additional examples of motion vector prediction include advanced temporal motion vector prediction (ATMVP) and Spatial-temporal motion vector prediction (STMVP).

[0024] As described above, intra prediction data or inter prediction data is used to produce reference sample values for a block of sample values. The difference between sample values included in a current PB, or another type of picture area structure, and associated reference samples (e.g., those generated using a prediction) may be referred to as residual data. Residual data may include respective arrays of difference values corresponding to each component of video data. Residual data may be in the pixel domain. A transform, such as, a discrete cosine transform (DCT), a discrete sine transform (DST), an integer transform, a wavelet transform, or a conceptually similar transform, may be applied to an array of difference values to generate transform coef-

ficients. In some cases, a transform process may include rotation, and/or performance of one or more one dimensional transforms. It should be noted that in ITU-T H.265 and JVET-M1001, a CU is associated with a transform unit (TU) structure having its root at the CU level. That is, in ITU-T H.265, an array of difference values may be partitioned for purposes of generating transform coefficients (e.g., four 8x8 transforms may be applied to a 16x16 array of residual values). For each component of video data, such sub-divisions of difference values may be referred to as Transform Blocks (TBs). It should be noted that in some cases, a core transform and a subsequent secondary transforms may be applied (in the video encoder) to generate transform coefficients. It should be noted that in ITU-T H.265, TBs are not necessarily aligned with PBs. Further, it should be noted that in ITU-T H.265, TBs may have the following sizes 4x4, 8x8, 16x16, and 32x32. It should be noted that in JEM, residual values corresponding to a CB are used to generate transform coefficients without further partitioning. That is, in JEM a QTBT leaf node may be analogous to both a PB and a TB in ITU-T H.265. It should be noted that in JEM, a core transform and a subsequent secondary transforms may be applied (in the video encoder) to generate transform coefficients. For a video decoder, the order of transforms is reversed. Further, in JEM, whether a secondary transform is applied to generate transform coefficients may be dependent on a prediction mode.

[0025] A quantization process may be performed on transform coefficients or residual sample values directly, e.g., in the case of palette coding quantization. Quantization approximates transform coefficients by amplitudes restricted to a set of specified values. Quantization essentially scales transform coefficients in order to vary the amount of data required to represent a group of transform coefficients. Quantization may include division of transform coefficients (or resulting values of addition of an offset value to transform coefficients) by a quantization scaling factor and any associated rounding functions (e.g., rounding to the nearest integer). Quantized transform coefficients may be referred to as coefficient level values. Inverse quantization (or "dequantization") may include multiplication of coefficient level values by the quantization scaling factor, and any reciprocal rounding or offset addition operations. It should be noted that as used herein the term quantization process in some instances may refer to division by a scaling factor to generate level values and multiplication by a scaling factor to recover transform coefficients in some instances. That is, a quantization process may refer to quantization in some cases and inverse quantization in some cases. Further, it should be noted that although in some of the examples below quantization processes are described with respect to arithmetic operations associated with decimal notation, such descriptions are for illustrative purposes and should not be construed as limiting. For example, the techniques described herein may be im-

plemented in a device using binary operations and the like. For example, multiplication and division operations described herein may be implemented using bit shifting operations and the like.

[0026] With respect to the equations used herein, the following arithmetic operators may be used:

+	Addition
−	Subtraction
*	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.
x % y	Modulus. Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0.

[0027] Further, the following logical operators may be used:

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.

[0028] Further, the following relational operators may be used:

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

[0029] Further, the following bit-wise operators may be used:

& 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.

[0030] Further, the following assignment operators may be used:

= 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).

[0031] Further, the following defined mathematical functions may be used:

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

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

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

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

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

[0032] FIGS. 2A-2B are conceptual diagrams illustrating examples of coding a block of video data. As illustrated in FIG. 2A, a current block of video data (e.g., a CB corresponding to a video component) is encoded by generating a residual by subtracting a set of prediction values from the current block of video data, performing a transformation on the residual, and quantizing the transform coefficients to generate level values. As illustrated in FIG. 2B, the current block of video data is decoded by performing inverse quantization on level values, performing an inverse transform, and adding a set of prediction values to the resulting residual. It should be noted that in the

examples in FIGS. 2A-2B, the sample values of the reconstructed block differs from the sample values of the current video block that is encoded. In this manner, coding may said to be lossy. However, the difference in sample values may be considered acceptable or imperceptible to a viewer of the reconstructed video.

[0033] Further, as illustrated in FIGS. 2A-2B, coefficient level values are generated using an array of scaling factors. In ITU-T H.265, an array of scaling factors is generated by selecting a scaling matrix and multiplying each entry in the scaling matrix by a quantization scaling factor. In ITU-T H.265, a scaling matrix is selected based in part on a prediction mode and a color component, where scaling matrices of the following sizes are defined: 4x4, 8x8, 16x16, and 32x32. It should be noted that in some examples, a scaling matrix may provide the same value for each entry (i.e., all coefficients are scaled according to a single value). In ITU-T H.265, the value of a quantization scaling factor, may be determined by a quantization parameter, QP. In ITU-T H.265, for a bit-depth of 8-bits, the QP can take 52 values from 0 to 51 and a change of 1 for QP generally corresponds to a change in the value of the quantization scaling factor by approximately 12%. Further, in ITU-T H.265, a QP value for a set of transform coefficients may be derived using a predictive quantization parameter value (which may be referred to as a predictive QP value or a QP predictive value) and an optionally signaled quantization parameter delta value (which may be referred to as a QP delta value or a delta QP value). In ITU-T H.265, a quantization parameter may be updated for each CU and a respective quantization parameter may be derived for each of the luma and chroma channels.

[0034] Referring again to FIG. 2A, quantized transform coefficients are coded into a bitstream. Quantized transform coefficients and syntax elements (e.g., syntax elements indicating a coding structure for a video block) may be entropy coded according to an entropy coding technique. An entropy coding process includes coding values of syntax elements using lossless data compression algorithms. Examples of entropy coding techniques include content adaptive variable length coding (CAVLC), context adaptive binary arithmetic coding (CABAC), probability interval partitioning entropy coding (PIPE), and the like. Entropy encoded quantized transform coefficients and corresponding entropy encoded syntax elements may form a compliant bitstream that can be used to reproduce video data at a video decoder. An entropy coding process, for example, CABAC, may include performing a binarization on syntax elements. Binarization refers to the process of converting a value of a syntax element into a series of one or more bits. These bits may be referred to as "bins." Binarization may include one or a combination of the following coding techniques: fixed length coding, unary coding, truncated unary coding, truncated Rice coding, Golomb coding, k-th order exponential Golomb coding, and Golomb-Rice coding. For example, binarization may

include representing the integer value of 5 for a syntax element as 00000101 using an 8-bit fixed length binarization technique or representing the integer value of 5 as 11110 using a unary coding binarization technique. As used herein each of the terms fixed length coding, unary coding, truncated unary coding, truncated Rice coding, Golomb coding, k-th order exponential Golomb coding, and Golomb-Rice coding may refer to general implementations of these techniques and/or more specific implementations of these coding techniques. For example, a Golomb-Rice coding implementation may be specifically defined according to a video coding standard, for example, ITU-T H.265.

[0035] In the example of CABAC in ITU-T H.265, for a particular bin, a context index provides an 8-bit variable, initValue. initValue is used to determine an initial most probable state (MPS) value for the bin (i.e., an MPS for a bin is one of 0 or 1) and a probability value of the bin being the least probably state (LPS). For example, a context index may indicate, at an initial state, that the MPS of a bin is 0 and the probability of the bin being 1 is 0.3. It should be noted that a context index may be selected based on values of previously coded syntax elements. For example, values of syntax elements associated with neighboring video blocks may be used to determine a context index.

[0036] In particular, ITU-T H.265 provides where initValue is used in the initialization of variables pStateIdx and valMps, where pStateIdx indicates a probability value and valMps indicates an LPS. In particular, initial values of pStateIdx and valMps are derived based on values of initValue as follows:

From the 8-bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

slopeIdx = initValue >> 4
offsetIdx = initValue & 15

The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

m = slopeIdx * 5 − 45
n = (offsetIdx << 3) − 16

The two values assigned to pStateIdx and valMps for the initialization are derived from SliceQpY, which is derived as follows:

SliceQpY = 26 + init_qp_minus26 + slice_qp_delta

where,
slice_qp_delta specifies the initial value of QpY (the luma channel quantization parameter) to be used for the coding blocks in the slice until modified; and
init_qp_minus26 plus 26 specifies the initial value of SliceQpY for each slice referring to the PPS.

Given the variables m and n, the initialization is specified as follows:
preCtxState = Clip3(1, 126, ((m * Clip3(0, 51, SliceQpY)) >> 4) + n)
valMps = (preCtxState <= 63) ? 0 : 1
pStateIdx = valMps ? (preCtxState − 64) : (63 − preCtxState)

[0037] It should be noted that in ITU-T H.265, a variable initType, which may have values of 0, 1, or 2, is determined based on slice type (i.e., whether a slice is a I slice, a P slice, or a B slice) and a CABAC initialization flag which is signaled in the picture parameter set (PPS). initType essentially determines a subset of possible context index value for a bin. For example, an initType value of 0 may indicates a context index is one of 0, 1, or 2, an initType value of 1 may indicate a context index is one of 3, 4, or 5, and initType value of 2 may indicate a context index is one of 6, 7, or 8. It should be noted that for a bin, difference context values do not necessary correspond to unique values of initValue.

[0038] In a manner similar to ITU-T H.265, JVET-M1001 uses an 8-bit variable, initValue to determine an initial most probable state (MPS) value for the bin and a probability value of the bin being the least probably state (LPS).

[0039] Binary arithmetic coding codes a series of 0's and 1's based on the principle of recursive interval subdivision. Essentially, for a binary string $(b_1,...,b_N)$, for an interval having an initial width (range) R_0, for $(b_1,...,b_N)$, R_0 is recursively divided as follows:

For i = (1,...,N),

if b_i equals LPS:

$$R_i = pLPS_i * R_{i-1},$$

otherwise

$$R_i = R_{i-1} - pLPS_i * R_{i-1},$$

Where,

LPS is the value of the least probably symbol, and

$pLPS_i$ is the estimated probability of b_i being the LPS.

[0040] As illustrated above, R_i is determined based on whether the observed value of b_i is the MPS or LPS. For example, for b_1 if R_0 is 512, the LPS is 0, and $pLPS_1 * R_0$ is 158, if b_1 is observed to be 1, R_1 = 354 and if b_1 is observed to be 0, R_1 = 158. As described above, in ITU-T H.265, a context index provides an MPS value for a bin and a probability value of the bin being the LPS, where the probability value of the bin being the LPS (i.e., pLPS) is indicated by one of 64 probability states. In particular, in ITU-T H.265, pStateIdx, is indexed such that, pStateIdx = 0 corresponds to a maximum LPS probability value, and decreasing LPS probabilities are indexed to higher values of pStateIdx. Further, in ITU-T H.265, R_0 is 512 which can be represented by 9-bits. However, R_i is quantized to a set {Q1,...,Q4} such that all possible values of $pLPS_i * R_{i-1}$ are pre-computed and indexed according to a 64x4 look-up table.

[0041] During encoding, after an interval for R_i is determined, i.e., based on $pLPS_i$ and the observed value of b_i, a renormalization process occurs. A renormalization process essentially determines whether bits are output (e.g., written to a bitstream) based on the value of R_i. Essentially, in renormalization, if R_i falls below a threshold value, and R_i is doubled and a bit value may be output. For example, in encoder side renormalization process described in ITU-T H.265, a determination is made if R_i is less than 256. If R_i is not less than 256, no bits are written to the bitstream and R_{i+1} is computed for b_{i+1}, using R_i. If R_i is less than 256, a 0-bit, a 1-bit, or no bit is conditionally written to the bitstream based on the lower end of the sub-interval, and R_i is doubled, and R_{i+1} is computed for b_{i+1} (i.e., based on the doubled value of R_i). A binary decoder receiving the output bits (i.e., the arithmetic code) recovers the binary string (b_1,...,b_N) by performing the same interval sub-division at each b_i as an encoder and by comparing subsets of the arithmetic code to R_i values.

[0042] In ITU-T H.265, the observed value of a bin is used to update the context index. ITU-T H.265 provides the following with respect to updating the context index based on the determined value of the bin:

```
if( binVal = = valMps )
        pStateIdx = transIdxMps( pStateIdx )
else {
        if( pStateIdx = = 0 )
                valMps = 1 - valMps
        pStateIdx = transIdxLps( pStateIdx )
}
```

Where,
valMps is the value of the MPS for the bin; and
transIdxMps() and transIdxLps() are a defined set of transition rules as provided in Table 1.

pStateIdx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
transIdxLps	0	0	1	2	2	4	4	5	6	7	8	9	9	11	11	12
transIdxMps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pStateIdx	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
transIdxLps	13	13	15	15	16	16	18	18	19	19	21	21	22	22	23	24
transIdxMps	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
pStateIdx	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
transIdxLps	24	25	26	26	27	27	28	29	29	30	30	30	31	32	32	33
transIdxMps	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
pStateIdx	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**
transIdxLps	33	33	34	34	35	35	35	36	36	36	37	37	37	38	38	63
transIdxMps	49	50	51	52	53	54	55	56	57	58	59	60	61	62	62	63

Table 1

[0043] Thus, in ITU-T H.265, if the bin value is determined to be equal to the MPS, the LPS probability value is decreased. If the bin value is determined to be equal to the LPS, the LPS probability value is increased and further, if the current probability state pLPS is 0.5 (i.e., pStateIdx equals 0), a LPS inversion occurs (i.e., the previous LPS value becomes the MPS). It should be noted, that according to ITU-T H.265, some syntax elements are entropy coded using arithmetic coding according to equal probability states, such coding may be referred to as bypass coding. In JVET-M1001 binary arithmetic coding is performed in a similar manner as ITU-T H.265.

[0044] Further, in one example, for each context variable, two variables pStateL and pStateH may be initialized as follows, where the variables pStateL and pStateH correspond to probability states presented at a low and a high precision, respectively.

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

slopeIdx = initValue >> 4

offsetIdx = initValue & 15

The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

m = slopeIdx * 5 − 45

n = (offsetIdx << 3) – 16

The variable preCtxStateIdx is derived from $SliceQp_Y$, which is derived as provided above, as follows:

preCtxStateIdx = Clip3(0, 127, ((m * Clip3(0, 51, $SliceQp_Y$)) >> 4) + n)

The values of pStateL and pStateH are derived by looking up initStateIdxToState as specified in Table 2, as follows:

pStateL = initStateIdxToState[preCtxStateIdx] >> 5

pStateH = initStateIdxToState[preCtxStateIdx] >> 1

preCtxStateIdx	0	1	2	3	4	5	6	7
initStateIdxToState	614	647	681	718	756	797	839	884
preCtxStateIdx	8	9	10	11	12	13	14	15
initStateIdxToState	932	982	1034	1089	1148	1209	1274	1342
preCtxStateIdx	16	17	18	19	20	21	22	23
initStateIdxToState	1414	1490	1569	1653	1742	1835	1933	2037
preCtxStateIdx	24	25	26	27	28	29	30	31
initStateIdxToState	2146	2261	2382	2509	2643	2785	2934	3091
preCtxStateIdx	32	33	34	35	36	37	38	39
initStateIdxToState	3256	3430	3614	3807	4011	4225	4452	4690
preCtxStateIdx	40	41	42	43	44	45	46	47
initStateIdxToState	4941	5205	5483	5777	6086	6412	6755	7116
preCtxStateIdx	48	49	50	51	52	53	54	55
initStateIdxToState	7497	7898	8320	8766	9235	9729	10249	10798
preCtxStateIdx	56	57	58	59	60	61	62	63
initStateIdxToState	11375	11984	12625	13300	14012	14762	15551	16384
preCtxStateIdx	64	65	66	67	68	69	70	71
initStateIdxToState	16384	17216	18005	18755	19467	20142	20783	21392
preCtxStateIdx	72	73	74	75	76	77	78	79
initStateIdxToState	21969	22518	23038	23532	24001	24447	24869	25270
preCtxStateIdx	80	81	82	83	84	85	86	87
initStateIdxToState	25651	26012	26355	26681	26990	27284	27562	27826
preCtxStateIdx	88	89	90	91	92	93	94	95
initStateIdxToState	28077	28315	28542	28756	28960	29153	29337	29511
preCtxStateIdx	96	97	98	99	100	101	102	103
initStateIdxToState	29676	29833	29982	30124	30258	30385	30506	30621
preCtxStateIdx	104	105	106	107	108	109	110	111
initStateIdxToState	30730	30834	30932	31025	31114	31198	31277	31353
preCtxStateIdx	112	113	114	115	116	117	118	119
initStateIdxToState	31425	31493	31558	31619	31678	31733	31785	31835
preCtxStateIdx	120	121	122	123	124	125	126	127
initStateIdxToState	31883	31928	31970	32011	32049	32086	32120	32153

Table 2

[0045] Further, in one example, for each context variable, two variables pStateL and pStateH may be initialized as follows:

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

```
slopeIdx = initValue >> 4
offsetIdx = initValue & 15
```

The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

```
m = slopeIdx * 5 – 45
n = ( offsetIdx << 3 ) – 16
```

The variable preCtxStateIdx is derived from SliceQpY, which is derived as provided above, as follows:

```
preCtxStateIdx = ((( m * Clip3( 0, 51, SliceQpY ) ) >> 4 ) + n - 64 ) * 19
mps = preCtxStateIdx < 0 ? 0 : 1
preCtxStateIdx = min(1216, mps == 0 ? –preCtxStateIdx-1 : preCtxStateIdx)
state = ((511 – ( preCtxStateIdx & 255)) * 32) >> ( preCtxStateIdx >> 8)
state = mps ? 32767 – p : p
```

The values of pStateL and pStateH are derived as follows:

```
pStateL = state >> 5
pStateH = state >> 1
```

[0046] Further, in one example, for each context variable, two variables pStateL and pStateH may be initialized as follows:

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

```
slopeIdx = initValue >> 4
offsetIdx = initValue & 15
```

The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

```
m = slopeIdx * 6 – 53
n = ( offsetIdx << 3 ) – 16
```

The variable preCtxStateIdx is derived from $SliceQp_Y$, which is derived as provided above, as follows:

```
preCtxStateIdx = ((( m * Clip3( 0, 51, SliceQpY ) ) >> 4 ) + 19*n - 1216 )
mps = preCtxStateIdx < 0 ? 0 : 1
preCtxStateIdx = min(1216, mps == 0 ? –preCtxStateIdx-1 : preCtxStateIdx)
state = ((511 – ( preCtxStateIdx & 255)) * 16) >> ( preCtxStateIdx >> 8)
state = mps ? 16383 – p : p
```

The values of pStateL and pStateH are derived as follows:

```
pStateL = state >> 4
pStateH = state
```

[0047] Further, in one example, for each context variable, two variables pStateL and pStateH may be initialized as follows:

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

slopeIdx = initValue >> 4

offsetIdx = initValue & 15

The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

m = slopeIdx - 8

n = (offsetIdx − 8) * 25

The variable preCtxStateIdx is derived from $SliceQp_Y$, which is derived as provided above, as follows:

preCtxStateIdx = m * Clip3(0, 51, $SliceQp_Y$) + n

mps = preCtxStateIdx < 0 ? 0 : 1

preCtxStateIdx = 3*min(255, mps == 0 ? –preCtxStateIdx-1 : preCtxStateIdx)

state = ((255 – (preCtxStateIdx & 127)) * 32) >> (preCtxStateIdx >> 7)

state = mps ? 16383 – p : p

The values of pStateL and pStateH are derived as follows:

pStateL = state >> 4
pStateH = state

[0048] With respect to pStateL and pStateH the arithmetic decoding process for a binary decision may be as follows:

Inputs to this process are the variables ctxTable, ctxIdx, ivlCurrRange, and ivlOffset.
Outputs of this process are the decoded value binVal, and the updated variables ivlCurrRange and ivlOffset.

1. The value of the variable ivlLpsRange is derived as follows:

– Given the current value of ivlCurrRange, the variable qRange is derived as follows:
qRange = ivlCurrRange >> 5

– Given pStateL and pStateH associated with ctxTable and ctxIdx, the variable pState is derived as follows:
pState = pStateH + (pStateL << 4)

– Given qRange and pState, ivlLpsRange is derived as follows:
ivlLpsRange = ((qRange * ((pState >= 16384 ? 32767 – pState : pState) >> 9)) >> 1) + 4

– Given pState, valMps is derived as follows:
valMps = (pState >= 16384) ? 1 : 0

2. The variable ivlCurrRange is set equal to ivlCurrRange - ivlLpsRange and the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 – valMps, ivlOffset is decremented by ivlCurrRange, and ivlCurrRange is set equal to ivlLpsRange.

– Otherwise, the variable binVal is set equal to valMps.

Given the value of binVal, the state update is performed as specified below. Depending on the current value of ivlRange, renormalization is performed.

[0049] With respect to pStateL and pStateH the state update process may be as follows:

Inputs to this process are the current pStateL and pStateH values of the context variable associated with ctxTable and ctxIdx, and the decoded value binVal.
Outputs of this process are the updated pStateL and pStateH values of the context variable associated with ctxIdx.
Depending on the decoded value binVal, the update of pStateL and pStateH associated with ctxIdx are derived from ctxInc, which is specified in a table providing an assignment of ctxInc to syntax elements with context coded bins (e.g., Table 26 below), as follows:

- Given ctxInc associtated with ctxIdx, two variables shiftL and shiftH are derived as follows:

```
shiftL = ( shift [ctxInc] >> 2 ) + 2;
shiftH = ( shift [ctxInc] & 3 ) + 3 + shiftL;
```

- Given the values of shiftL and shiftH, updates of pStateL and pStateH are derived as follows:

```
if( binVal  ==  1 ) {
        pStateL += ( 1023 - pStateL ) >> shiftL;
        pStateH += ( 16383 - pStateH ) >> shiftH;
}
else {
        pStateL -= pStateL >> shiftL;
        pStateH -= pStateH >> shiftH;
}
```

[0050] Tables 3-24 illustrate the tile group data syntax as currently expressed in JVET-M1001. It should be noted that as currently expressed in JVET-M1001 may refer to as currently being described in JVET-M1001 and/or as currently being implemented in the corresponding reference software. Further, it should be noted that with respect to Tables 3-24, the descriptor ae(v) refers to a context-adaptive arithmetic entropy-coded syntax element. Further, for each syntax element in Tables 3-24, a binarization is provided.

[0051] Tables 3A etc. illustrate the slice data syntax as currently expressed in JVET-L1001. It should be noted that as currently expressed in JVET-L1001 may refer to as currently being described in JVET-L1001 and/or as currently being implemented in the corresponding reference software. Further, it should be noted that with respect to Tables 3A etc., the descriptor ae(v) refers to a context-adaptive arithmetic entropy-coded syntax element. Further, for each syntax element in Tables 3A etc. , a binarization is provided.

[0052] FL refers to a fixed-length binarization with cMax as an input. An 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.

[0053] TR refers to a Truncated Rice binarization process with cMax and cRiceParam as an inputs. A TR binarization A TR bin string is a concatenation of a prefix bin string and, when present, a suffix bin string. For the derivation of the prefix bin string, the following applies:

-The prefix value of symbolVal, prefixVal, is derived as follows:

prefixVal = symbolVal >> cRiceParam

-The prefix of the TR bin string is specified as follows:

-If prefixVal is less than cMax >> cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0.

-Otherwise, the bin string is a bit string of length cMax >> cRiceParam with all bins being equal to 1.

When cMax is greater than symbolVal and cRiceParam is greater than 0, the suffix of the TR bin string is present and it is derived as follows:

-The suffix value suffixVal is derived as follows:

suffixVal = symbolVal – ((prefixVal) << cRiceParam)

– The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified above for suffixVal with a cMax value equal to (1 << cRiceParam) – 1.

[0054] TB refers to a Truncated Binary (TB) binarization with value synVal and cMax as inputs. The bin string of the TB binarization process of a syntax element synVal is specified as follows:

n = cMax + 1
k = Floor(Log2(n))
u = (1 << (k + 1)) – n

- If synVal is less than u, the TB bin string is derived by invoking the FL binarization process 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 for (synVal + u) with a cMax value equal to (1 << (k + 1)) – 1.

[0055] EGk refers to a k-th order Exp-Golomb (EGk) binarization. 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 )
```

[0056] In Tables 3-24, if Specified is provided as the binarization for a syntax element, the binarization is provided with the semantics of the syntax element below.

[0057] Table 3 illustrates the coding tree unit syntax in JVET-M1001 and Table 3 illustrates the dual tree implicit quadtree split syntax in JVET-M1001.

coding_tree_unit( ) {	**Descriptor**	**Binarization**
xCtb = (CtbAddrInRs % PicWidthInCtbsY) << CtbLog2SizeY		
yCtb = (CtbAddrInRs / PicWidthInCtbsY) << CtbLog2SizeY		
if(tile_group_sao_luma_flag || tile_group_sao_chroma_flag)		
sao(xCtb >> CtbLog2SizeY, yCtb >> CtbLog2SizeY)		
if(tile_group_alf_enabled_flag){		
alf_ctb_flag[0][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)	FL; cMax = 1
if(alf_chroma_idc == 1 || alf_chroma_idc == 3)		
alf_ctb_flag[1][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)	FL; cMax = 1
if(alf_chroma_idc == 2 || alf_chroma_idc == 3)		
alf_ctb_flag[2][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)	FL; cMax = 1
}		
if((tile_group_type == I || CurrPicIsOnlyRef) && qtbtt_dual_tree_intra_flag)		
dual_tree_implicit_qt_split (xCtb, yCtb, CtbLog2SizeY, 0)		
else		
coding_tree(xCtb, yCtb, 1 << CtbLog2SizeY, 1 << CtbLog2SizeY, 0, 0, 0, SINGLE_TREE)		
}		

Table 3

dual_tree_implicit_qt_split(x0, y0, log2CbSize, cqtDepth) {	Descriptor	Binarization
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_tree(x0, y0, 1 << log2CbSize, 1 << log2CbSize , cqtDepth, 0, 0, 0, DUAL_TREE_LUMA)		
coding_tree(x0, y0, 1 << log2CbSize, 1 << log2CbSize, cqtDepth,, 0, 0, 0, DUAL_TREE_CHROMA)		
}		
}		

Table 4

[0058] In Tables 3A etc. , if Specified is provided as the binarization for a syntax element, the binarization is provided with the semantics of the syntax element below.

[0059] Table 3A illustrates the coding tree unit syntax in JVET-L1001 and Table 4A illustrates the dual tree implicit quadtree split syntax in JVET-L1001.

coding_tree_unit( ) {	Descriptor	Binarization
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)	FL; cMax = 1
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)	FL; cMax = 1
if(alf_chroma_idc == 2 || alf_chroma_idc == 3)		
alf_ctb_flag[2][xCtb >> Log2CtbSize][yCtb >> Log2CtbSize]	ae(v)	FL; cMax = 1
}		
}		
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)		
}		

Table 3A

dual_tree_implicit_qt_split(x0, y0, log2CbSize, cqtDepth) {	Descriptor	Binarization
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)		
}		
}		

Table 4A

[0060] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 3.

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.

[0061] Table 5 illustrates the sample adaptive offset syntax in JVET-M1001.

sao(rx, ry) {	Descriptor	Binarization
if(rx > 0) {		
leftCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − 1]]		
if(leftCtbInTile)		
sao_merge_left_flag	ae(v)	FL; cMax = 1
}		
if(ry > 0 && !sao_merge_left_flag) {		
upCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − PicWidthInCtbsY]]		
if(upCtbInTile)		
sao_merge_up_flag	ae(v)	FL; cMax = 1
}		
if(!sao_merge_up_flag && !sao_merge_left_flag)		
for(cIdx = 0; cIdx < (ChromaArrayType != 0 ? 3 : 1); cIdx++)		
if((tile_group_sao_luma_flag && cIdx == 0) || (tile_group_sao_chroma_flag && cIdx > 0)) {		
if(cIdx == 0)		
sao_type_idx_luma	ae(v)	TR; cMax = 2, cRiceParam = 0
else if(cIdx == 1)		
sao_type_idx_chroma	ae(v)	TR; cMax = 2, cRiceParam = 0
if(SaoTypeIdx[cIdx][rx][ry] != 0) {		
for(i = 0; i < 4; i++)		
sao_offset_abs[cIdx][rx][ry][i]	ae(v)	TR; cMax = (1 << (Min(bitDepth, 10) − 5) − 1, cRiceParam = 0
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)	FL; cMax = 1
sao_band_position[cIdx][rx][ry]	ae(v)	FL; cMax = 31
} else {		
if(cIdx == 0)		
sao_eo_class_luma	ae(v)	FL; cMax = 3
if(cIdx == 1)		
sao_eo_class_chroma	ae(v)	FL; cMax = 3
}		
}		
}		
}		

Table 5

[0062] Table 5A illustrates the sample adaptive offset syntax in JVET-L1001.

sao(rx, ry) {	Descriptor	Binarization
if(rx > 0) {		
leftCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − 1]]		
if(leftCtbInTile)		
sao_merge_left_flag	ae(v)	FL; cMax = 1
}		
if(ry > 0 && !sao_merge_left_flag) {		
upCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs − PicWidthInCtbsY]]		
if(upCtbInTile)		
sao_merge_up_flag	ae(v)	FL; cMax = 1
}		
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)	TR; cMax = 2, cRiceParam = 0
else if(cIdx == 1)		
sao_type_idx_chroma	ae(v)	TR; cMax = 2, cRiceParam = 0
if(SaoTypeIdx[cIdx][rx][ry] != 0) {		
for(i = 0; i < 4; i++)		
sao_offset_abs[cIdx][rx][ry][i]	ae(v)	TR; cMax = (1 << (Min(bitDepth, 10) − 5) − 1, cRiceParam = 0
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)	FL; cMax = 1
sao_band_position[cIdx][rx][ry]	ae(v)	FL; cMax = 31
} else {		
if(cIdx == 0)		
sao_eo_class_luma	ae(v)	FL; cMax = 3
if(cIdx == 1)		
sao_eo_class_chroma	ae(v)	FL; cMax = 3
}		
}		
}		
}		

Table 5A

[0063] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 5.

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 6 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.

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

Table 6

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 ]
```

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 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.

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

Table 7

[0064] Table 7A illustrates the coding quadtree syntax in JVET-L1001.

coding_quadtree(x0, y0, log2CbSize, cqtDepth, treeType) {	Descriptor	Binarization
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)	FL; cMax = 1
}		
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)		
}		

Table 7A

[0065] JVET-L1001 provides the following definitions for the respective syntax elements illustrated in Table 7A

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.

[0066] Table 8 illustrates the Coding tree syntax in JVET-M1001.

coding_tree(x0, y0, cbWidth, cbHeight, mttDepth, depthOffset, partIdx, treeType) {	Descriptor	Binarization
if((allowSplitBtVer || allowSplitBtHor || allowSplitTtVer || allowSplitTtHor || allowSplitQt) && (x0 + cbWidth <= pic_width_in_luma_samples) && (y0 + cbHeight <= pic_height_in_luma_samples))		
split_cu_flag	ae(v)	FL; cMax = 1
if(cu_qp_delta_enabled_flag && (cqtDepth + mttDepth) <= diff_cu_qp_delta_depth) {		
IsCuQpDeltaCoded = 0		
CuQpDeltaVal = 0		
CuQgTopLeftX = x0		
CuQgTopLeftY = y0		
}		
if(split_cu_flag) {		
if(allowSplitQt && (allowSplitBtVer || allowSplitBtHor || allowSplitTtVer || allowSplitTtHor))		
split_qt_flag	ae(v)	FL; cMax = 1
if(!split_qt_flag) {		
if((allowSplitBtHor || allowSplitTtHor) && (allowSplitBtVer || allowSplitTtVer))		
mtt_split_cu_vertical_flag	ae(v)	FL; cMax = 1
if((allowSplitBtVer && allowSplitTtVer && mtt_split_cu_vertical_flag) || (allowSplitBtHor && allowSplitTtHor && !mtt_split_cu_vertical_flag))		
mtt_split_cu_binary_flag	ae(v)	FL; cMax = 1
if(MttSplitMode[x0][y0][mttDepth] == SPLIT_BT_VER) {		
depthOffset += (x0 + cbWidth > pic_width_in_luma_samples) ? 1 : 0		
x1 = x0 + (cbWidth / 2)		

coding_tree(x0, y0, cbWidth / 2, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)		
if(x1 < pic_width_in_luma_samples)		
coding_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)		
coding _tree(x0, y0, cbWidth, cbHeight / 2, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)		
if(y1 < pic_height_in_luma_samples)		
coding_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)		
coding _tree(x0, y0, cbWidth / 4, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)		
coding_tree(x1, y0, cbWidth / 2, cbHeight, cqtDepth, mttDepth + 1, depthOffset, 1, treeType)		
coding_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)		
coding_tree(x0, y0, cbWidth, cbHeight / 4, cqtDepth, mttDepth + 1, depthOffset, 0, treeType)		

codin _tree(x0, y1, cbWidth, cbHeight / 2, cqtDepth, mttDepth + 1, depthOffset, 1, treeType)		
coding_tree(x0, y2, cbWidth, cbHeight / 4, cqtDepth, mttDepth + 1, depthOffset, 2 , treeType)		
}		
} else {		
x1 = x0 + (cbWidth / 2)		
y1 = y0 + (cbHeight / 2)		
coding_tree(x0, y0, cbWidth / 2, cbHeight / 2, cqtDepth + 1, 0, 0, 0, treeType)		
if(x1 < pic_width_in_luma_samples)		
coding_tree(x1, y0, cbWidth / 2, cbHeight / 2, cqtDepth + 1, 0, 0, 1, treeType)		
if(y1 < pic_height_in_luma_samples)		
coding_tree(x0, y1, cbWidth / 2, cbHeight / 2, cqtDepth + 1, 0, 0, 2, treeType)		
if(y1 < pic_height_in_luma_samples && x1 < pic_width_in_luma_samples)		
coding_tree(x1, y1, cbWidth / 2, cbHeight / 2, cqtDepth + 1, 0, 0, 3, treeType)		
}		
} else		
coding_unit(x0, y0, cbWidth, cbHeight, treeType)		
}		

Table 8

[0067] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 8.

[0068] Table 8A illustrates the multi type tree syntax in JVET-L1001.

multi_type_tree(x0, y0, cbWidth, cbHeight, mttDepth, depthOffset, partIdx, treeType) {	**Descriptor**	**Binarization**
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)	FL; cMax = 1
if(mtt_split_cu_flag) {		
if((allowSplitBtHor || allowSplitTtHor) && (allowSplitBtVer || allowSplitTtVer))		
mtt_split_cu_vertical_flag	ae(v)	FL; cMax = 1
if((allowSplitBtVer && allowSplitTtVer && mtt_split_cu_vertical_flag) || (allowSplitBtHor && allowSplitTtHor && !mtt_split_cu_vertical_flag))		
mtt_split_cu_binary_flag	ae(v)	FL; cMax = 1
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, depth Offset, 0, treeType)		
multi_type_tree(x0, y1, cbWidth, cbHeight / 2, mttDepth + 1, depth Offset, 1, treeType)		
multi_type_tree(x0, y2, cbWidth, cbHeight / 4, mttDepth + 1, depth Offset, 2 , treeType)		
}		
} else		
coding_unit(x0, y0, cbWidth, cbHeight, treeType)		
}		

Table 8A

[0069] JVET-L1001 provides the following definitions for the respective syntax elements illustrated in Table 8A.

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, or four coding units using a quad split, as indicated by the syntax elements split_qt_flag, and mtt_split_cu_binary_flag. The binary or ternary split can be either vertical or horizontal as indicated by the syntax element mtt_split_cu_vertical_flag.
When split_cu_flag is not present, the value of split_cu_flag is inferred as follows:

- If one or more of the following conditions are true, the value of split_cu_flag is inferred to be equal to 1:
  - x0 + cbWidth is greater than pic_width_in_luma_samples.
  - y0 + cbHeight is greater than pic_height_in_luma_samples.
- Otherwise, the value of split_cu_flag is inferred to be equal to 0.

split_qt_flag specifies whether a coding unit is split into coding units with half horizontal and vertical size.

When split_qt_flag is not present, the following applies:

- If one or more of the following conditions are true, the value of split_qt_flag is inferred to be equal to 1:
  - allowSplitQt is equal to TRUE.
- Otherwise, the value of split_qt_flag is inferred to be equal to 0.

mtt_split_cu_vertical_flag equal to 0 specifies that a coding unit is split horizontally. mtt_split_cu_vertical_flag equal to 1 specifies that a coding unit is split vertically

When mtt_split_cu_vertical_flag is not present, it is inferred as follows:

- If allowSplitBtHor is equal to TRUE or allowSplitTtHor is equal to TRUE, the value of mtt_split_cu_vertical_flag is inferred to be equal to 0.
- Otherwise, the value of mtt_split_cu_vertical_flag is inferred to be equal to 1.

mtt_split_cu_binary_flag equal to 0 specifies that a coding unit is split into three coding units using a ternary split. mtt_split_cu_binary_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split.
When mtt_split_cu_binary_flag is not present, it is inferred as follows:

- If allowSplitBtVer is equal to FALSE and allowSplitBtHor is equal to FALSE, the value of mtt_split_cu_binary_flag is inferred to be equal to 0.
- Otherwise, if allowSplitTtVer is equal to FALSE and allowSplitTtHor is equal to FALSE, the value of mtt_split_cu_binary_flag is inferred as to be equal to 1.
- Otherwise, if allowSplitBtHor is equal to TRUE and allowSplitTtVer is equal to TRUE, the value of mtt_split_cu_binary_flag is inferred to be equal to !mtt_split_cu_vertical_flag.
- Otherwise (allowSplitBtVer is equal to TRUE and allowSplitTtHor is equal to TRUE), the value of mtt_split_cu_binary_flag is inferred to be equal to mtt_split_cu_vertical_flag.

The variable MttSplitMode[x][y][mttDepth] is derived from the value of mtt_split_cu_vertical_flag and from the value of mtt_split_cu_binary_flag as defined in Table 9 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

Table 9

[0070] Table 10 illustrates the coding unit syntax in JVET-M1001.

coding_unit(x0, y0, cbWidth, cbHeight, treeType) {	Descriptor	Binarization
if(tile_group_type != I) {		
if(treeType != DUAL_TREE_CHROMA)		
cu_skip_flag[x0][y0]	ae(v)	FL; cMax = 1
if(cu_skip_flag[x0][y0] = = 0)		
pred_mode_flag	ae(v)	FL; cMax = 1
if((tile_group_type = = I && cu_skip_flag[x0][y0] = =0) || (tile_group_type != I && (cu_skip_flag[x0][y0] || CuPredMode[x0][y0] != MODE_INTRA)) && sps_cpr_enabled_flag && cbWidth < 32 && cbHeight < 32)		
pred_mode_cpr_flag	ae(v)	FL; cMax = 1
}		
if(CuPredMode[x0][y0] = = MODE_INTRA) {		
if(pcm_enabled_flag && cbWidth >= MinIpcmCbSizeY && cbWidth <= MaxIpcmCbSizeY && cbHeight >= MinIpcmCbSizeY && cbHeight <= MaxIpcmCbSizeY)		
pcm_flag[x0][y0]	ae(v)	FL; cMax = 1
if(pcm_flag[x0][y0]) {		
while(!byte_aligned( ))		
pcm_alignment_zero_bit	f(1)	0
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)	TR; cMax = 2, cRiceParam = 0
if (intra_luma_ref_idx[x0][y0] = = 0 && (cbWidth <= MaxTbSizeY || cbHeight <= MaxTbSizeY) && (cbWidth * cbHeight > MinTbSizeY * MinTbSizeY))		
intra_subpartitions_mode_flag[x0][y0]	ae(v)	FL; cMax = 1

if(intra_subpartitions_mode_flag[x0][y0] = = 1 && cbWidth <= MaxTbSizeY && cbHeight <= MaxTbSizeY)		
intra_subpartitions_split_flag[x0][y0]	ae(v)	FL; cMax = 1
if(intra_luma_ref_idx[x0][y0] = = 0 && intra_subpartitions_mode_flag[x0][y0] = = 0)		
intra_luma_mpm_flag[x0][y0]	ae(v)	FL; cMax = 1
if(intra_luma_mpm_flag[x0][y0])		
intra_luma_mpm_idx[x0][y0]	ae(v)	TR; cMax = 5, cRiceParam = 0
else		
intra_luma_mpm_remainder[x0][y0]	ae(v)	TB; cMax = 60,
}		
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA)		
intra_chroma_pred_mode[x0][y0]	ae(v)	Specified
}		
} else if(treeType != DUAL_TREE_CHROMA) { /* MODE_INTER */		
if(cu_skip_flag[x0][y0] = = 0)		
merge_flag[x0][y0]	ae(v)	FL; cMax = 1
if(merge_flag[x0][y0]) {		
merge_data(x0, y0, cbWidth, cbHeight)		
} else {		
if(tile_group_type = = B)		
inter_pred_idc[x0][y0]	ae(v)	Specified
if(sps_affine_enabled_flag && cbWidth >= 16 && cbHeight >= 16) {		
inter_affine_flag[x0][y0]	ae(v)	FL; cMax = 1
if(sps_affine_type_flag && inter_affine_flag[x0][y0])		
cu_affine_type_flag[x0][y0]	ae(v)	FL; cMax = 1
}		
if(inter_pred_idc[x0][y0] == PRED_BI && refIdxSymL0 > -1 && refIdxSymL1 > -1 && inter_affine_flag[x0][y0] == 0)		

sym_mvd_flag[x0][y0]	ae(v)	FL; cMax = 1
if(inter_pred_idc[x0][y0] != PRED_L1) {		
if(num_ref_idx_l0_active_minus1 > 0 && !sym_mvd_flag[x0][y0])		
ref_idx_l0[x0][y0]	ae(v)	TR; cMax = num_ref_idx_l0_active_minus1, cRiceParam = 0
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)	FL; cMax = 1
} 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 && !sym_mvd_flag[x0][y0])		
ref_idx_l1[x0][y0]	ae(v)	TR; cMax = num_ref_idx_l1_active_minus1, cRiceParam = 0
if(mvd_l1_zero_flag && inter_pred_idc[x0][y0] == PRED_BI) {		
MvdL1[x0][y0][0] = 0		
MvdL1[x0][y0][1] = 0		
MvdCpL1[x0][y0][0][0] = 0		
MvdCpL1[x0][y0][0][1] = 0		
MvdCpL1[x0][y0][1][0] = 0		
MvdCpL1[x0][y0][1][1] = 0		
MvdCpL1[x0][y0][2][0] = 0		

MvdCpL1[x0][y0][2][1] = 0		
} else {		
if(!sym_mvd_flag[x0][y0]) {		
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)	FL; cMax = 1
} 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)) || (inter_affine_flag == 1 && (MvdCpL0[x0][y0][0] [0] != 0 || MvdCpL0[x0][y0][0] [1] != 0 || MvdCpL1[x0][y0][0] [0] != 0 || MvdCpL1[x0][y0][0] [1] != 0 || MvdCpL0[x0][y0][1] [0] != 0 || MvdCpL0[x0][y0][1] [1] != 0 || MvdCpL1[x0][y0][1] [0] != 0 || MvdCpL1[x0][y0][1] [1] != 0 || MvdCpL0[x0][y0][2] [0] != 0 || MvdCpL0[x0][y0][2] [1] != 0 || MvdCpL1[x0][y0][2] [0] != 0 || MvdCpL1[x0][y0][2] [1] != 0 ||))){		
if(!sps_cpr_enabled_flag || !(inter_pred_idc[x0][y0] == PRED_L0 && ref_idx_l0[x0][y0] == num_ref_idx_l0_active_minus1))		
amvr_flag[x0][y0]	ae(v)	FL; cMax = 1
if(amvr_flag[x0][y0])		
amvr_4pel_flag[x0][y0]	ae(v)	FL; cMax = 1
}		
if(sps_gbi_enabled_flag && inter_pred_idc[x0][y0] == PRED_BI && cbWidth * cbHeight >= 256)		
gbi_idx[x0][y0]	ae(v)	TR; cMax = NoBackwardPredFlag ? 4: 2
}		
}		
if(!pcm_flag[x0][y0]) {		

if(CuPredMode[x0][y0] != MODE_INTRA && cu_skip_flag[x0][y0] == 0)		
cu_cbf	ae(v)	
if(cu_cbf)		
if(CuPredMode[x0][y0] != MODE_INTRA && sps_sbt_enable_flag) {		
if(cbWidth <= maxSbtSize && cbHeight <= maxSbtSize) {		
allowSbtVerHalf = cbWidth >= 8		
allowSbtVerQuad = cbWidth >= 16		
allowSbtHorHalf = cbHeight >= 8		
allowSbtHorQuad = cbHeight >= 16		
if(allowSbtVerHalf || allowSbtHorHalf || allowSbtVerQuad || allowSbtHorQuad)		
cu_sbt_flag[x0][y0]	ae(v)	FL; cMax = 1
}		
if(cu_sbt_flag[x0][y0]) {		
if((allowSbtVerHalf || allowSbtHorHalf) && (allowSbtVerQuad || allowSbtHorQuad))		
cu_sbt_quad_flag[x0][y0]	ae(v)	FL; cMax = 1
if((cu_sbt_quad_flag[x0][y0] && allowSbtVerQuad && allowSbtHorQuad) || (!cu_sbt_quad_flag[x0][y0] && allowSbtVerHalf && allowSbtHorHalf))		
cu_sbt_horizontal_flag[x0][y0]	ae(v)	FL; cMax = 1
cu_sbt_pos_flag[x0][y0]	ae(v)	FL; cMax = 1
}		
}		
transform_tree(x0, y0, cbWidth, cbHeight, treeType)		
}		
}		

Table 10

[0071] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 10.

[0072] Table 10A illustrates the coding unit syntax in JVET-L1001.

coding_unit(x0, y0, cbWidth, cbHeight, treeType) {	Descriptor	Binarization
if(slice_type != I) {		
cu_skip_flag[x0][y0]	ae(v)	FL; cMax = 1
if(cu_skip_flag[x0][y0] = = 0)		
pred_mode_flag	ae(v)	FL; cMax = 1
}		
if(CuPredMode[x0][y0] = = MODE_INTRA) {		
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA) {		
intra_luma_mpm_flag[x0][y0]	ae(v)	TR; cRiceParam = 0
if(intra_luma_mpm_flag[x0][y0])		
intra_luma_mpm_idx[x0][y0]	ae(v)	FL; cMax = 63
else		
intra_luma_mpm_remainder[x0][y0]	ae(v)	FL; cMax = 1
}		
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA)		
intra_chroma_pred_mode[x0][y0]	ae(v)	Specified
} 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)	FL; cMax = 1
if(merge_affine_flag[x0][y0] = = 0 && MaxNumMergeCand > 1)		
merge_idx[x0][y0]	ae(v)	FL; cMax = maxNumMergeCand -1, cRiceParam = 0
} else {		
merge_flag[x0][y0]	ae(v)	FL; cMax = 1
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)	FL; cMax = 1
if(merge_affine_flag[x0][y0] == 0 && MaxNumMergeCand > 1)		
merge_idx[x0][y0]	ae(v)	TR; cMax = maxNumMergeCand -1, cRiceParam = 0
} else {		
if(slice_type == B)		
inter_pred_idc[x0][y0]	ae(v)	Specified
if(sps_affine_enabled_flag && cbWidth >= 16 && cbHeight >= 16) {		
inter_affine_flag[x0][y0]	ae(v)	FL; cMax = 1
if(sps_affine_type_flag && inter_affine_flag[x0][y0])		
cu_affine_type_flag[x0][y0]	ae(v)	FL; cMax = 1
}		
if(inter_pred_idc[x0][y0] != PRED_L1) {		
if(num_ref_idx_l0_active_minus1 > 0)		
ref_idx_l0[x0][y0]	ae(v)	TR; cMax = maxNumMergeCand -1, cRiceParam = 0
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)	FL; cMax = 1
} 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)	TR; cMax = maxNumMergeCand -1, cRiceParam = 0
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)	FL; cMax = 1
} 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)	TR; cMax = 2, cRiceParam = 0
}		
}		
}		
if(CuPredMode[x0][y0] != MODE_INTRA && cu_skip_flag[x0][y0] == 0)		
cu_cbf	ae(v)	FL; cMax = 1
if(cu_cbf)		
transform_tree(x0, y0, cbWidth, cbHeight, treeType)		
}		

Table 10A

[0073] JVET-L1001 provides the following definitions for the respective syntax elements illustrated in Table 10A.

cu_skip_flag[x0][y0] equal to 1 specifies that for the current coding unit, when decoding a P or B tile group, no more syntax elements except one or more of the following are parsed after cu_skip_flag[x0][y0]: the merge plus MVD flag mmvd_flag[x0][y0], the merge plus MVD index mmvd_merge_flag[x0][y0], the merge plus MVD distance index mmvd distance_idx[x0][y0], the merge plus MVD direction index mmvd_direction_idx[x0][y0], the merging candidate index merge_idx[x0][y0] the subblock-based merge flag merge_subblock_flag[x0][y0], the subblock-based merging candidate index merge_subblock_idx[x0][y0], the merge triangle flag merge_triangle_flag[x0][y0], and the merge triangle index merge_triangle_index[x0][y0]. 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.

pred_mode_cpr_flag equal to 0 specifies that the current coding unit is not coded in cpr prediction mode pred_mode_idx equal to 1 specifies that the current coding unit is coded in cpr 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_ cpr _flag is equal to 0, CuPredMode[x][y] is set equal to MODE_INTER.
- Otherwise (pred_mode_ cpr_ flag is equal to 1), CuPredMode[x][y] is set equal to MODE_CPR.
- When pred_mode_cpr_flag is not present, the variable CuPredMode[x][y] is inferred to be equal to MODE_INTRA when decoding a I tile group, and MODE_INTER when decoding a P or B tile group, is derived as follows 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 11.
When intra_luma_ref_idx[x0][y0] is not present it is inferred to be equal to 0.

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

Table 11

intra_subpartitions_mode_flag[x0][y0] equal to 1 specifies that the current intra coding unit is partitioned into NumIntraSubPartitions[x0][y0] rectangular transform block subpartitions. intra_subpartitions_mode_flag[x0][y0] equal to 0 specifies that the current intra coding unit is not partitioned into rectangular transform block subpartitions.
When intra_subpartitions_mode_flag[x0][y0] is not present, it is inferred to be equal to 0.
intra_subpartitions_split_flag[x0][y0] specifies whether the intra subpartitions split type is horizontal or vertical. When intra_subpartitions_mode_flag[x0][y0] is not present, it is inferred to be equal to 0.
The variable IntraSubPartitionsSplitType specifies the type of split used for the current luma coding block as illustrated in Table 12. IntraSubPartitionsSplitType is derived as follows:

- If intra_subpartitions_mode_flag[x0][y0] is equal to 0, IntraSubPartitionsSplitType is set equal to 0.
- Otherwise, the IntraSubPartitionsSplitType is set equal to 1 + intra_subpartitions_split_flag[x0][y0].

IntraSubPartitionsSplitType	Name of IntraSubPartitionsSplitType
0	ISP_NO_SPLIT
1	ISP_HOR_SPLIT
2	ISP_VER_SPLIT

Table 12

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

- If IntraSubPartitionsSplitType is equal to ISP_NO_SPLIT, NumIntraSubPartitions is set equal to 1.
- Otherwise, if one of the following conditions is true, NumIntraSubPartitions is set equal to 2:
  - cbWidth is equal to 4 and cbHeight is equal to 8,
  - cbWidth is equal to 8 and cbHeight is equal to 4.
- Otherwise, NumIntraSubPartitions is set equal to 4.

The syntax elements **intra_luma_mpm_flag**[x0][y0], **intra_luma_mpm_idx**[x0][y0] and **intra_luma_mpm_remainder**[x0][y0] specify the intra prediction mode for luma samples. The array indices x0, y0 specify the location (x0 , y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When intra_luma_mpm_flag[x0][y0] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted coding unit.

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.

The binarization of **intra_chroma_pred_mode** is provided in Table 13

Value of intra_chroma_pred_mode	cross-component linear model enabled	
	Yes	No
	Bin string	Bin string
7	0	na
4	10	0
5	1110	na
6	1111	na
0	11000	100
1	11001	101
2	11010	110
3	11011	111

Table 13

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 14. 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.

inter_pred_idc	Name of inter_pred_idc	
	(cbWidth + cbHeight) != 8	(cbWidth + cbHeight) == 8
0	PRED_L0	PRED_L0
1	PRED_L1	PRED_L1
2	PRED_BI	n.a.

Table 14

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

The binarization of **inter_pred_idc** is provided in Table 15

Value of inter_pred_idc	Name of inter_pred_idc	Bin string	
		(cbWidth + cbHeight) != 8	(cbWidth + cbHeight) == 8
0	PRED_L0	00	0
1	PRED_L1	01	1
2	PRED_BI	1	-

Table 15

sym_mvd_flag[x0][y0] specifies that the syntax elements **ref_idx_l0**[x0][y0] and **ref_idx_l1**[x0][y0], and the mvd_coding() syntax structure are not presented.

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.
It is a requirement of bitstream conformance that when the reference picture corresponding to ref_idx_l0[x0][y0] for the current coding unit is the current decoded picture, inter_pred_idc[x0][y0] shall 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 tile group, affine model based motion compensation is used to generate the prediction samples of the current coding unit. inter_affine_flag[x0][y0] equal to 0 specifies that the coding unit is not predicted by affine model based motion compensation. When inter_affine_flag[x0][y0] is not present, it is inferred to be equal to 0.

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

MotionModelIdc[x][y] represents motion model of a coding unit as illustrated in Table 16. The array indices x, y specify the luma sample location (x, y) relative to the top-left luma sample of the picture.
The variable MotionModelIdc[x][y] is derived as follows for x = x0..x0 + cbWidth − 1 and
y = y0..y0 + cbHeight − 1:

- If merge_flag[x0][y0] is equal to 1, the following applies:

  MotionModelIdc[x][y] = merge_subblock_flag[x0][y0]

- 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]

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

Table 16

amvr_flag[x0][y0] specifies the resolution of motion vector difference. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. amvr_flag[x0][y0] equal to 0 specifies that the resolution of the motion vector difference is 1/4 of a luma sample. amvr_flag[x0][y0] equal to 1 specifies that the resolution of the motion vector difference is further specified by amvr_4pel_flag[x0][y0].
When amvr_flag[x0][y0] is not present, it is inferred as follows:

- If sps_cpr_enabled_flag is equal to 1, amvr_flag[x0][y0] is inferred to be equal to 1.
- Otherwise (sps_cpr_enabled_flag is equal to 0), amvr_flag[x0][y0] is inferred to be equal to 0.

amvr_4pel_flag[x0][y0] equal to 1 specifies that the resolution of the motion vector difference is four 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 amvr_4pel_flag[x0][y0] is not present, it is inferred to be equal to 0.
The variable MvShift is set equal to (amvr_flag[x0][y0] + amvr_4pel_flag[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)

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

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

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

gbi_idx[x0][y0] specifies the weight index of bi-prediction with CU weights. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.
When gbi_idx[x0][y0] is not present, it is inferred to be equal to 0.

cu_cbf equal to 1 specifies that the transform_tree( ) syntax structure is present for the current coding unit. cu_cbf equal to 0 specifies that the transform_tree( ) syntax structure is not present for the current coding unit.
When cu_cbf is not present, it is inferred as follows:

- If cu_skip_flag[x0][y0] is equal to 1, cu_cbf is inferred to be equal to 0.
- Otherwise, cu_cbf is inferred to be equal to 1.

cu_sbt_flag[x0][y0] equal to 1 specifies that for the current coding unit, sub-block transform is used. cu_sbt_flag[x0][y0] equal to 0 specifies that for the current coding unit, sub-block transform is not used.
When cu_sbt_flag[x0][y0] is not present, its value is inferred to be equal to 0.
NOTE – : When sub-block transform is used, a coding unit is tiled into two transform units; one transform unit has residual, the other does not have residual.

cu_sbt_quad_flag[x0][y0] equal to 1 specifies that for the current coding unit, the sub-block transform includes a transform unit of 1/4 size of the current coding unit. cu_sbt_quad_flag[x0][y0] equal to 0 specifies that for the current coding unit the sub-block transform includes a transform unit of 1/2 size of the current coding unit.
When cu_sbt_quad_flag[x0][y0] is not present, its value is inferred to be equal to 0.

cu_sbt_horizontal_flag[x0][y0] equal to 1 specifies that the current coding unit is tiled into 2 transform units by a horizontal split. cu_sbt_horizontal_flag[x0][y0] equal to 0 specifies that the current coding unit is tiled into 2 transform units by a vertical split.
When cu_sbt_horizontal_flag[x0][y0] is not present, its value is derived as follows:

- If cu_sbt_quad_flag[x0][y0] is equal to 1, cu_sbt_horizontal_flag[x0][y0] is set to be equal to allowSbtHoriQuad.
- Otherwise (cu_sbt_quad_flag[x0][y0] is equal to 0), cu_sbt_horizontal_flag[x0][y0] is set to be equal to allowSbtHoriHalf.

cu_sbt_pos_flag[x0][y0] equal to 1 specifies that the tu_cbf_luma, tu_cbf_cb and tu_cbf_cr of the first transform unit in the current coding unit are not present in the bitstream. cu_sbt_pos_flag[x0][y0] equal to 0 specifies that the tu_cbf_luma, tu_cbf_cb and tu_cbf_cr of the second transform unit in the current coding unit are not present in the bitstream.

[0074] Table 17 illustrates the PCM sample syntax in JVET-M1001.

pcm_sample(cbWidth, cbHeight, treeType) {	**Descriptor**	**Binarization**
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)	
}		
}		

Table 17

[0075] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 17.

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

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

[0076] Table 18 illustrates the merge data syntax in JVET-M1001.

merge_data(x0, y0, cbWidth, cbHeight) {	**Descriptor**	**Binarization**
mmvd_flag[x0][y0]	ae(v)	FL; cMax = 1
if(mmvd_flag[x0][y0] = = 1) {		
mmvd_merge_flag[x0][y0]	ae(v)	FL; cMax = 1
mmvd_distance_idx[x0][y0]	ae(v)	TR; cMax = 7, cRiceParam = 0
mmvd_direction_idx[x0][y0]	ae(v)	FL; cMax = 3
} else {		
if(MaxNumSubblockMergeCand > 0 && cbWidth >= 8 && cbHeight >= 8)		
merge_subblock_flag[x0][y0]	ae(v)	FL; cMax = 1
if(merge_subblock_flag[x0][y0] = = 1) {		
if(MaxNumSubblockMergeCand > 1)		
merge_subblock_idx[x0][y0]	ae(v)	TR; cMax = MaxNumSubblockMergeCand -1, cRiceParam = 0
} else {		
if(sps_ciip_enabled_flag && cu_skip_flag[x0][y0] = = 0 && (cbWidth * cbHeight) >= 64 && cbWidth < 128 && cbHeight < 128) {		
ciip_flag[x0][y0]	ae(v)	FL; cMax = 1
if(ciip_flag[x0][y0]) {		
if (cbWidth <= 2 * cbHeight || cbHeight <= 2 * cbWidth)		
ciip_luma_mpm_flag[x0][y0]	ae(v)	FL; cMax = 1
if(ciip_luma_mpm_flag[x0][y0])		
ciip_luma_mpm_idx[x0][y0]	ae(v)	TR; cMax = 2, cRiceParam = 0
}		
}		

if(sps_triangle_enabled_flag && tile_group_type = = B && cbWidth * cbHeight >= 64)		
merge_triangle_flag[x0][y0]	ae(v)	FL; cMax = 1
if(merge_triangle_flag[x0][y0])		
merge_triangle_idx[x0][y0]	ae(v)	EG1
else if(MaxNumMergeCand > 1)		
merge_idx[x0][y0]	ae(v)	TR; cMax = MaxNumMergeCand-1, cRiceParam = 0
}		
}		
}		

Table 18

[0077] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 18.

mmvd_flag[x0][y0] equal to 1 specifies that merge mode with motion vector difference is used to generate the inter prediction parameters of the current coding unit. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

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

mmvd_merge_flag[x0][y0] specifies whether the first (0) or the second (1) candidate in the merging candidate list is used with the motion vector difference derived from mmvd_distance_idx[x0][y0] and mmvd_direction_idx[x0][y0]. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

mmvd_distance_idx[x0][y0] specifies the index used to derive MmvdDistance[x0][y0] as specified in Table 19 The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

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

Table 19

mmvd_direction_idx[x0][y0] specifies index used to derive MmvdSign[x0][y0] as specified in Table 20. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

mmvd_direction_idx[x0][y0]	MmvdSign[x0][y0][0]	MmvdSign[x0][y0][1]
0	+1	0
1	-1	0
2	0	+1
3	0	-1

Table 20

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

MmvdOffset[x0][y0][0] = (MmvdDistance[x0][y0] << 2) * MmvdSign[x0][y0][0]
MmvdOffset[x0][y0][1] = (MmvdDistance[x0][y0] << 2) * MmvdSign[x0][y0][1]

merge_subblock_flag[x0][y0] specifies whether the subblock-based inter prediction parameters for the current coding unit are inferred from neighbouring blocks. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When merge_subblock_flag[x0][y0] is not present, it is inferred to be equal to 0.

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

ciip_flag[x0][y0] specifies whether the combined inter-picture merge and intra-picture prediction is applied for the current coding unit. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.
When ciip_flag[x0][y0] is not present, it is inferred to be equal to 0.

The syntax elements **ciip_luma_mpm_flag**[x0][y0], and **ciip_luma_mpm_idx**[x0][y0] specify the intra prediction mode for luma samples used in combined inter-picture merge and intra-picture prediction. The array indices x0, y0 specify the location (x0 , y0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. The intra prediction mode is derived according to clause 8.4.5.
When ciip_luma_mpm_flag[x0][y0] is not present, it is inferred as follows:

- If cbWidth is greater than 2 * cbHeight or cbHeight is greater than 2 * cbWidth, ciip_luma_mpm_flag[x0][y0] is inferred to be equal to 1.
- Otherwise, ciip_luma_mpm_flag[x0][y0] is inferred to be equal to 0.

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

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

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

- If mmvd_flag[x0][y0] is equal to 1, merge_idx[x0][y0] is inferred to be equal to mmvd_merge_flag[x0][y0].
- Otherwise (mmvd_flag[x0][y0] is equal to 0), merge_idx[x0][y0] is inferred to be equal to 0.

It is a requirement of bitstream conformance that when the reference picture corresponding to ref_idx_l0[x0][y0] for the current coding unit is the current decoded picture, mmvd_flag[x0][y0], merge_subblock_flag[x0][y0], ciip_flag[x0][y0] and merge_triangle_flag[x0][y0] shall all be equal to 0.

[0078] Table 21 illustrates the motion vector difference syntax in JVET-M1001.

mvd_coding(x0, y0, refList ,cpIdx) {	**Descriptor**	**Binarization**
abs_mvd_greater0_flag[0]	ae(v)	FL; cMax = 1
abs_mvd_greater0_flag[1]	ae(v)	FL; cMax = 1
if(abs_mvd_greater0_flag[0])		
abs_mvd_greater1_flag[0]	ae(v)	FL; cMax = 1
if(abs_mvd_greater0_flag[1])		
abs_mvd_greater1_flag[1]	ae(v)	FL; cMax = 1
if(abs_mvd_greater0_flag[0]) {		
if(abs_mvd_greater1_flag[0])		
abs_mvd_minus2[0]	ae(v)	EG1
mvd_sign_flag[0]	ae(v)	FL; cMax = 1
}		
if(abs_mvd_greater0_flag[1]) {		
if(abs_mvd_greater1_flag[1])		
abs_mvd_minus2[1]	ae(v)	EG1
mvd_sign_flag[1]	ae(v)	FL; cMax = 1
}		
}		

Table 21

[0079] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 21.

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])

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.

[0080] Table 22 illustrates the transform tree syntax in JVET-M1001 and Table 23 illustrates the transform unit syntax in JVET-M1001.

transform_tree(x0, y0, tbWidth, tbHeight , treeType) {	Descriptor	Binarization
InferTuCbfLuma = 1		
if(IntraSubPartSplitType = = NO_ISP_SPLIT) {		
if(tbWidth > MaxTbSizeY || tbHeight > MaxTbSizeY) {		
trafoWidth = (tbWidth > MaxTbSizeY) ? (tbWidth / 2) : tbWidth		
trafoHeight = (tbHeight > MaxTbSizeY) ? (tbHeight / 2) : tbHeight		
transform_tree(x0, y0, trafoWidth, trafoHeight)		
if(tbWidth > MaxTbSizeY)		
transform_tree(x0 + trafoWidth, y0, trafoWidth, trafoHeight, treeType)		
if(tbHeight > MaxTbSizeY)		
transform_tree(x0, y0 + trafoHeight, trafoWidth, trafoHeight, treeType)		
if(tbWidth > MaxTbSizeY && tbHeight > MaxTbSizeY)		
transform_tree(x0 + trafoWidth, y0 + trafoHeight, trafoWidth, trafoHeight, treeType)		
} else {		
transform_unit(x0, y0, tbWidth, tbHeight, treeType, 0)		
}		
} else if(IntraSubPartitionsSplitType = = ISP_HOR_SPLIT) {		
trafoHeight = tbHeight / NumIntraSubPartitions		
for(partIdx = 0; partIdx < NumIntraSubPartitions; partIdx++)		
transform_unit(x0, y0 + trafoHeight * partIdx, tbWidth, trafoHeight, treeType, partIdx)		
} else if(IntraSubPartitionsSplitType = = ISP_VER_SPLIT) {		
trafoWidth = tbWidth / NumIntraSubPartitions		
for(partIdx = 0; partIdx < NumIntraSubPartitions; partIdx++)		
transform_unit(x0 + trafoWidth * partIdx, y0, trafoWidth, tbHeight, treeType, partIdx)		
}		
}		

Table 22

transform_unit(x0, y0, tbWidth, tbHeight, treeType) {	Descriptor	Binarization
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA) {		
if(IntraSubPartitionsSplitType = = ISP_NO_SPLIT || (IntraSubPartitionsSplitType != ISP_NO_SPLIT && (subTuIndex < NumIntraSubPartitions − 1 || !InferTuCbfLuma))){		
tu_cbf_luma[x0][y0]	ae(v)	FL; cMax = 1
if (IntraSubPartitionsSplitType != ISP_NO_SPLIT)		
InferTuCbfLuma = InferTuCbfLuma && !tu_cbf_luma[x0][y0]		
}		
if((treeType = = SINGLE_TREE || && subTuIndex = = NumIntraSubPartitions − 1) || treeType = = DUAL_TREE_CHROMA) {		
tu_cbf_cb[x0][y0]	ae(v)	FL; cMax = 1
tu_cbf_cr[x0][y0]	ae(v)	FL; cMax = 1
}		
if(IntraSubPartitionsSplitType != ISP_NO_SPLIT && treeType = = SINGLE_TREE && subTuIndex = = NumIntraSubPartitions − 1))		
xC = CbPosX[x0][y0]		
yC = CbPosY[x0][y0]		
wC = CbWidth[x0][y0] / 2		
hC = CbHeight[x0][y0] / 2		
} else		
xC = x0		
yC = y0		
wC = tbWidth / SubWidthC		
hC = tbHeight / SubHeightC		
}		
if((tu_cbf_luma[x0][y0] || tu_cbf_cb[x0][y0] || tu_cbf_cr[x0][y0]) && treeType != DUAL_TREE_CHROMA) {		

if(cu_qp_delta_enabled_flag && !IsCuQpDeltaCoded) {		
cu_qp_delta_abs	ae(v)	Specified
if(cu_qp_delta_abs)		
cu_qp_delta_sign_flag	ae(v)	FL; cMax = 1
}		
}		
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) && (IntraSubPartitionsSplit[x0][y0] = = ISP_NO_SPLIT))		
tu_mts_flag[x0][y0]	ae(v)	FL; cMax = 1
if(tu_cbf_luma[x0][y0])		
residual_coding(x0, y0, Log2(tbWidth), Log2(tbHeight), 0)		
if(tu_cbf_cb[x0][y0])		
residual_coding(xC, yC, Log2(wC), Log2(hC), 1)		
if(tu_cbf_cr[x0][y0])		
residual_coding(xC, yC, Log2(wC), Log2(hC), 2)		
}		

Table 23

[0081] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 23.

transform_unit(x0, y0, tbWidth, tbHeight, treeType) {	Descriptor	Binarization
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_LUMA)		
tu_cbf_luma[x0][y0]	ae(v)	FL; cMax = 1
if(treeType = = SINGLE_TREE || treeType = = DUAL_TREE_CHROMA) {		
tu_cbf_cb[x0][y0]	ae(v)	FL; cMax = 1
tu_cbf_cr[x0][y0]	ae(v)	FL; cMax = 1
}		
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)	FL; cMax = 1
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)		
}		

Table 23A

[0082] JVET-L1001 provides the following definitions for the respective syntax elements illustrated in Table 23A.

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.as follows:

– If IntraSubPartitionsSplitType is equal to ISP_NO_SPLIT, tu_cbf_luma[x0][y0] is inferred to be equal to 0.

– Otherwise, tu_cbf_luma[x0][y0] is inferred to be equal to 1.

tu_cbf_cb[x0][y0] equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location (x0, y0) of the considered transform block.

When tu_cbf_cb[x0][y0] is not present, 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

CuQpDeltaVal = cu_qp_delta_abs * (1 − 2 * cu_qp_delta_sign_flag)

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.

[0083] Table 24 illustrates the residual coding syntax in JVET-M1001.

residual_coding(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	Descriptor	Binarization
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)	FL; cMax = 1
if(log2TbWidth > 0)		
last_sig_coeff_x_prefix	ae(v)	TR; cMax = (log2TbWidth << 1) − 1, cRiceParam = 0
if(log2TbHeight > 0)		
last_sig_coeff_y_prefix	ae(v)	TR; cMax = (log2TbWidth << 1) − 1, cRiceParam = 0
if(last_sig_coeff_x_prefix > 3)		
last_sig_coeff_x_suffix	ae(v)	FL; cMax = (1 << ((last_sig_coeff_x_prefix >> 1) − 1) − 1)
if(last_sig_coeff_y_prefix > 3)		
last_sig_coeff_y_suffix	ae(v)	FL; cMax = (1 << ((last_sig_coeff_x_prefix >> 1) − 1) − 1)
log2SbW = (Min(log2TbWidth, log2TbHeight) < 2 ? 1 : 2)		
log2SbH = log2SbW		
if (log2TbWidth < 2 && cIdx == 0) {		
log2SbW = log2TbWidth		
log2SbH = 4 − log2SbW		
} else if (log2TbHeight < 2 && cIdx == 0) {		
log2SbH = log2TbHeight		
log2SbW = 4 − log2SbH		
}		
numSbCoeff = 1 << (log2SbW + log2SbH)		
lastScanPos = numSbCoeff		
lastSubBlock = (1 << (log2TbWidth + log2TbHeight − (log2SbW + log2SbH))) − 1		
do {		
if(lastScanPos == 0) {		
lastScanPos = numSbCoeff		

lastSubBlock− −		
}		
lastScanPos− −		
xS = DiagScanOrder[log2TbWidth − log2SbW][log2TbHeight − log2 SbH] [lastSubBlock][0]		
yS = DiagScanOrder[log2TbWidth − log2SbW][log2TbHeight − log2 SbH] [lastSubBlock][1]		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][lastScanPos][0]		
yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][lastScanPos][1]		
} while((xC != LastSignificantCoeffX) || (yC != LastSignificantCoeffY))		
numSigCoeff = 0		
QState = 0		
for(i = lastSubBlock; i >= 0; i− −) {		
startQStateSb = QState		
xS = DiagScanOrder[log2TbWidth − log2SbW][log2TbHeight − log2 SbH] [lastSubBlock][0]		
yS = DiagScanOrder[log2TbWidth − log2SbW][log2TbHeight − log2 SbH] [lastSubBlock][1]		
inferSbDcSigCoeffFlag = 0		
if((i < lastSubBlock) && (i > 0)) {		
coded_sub_block_flag[xS][yS]	ae(v)	FL; cMax = 1
inferSbDcSigCoeffFlag = 1		
}		
firstSigScanPosSb = numSbCoeff		
lastSigScanPosSb = −1		
remBinsPass1 = ((log2SbW + log2SbH) < 4 ? 6 : 28)		
remBinsPass2 = ((log2SbW + log2SbH) < 4 ? 2 : 4)		
firstPosMode0 = (i == lastSubBlock ? lastScanPos − 1 : numSbCoeff − 1)		
firstPosMode1 = −1		
firstPosMode2 = −1		
for(n = (i == firstPosMode0; n >= 0 && remBinsPass1 >= 3; n− −) {		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		

yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][n][1]		
if(coded_sub_block_flag[xS][yS] && (n > 0 || !inferSbDcSigCoeffFlag)) {		
sig_coeff_flag[xC][yC]	ae(v)	FL; cMax = 1
remBinsPass1−−		
if(sig_coeff_flag[xC][yC])		
inferSbDcSigCoeffFlag = 0		
}		
if(sig_coeff_flag[xC][yC]) {		
numSigCoeff++		
abs_level_gt1_flag[n]	ae(v)	FL; cMax = 1
remBinsPass1−−		
if(abs_level_gt1_flag[n]) {		
par_level_flag[n]	ae(v)	FL; cMax = 1
remBinsPass1−−		
if(remBinsPass2 > 0) {		
remBinsPass2−−		
if(remBinsPass2 == 0)		
firstPosMode1 = n − 1		
}		
}		
if(lastSigScanPosSb == −1)		
lastSigScanPosSb = n		
firstSigScanPosSb = n		
}		
AbsLevelPass1[xC][yC] = sig_coeff_flag[xC][yC] + par_level_flag[n] + abs_level_gt1_flag[n]		
if(dep_quant_enabled_flag)		
QState = QStateTransTable[QState][AbsLevelPass1[xC][yC] & 1]		
if(remBinsPass1 < 3)		
firstPosMode2 = n − 1		
}		
if(firstPosMode1 < firstPosMode2)		
firstPosMode1 = firstPosMode2		
for(n = numSbCoeff − 1; n >= firstPosMode2; n−−)		
if(abs_level_gt1_flag[n])		
abs_level_gt3_flag[n]	ae(v)	FL; cMax = 1
for(n = numSbCoeff − 1; n >= firstPosMode1; n−−) {		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		

yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][n][1]		
if(abs_level_gt3_flag[n])		
abs_remainder[n]	ae(v)	Specified
AbsLevel[xC][yC] = AbsLevelPass1[xC][yC] + 2 * (abs_level_gt3_flag[n] + abs_remainder[n])		
}		
for(n = firstPosMode1; n > firstPosMode2; n− −) {		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		
yC = (yS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][1]		
if(abs_level_gt1_flag[n])		
abs_remainder[n]	ae(v)	Specified
AbsLevel[xC][yC] = AbsLevelPass1[xC][yC] + 2 * abs_remainder[n]		
}		
for(n = firstPosMode2; n >= 0; n− −) {		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		
yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][n][1]		
dec_abs_level[n]	ae(v)	Specified
if(AbsLevel[xC][yC] > 0)		
firstSigScanPosSb = n		
if(dep_quant_enabled_flag)		
QState = QStateTransTable[QState][AbsLevel[xC][yC] & 1]		
}		
if(dep_quant_enabled_flag || !sign_data_hiding_enabled_flag)		
signHidden = 0		
else		
signHidden = (lastSigScanPosSb − firstSigScanPosSb > 3 ? 1 : 0)		
for(n = numSbCoeff − 1; n >= 0; n− −) {		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		
yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][n][1]		
if(sig_coeff_flag[xC][yC] && (!signHidden || (n != firstSigScanPosSb)))		
coeff_sign_flag[n]	ae(v)	FL; cMax = 1
}		
if(dep_quant_enabled_flag) {		

QState = startQStateSb		
for(n = numSbCoeff − 1; n >= 0; n− −) {		
xC = (xS << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		
yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][n][1]		
if(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 << log2SbW) + DiagScanOrder[log2SbW][log2SbH][n][0]		
yC = (yS << log2SbH) + DiagScanOrder[log2SbW][log2SbH][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)	FL; cMax = 3
}		

Table 24

[0084] JVET-M1001 provides the following definitions for the respective syntax elements illustrated in Table 24.

[0085] Table 24A illustrates the residual coding syntax in JVET-L1001.

residual_coding(x0, y0, log2TbWidth, log2TbHeight, cIdx) {	Descriptor	Binarization
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)	FL; cMax =1
last_sig_coeff_x_prefix	ae(v)	TR; cMax = (log2TrafoSize << 1)-1, cRiceParam = 0
last_sig_coeff_y_prefix	ae(v)	TR; cMax = (log2TrafoSize << 1)-1, cRiceParam = 0
if(last_sig_coeff_x_prefix > 3)		
last_sig_coeff_x_suffix	ae(v)	FL; cMax = (1 << ((last_sig_coeff_x_prefix >> 1)−1) −1)
if(last_sig_coeff_y_prefix > 3)		
last_sig_coeff_y_suffix	ae(v)	FL; cMax = (1 << ((last_sig_coeff_x_prefix >> 1)−1)−1)
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)	FL; cMax =1
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)	FL; cMax =1
if(sig_coeff_flag[xC][yC])		
inferSbDcSigCoeffFlag = 0		
}		
if(sig_coeff_flag[xC][yC]) {		
numSigCoeff++		
par_level_flag[n]	ae(v)	FL; cMax =1
rem_abs_gt1_flag[n]	ae(v)	FL; cMax =1
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)	FL; cMax =1
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]		Specified
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)	FL; cMax =1
}		
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)	FL; cMax = 3
}		

Table 24A

[0086] JVET-L1001 provides the following definitions for the respective syntax elements illustrated in Table 24A.

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.
When last_sig_coeff_x_prefix is not present, it is inferred to be 0.

last_sig_coeff_y_prefix specifies the prefix of the row position of the last significant coefficient in scanning order within a transform block. The values of last_sig_coeff_y_prefix shall be in the range of 0 to (log2TbHeight << 1) − 1, inclusive.
When last_sig_coeff_y_prefix is not present, it is inferred to be 0.

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

The column position of the last significant coefficient in scanning order within a transform block LastSignificantCoeffX is derived as follows:

– If last_sig_coeff_x_suffix is not present, the following applies:

LastSignificantCoeffX = last_sig_coeff_x_prefix

– 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

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

– 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

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.

abs_level_gt1_flag[n] specifies whether the absolute value of the transform coefficient level (at scanning position n) is greater than 1. When abs_level_gt1_flag[n] is not present, it is inferred to be equal to 0.

par_level_flag[n] specifies the parity of the transform coefficient level at scanning position n. When par_level_flag[n] is not present, it is inferred to be equal to 0.

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_level_gt3_flag[n] specifies whether the absolute value of the transform coefficient level (at scanning position n) is greater than 3. When abs_level_gt3_flag[n] is not present, it is inferred to be equal to 0.

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

It is a requirement of bitstream conformance that the value of abs_remainder[n] shall be constrained such that the corresponding value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is in the range of CoeffMin to CoeffMax, inclusive.

The binarization of **abs_remainder** is as follows:

Input to this process is a request for a binarization for the syntax element abs_remainder[n], the colour component cIdx, the current sub-block index i, and the luma location (x0, y0) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the picture) , 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 variables lastAbsRemainder and lastRiceParam are derived as follows:

- If this process is invoked for the first time for the current sub-block index i, lastAbsRemainder and lastRiceParam are both set equal to 0.
- Otherwise (this process is not invoked for the first time for the current sub-block index i), lastAbsRemainder and lastRiceParam are set equal to the values of abs_remainder[n] and cRiceParam, respectively, that have been derived during the last invocation of the binarization process for the syntax element abs_remainder[n] as specified in this clause.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process as follows 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.

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 ]
}
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.

The variable cRiceParam is derived from lastAbsRemainder and lastRiceParam as follows:

cRiceParam = Min(lastRiceParam + ((lastAbsRemainder > (3 * (1 << lastRiceParam))) ? 1 : 0), 3)

The variable cMax is derived from cRiceParam as:

cMax = (cRiceParam == 1 ? 6 : 7) << cRiceParam

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])

- The prefix bin string is specified by invoking the TR binarization process as specified above 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

The suffix bin string is specified by invoking the k-th order EGk binarization process as specified above for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1.

dec_abs_level[n] is an intermediate value that is coded with Golomb-Rice code at the scanning position n. Given ZeroPos[n] that is derived as provided below during the parsing of dec_abs_level[n], the absolute value of a transform coefficient level at location (xC, yC) AbsLevel[xC][yC] is derived using as follows:

- If dec_abs_level[n] is equal to ZeroPos[n], AbsLevel[xC][yC] is set equal to 0.
- Otherwise if dec_abs_level[n] is less than ZeroPos[n], AbsLevel[xC][yC] is set equal to dec_abs_level[n] + 1;
- Otherwise (dec_abs_level[n] is greater than ZeroPos[n]), AbsLevel[xC][yC] is set equal to dec_abs_level[n].

It is a requirement of bitstream conformance that the value of dec_abs_level[n] shall be constrained such that the corresponding value of TransCoeffLevel[x0][y0][cIdx][xC][yC] is in the range of CoeffMin to CoeffMax, inclusive.

ZeroPos[n] that is derived as follows:

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][y] for the transform block with component index cIdx and the top-left luma location (x0, y0), the variable locSumAbs is derived as specified by the following pseudo code:

```
locSumAbs = 0
if( xC < (1 << log2TbWidth) − 1 ) {
    locSumAbs += AbsLevel[ xC + 1 ][ yC ] − 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 ]
}
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 ]
}
if( locSumAbs > 31 )
    locSumAbs = 31
```

The variable s is set equal to Max(0, QState − 1).
Given the variables locSumAbs and s, the Rice parameter cRiceParam and the variable ZeroPos[n]0 are derived as specified in Table 25.

	locSumAbs															
	cRiceParam	0	0	0	0	0	0	0	1	1	1	1	1	1	1	2	2
	ZeroPos[n]	0	0	0	0	0	1	2	2	2	2	2	2	4	4	4	4
	ZeroPos[n]	1	1	1	1	2	3	4	4	4	6	6	6	8	8	8	8
	ZeroPos[n]	1	1	2	2	2	3	4	4	4	6	6	6	8	8	8	8
	locSumAbs																
	cRiceParam	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3
	ZeroPos[n]	4	4	4	4	4	4	4	8	8	8	8	8	16	16	16	16
	ZeroPos[n]	4	4	12	12	12	12	12	12	12	12	16	16	16	16	16	16
	ZeroPos[n]	8	8	12	12	12	12	12	12	12	16	16	16	16	16	16	16

Table 25

The binarization of **dec_abs_level** is as follows:

Input to this process is a request for a binarization of the syntax element dec_abs_level[n], the colour component cIdx, the luma location (x0, y0) specifying the top-left sample of the current transform block relative to the top-left luma sample of the picture, the current coefficient scan location (xC, yC), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the binarization of the syntax element.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process for dec_abs_level[] as specified above 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

The binarization of dec_abs_level[n] is a concatenation of a prefix bin string and (when present) a suffix bin string.
For the derivation of the prefix bin string, the following applies:

– The prefix value of dec_abs_level[n], prefixVal, is derived as follows:

prefixVal = Min(cMax, dec_abs_level[n])

– The prefix bin string is specified by invoking the TR binarization process as specified above 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 dec_abs_level[n], suffixVal, is derived as follows:

suffixVal = dec_abs_level[n] − cMax

– The suffix bin string is specified by invoking the k-th order EGk binarization process as specified above for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1

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.

[0087] As described above, JVET-M1001 uses an 8-bit variable, initValue to determine an initial most probable state (MPS) value for the bin and a probability value of the bin being the least probably state (LPS). With respect to the each of the syntax elements described above with respect to Tables 2-24, Table 26 provides for each bin of the respective syntax element (indicated by binIdx) whether the bin is: bypass coded (i.e., arithmetic code according to equal probability states); or specifies a ctxInc value for the bin. The ctxInc is used in conjunction with Table 28 and a respective tables to determine an initValue for a bin. It should be noted that in Table 26 when more than one value is provided ctxInc in Table 26, the assignment process is further specified.

Syntax Element	binIdx					
	0	1	2	3	4	>=5
alf_ctb_flag	0..8	na	na	na	na	na
sao_merge_left_flag	0	na	na	na	na	na
sao_merge_up_flag	0	na	na	na	na	na
sao_type_idx_luma	0	bypass	na	na	na	na
sao_type_idx_chroma	0	bypass	na	na	na	na
sao_offset_abs	bypass	bypass	bypass	bypass	bypass	na
sao_offset_sign	bypass	na	na	na	na	na
sao_band_position	bypass	bypass	bypass	bypass	bypass	bypass
sao_eo_class_luma	bypass	bypass	na	na	na	na
sao_eo_class_chroma	bypass	bypass	na	na	na	na
split_qt_flag	0..5	na	na	na	na	na
split_cu_flag	0..8	na	na	na	na	na
mtt_split_cu_vertical_flag	0..4	na	na	na	na	na
mtt_split_cu_binary_flag	0..3 (2 * mtt_split_cu_vertical_flag) + (mttDepth < = 1 ? 1 0)	na	na	na	na	na
cu_skip_flag	0,1,2	na	na	na	na	na
pred_mode_flag	0,1	na	na	na	na	na
pred_mode_cpr_flag	0	na	na	na	na	na
pcm_flag	terminate	na	na	na	na	na
intra_luma_ref_idx[ ][ ]	0	1	2	na	na	na
intra_subpartitions_mode_flag	0	na	na	na	na	na
intra_subpartition_split_flag	0	na	na	na	na	na
intra_luma_mpm_flag	0	na	na	na	na	na
intra_luma_mpm_idx	bypass	bypass	bypass	bypass	bypass	na
intra_luma_mpm_remainder	bypass	bypass	bypass	bypass	bypass	bypass
intra_chroma_pred_mode sps_cclm_enabled_flag = = 0	0	bypass	bypass	na	na	na
intra_chroma_pred_mode[ ][ ]	0	1	2	bypass	bypass	na

sps_cclm_enabled_flag = = 1 && bin at binIdx equal to 2 = = 0						
intra_chroma_pred_mode[ ][ ] sps_cclm_enabled_flag = = 1 && bin at binIdx equal to 2 = = 1	0	1	2	2	na	na
merge_subblock_flag[ ][ ]	0	na	na	na	na	na
merge_flag	0	na	na	na	na	na
mmvd_flag	0	na	na	na	na	na
mmvd_merge_flag	0	na	na	na	na	na
mmvd_distance_idx	0	bypass	bypass	bypass	bypass	bypass
mmvd_direction_idx	bypass	bypass	na	na	na	na
merge_triangle_flag	0,1,2	na	na	na	na	na
merge_triangle_idx	0	bypass	bypass	bypass	bypass	bypass
merge_idx	0	bypass	bypass	bypass	bypass	na
ciip_flag	0	na	na	na	na	na
ciip_luma_mpm_flag	0	na	na	na	na	na
ciip_luma_mpm_idx	bypass	bypass	na	na	na	na
inter_pred_idc	(cbWidth + cbHeight) != 8 ? 7 – ((1 + Log2(cbWidth) + Log2(cbHeight)) >> 1) : 4	4	na	na	na	na
inter_affine_flag	0..2	na	na	na	na	na
cu_affine_type_flag	0	na	na	na	na	na
sym_mvd_flag	0	na	na	na	na	na
ref_idx_l0	0	1	bypass	bypass	bypass	bypass
ref_idx_l1	0	1	bypass	bypass	bypass	bypass
mvp_l0_flag	0	na	na	na	na	na
mvp_l1_flag	0	na	na	na	na	na
amvr_flag (inter_affine_flag[x0][y0] is equal to 0)	0,1,2	na	na	na	na	na
amvr_flag	3	na	na	na	na	na

(inter_affine_flag[x0][y0] is equal to 0)						
amvr_4pel_flag	inter_affine_flag[x0][y0] == 0 ? 0 : 1	na	na	na	na	na
gbi_idx NoBackwardPredFlag = = 0	0	1	na	na	na	na
gbi_idx NoBackwardPredFlag = = 1	0	1	2	3	na	na
cu_cbf	0	na	na	na	na	na
cu_sbt_flag[ ][ ]	0	na	na	na	na	na
cu_sbt_quad_flag[ ][ ]	0	na	na	na	na	na
cu_sbt_horizontal_flag[ ][ ]	(cbWidth = = cbHeight) ? 0 : ((cbWidth < cbHeight) ? 1 : 2)	na	na	na	na	na
cu_sbt_pos_flag[ ][ ]	0	na	na	na	na	na
abs_mvd_greater0_flag	0	na	na	na	na	na
abs_mvd_greater1_flag	0	na	na	na	na	na
abs_mvd_minus2	bypass	bypass	bypass	bypass	bypass	bypass
mvd_sign_flag	bypass	na	na	na	na	na
tu_cbf_luma	0,1,2,3	na	na	na	na	na
tu_cbf_cb	trafoDepth==0 ? 0 : 1	na	na	na	na	na
tu_cbf_cr	tu_cbf_cb	na	na	na	na	na
cu_qp_delta_abs	0	1	1	1	1	bypass
cu_qp_delta_sign_flag	bypass	na	na	na	na	na
last_sig_coeff_x_prefix	0..22					
last_sig_coeff_y_prefix	0..22					
last_sig_coeff_x_suffix	bypass	bypass	bypass	bypass	bypass	bypass
last_sig_coeff_y_suffix	bypass	bypass	bypass	bypass	bypass	bypass
coded_sub_block_flag	0..3	na	na	na	na	na
sig_coeff_flag	0..89	na	na	na	na	na
par_level_flag	0..31	na	na	na	na	na
abs_level_gt1_flag	0..31	na	na	na	na	na
abs_level_gt3_flag	0..31	na	na	na	na	na
abs_remainder	bypass	bypass	bypass	bypass	bypass	bypass
dec_abs_level	bypass	bypass	bypass	bypass	bypass	bypass
coeff_sign_flag	bypass	na	na	na	na	na

tu_mts_idx[ ][ ] MaxMtsIdx == 1	0	na	na	na	na	na
tu_mts_idx[ ][ ] MaxMtsIdx == 4	1+cqtDepth	7	8	9	na	na
tu_mts_idx[ ][ ] MaxMtsIdx == 5	0	1+cqtDepth	7	8	9	na

Table 26

[0088] As described above, JVET-L1001 uses an 8-bit variable, initValue to determine an initial most probable state (MPS) value for the bin and a probability value of the bin being the least probably state (LPS). With respect to the each of the syntax elements described above with respect to Tables 3A etc., Table 26A provides for each bin of the respective syntax element (indicated by binIdx) whether the bin is: bypass coded (i.e., arithmetic code according to equal probability states); or specifies a ctxInc value for the bin. The ctxInc is used in conjunction with Table 28A and respective Tables 24C-63C to determine an initValue for a bin. It should be noted that in Table 26A when more than one value is provided ctxInc in Table 26A, the assignment process is further specified.

Syntax Element	binIdx					
	0	1	2	3	4	>=5
alf_ctb_flag	0..8	na	na	na	na	na
sao_merge_left_flag	0	na	na	na	na	na
sao_merge_up_flag	0	na	na	na	na	na
sao_type_idx_luma	0	bypass	na	na	na	na
sao_type_idx_chroma	0	bypass	na	na	na	na
sao_offset_abs	bypass	bypass	bypass	bypass	bypass	na
sao_offset_sign	bypass	na	na	na	na	na
sao_band_position	bypass	bypass	bypass	bypass	bypass	bypass
sao_eo_class_luma	bypass	bypass	na	na	na	na
sao_eo_class_chroma	bypass	bypass	na	na	na	na
qt_split_cu_flag	0..5	na	na	na	na	na
mtt_split_cu_flag	0..12	0..1	0..1	na	na	na
mtt_split_cu_vertical_flag	(cbWidth= = cbHeight) ? 0 : ((cbWidth > cbHeight) ? 1 : 2)	na	na	na	na	na
mtt_split_cu_binary_flag	0	na	na	na	na	na
cu_skip_flag	0,1,2	na	na	na	na	na
pred_mode_flag	0	na	na	na	na	na
intra_luma_mpm_flag	0	na	na	na	na	na
intra_luma_mpm_idx	bypass	bypass	bypass	bypass	bypass	na
intra_luma_mpm_remainder	bypass	bypass	bypass	bypass	bypass	bypass
intra_chroma_pred_mode	0	bypass	bypass	na	na	na
mvp_l0_flag	0	na	na	na	na	na
mvp_l1_flag	0	na	na	na	na	na
merge_flag	0	na	na	na	na	na
merge_idx	0	bypass	bypass	bypass	na	na
inter_pred_idc	(nPbW + nPbH) != 12 ? CtDepth[x0][y0] : 4	4	na	na	na	na
ref_idx_l0	0	1	bypass	bypass	bypass	bypass
ref_idx_l1	0	1	bypass	bypass	bypass	bypass
merge_affine_flag	0..2	na	na	na	na	na
inter_affine_flag	0..2	na	na	na	na	na
cu_affine_type_flag	0	na	na	na	na	na
amvr_mode	0..2	3	na	na	na	na
cu_cbf	0	na	na	na	na	na
abs_mvd_greater0_flag	0	na	na	na	na	na
abs_mvd_greater1_flag	0	na	na	na	na	na
abs_mvd_minus2	bypass	bypass	bypass	bypass	bypass	bypass
mvd_sign_flag	bypass	na	na	na	na	na
tu_cbf_luma	0..1	na	na	na	na	na
tu_cbf_cb	trafoDepth	na	na	na	na	na
tu_cbf_cr	trafoDepth	na	na	na	na	na
tu_mts_flag	trafoDepth	na	na	na	na	na
transform_skip_flag	0	na	na	na	na	na
last_sig_coeff_x_prefix	0..23					
last_sig_coeff_y_prefix	0..23					

last_sig_coeff_x_suffix	bypass	bypass	bypass	bypass	bypass	bypass
last_sig_coeff_y_suffix	bypass	bypass	bypass	bypass	bypass	bypass
coded_sub_block_flag	0..3	na	na	na	na	na
sig_coeff_flag	0..89	na	na	na	na	na
par_level_flag	0..31	na	na	na	na	na
rem_abs_gt1_flag	0..31	na	na	na	na	na
rem_abs_gt2_flag	0..31	na	na	na	na	na
abs_remainder	bypass	bypass	bypass	bypass	bypass	bypass
coeff_sign_flag	bypass	na	na	na	na	na
mts_idx	0,2	1,3	na	na	na	na

Table 26A

[0089] With respect to alf_ctb_flag in Table 19, the derivation of ctxInc may be as provided "Specification draft for Adaptive Loop Filter," 11th Meeting of ISO/IEC JTC1/SC29/WG11 10-18 July 2018, Ljubljana, SI, document JVET-K0564-v1, which is provided in Table 26B.

Syntax element	**condL**	**condA**	**ctxIdxInc**
alf_ctb_flag[cIdx][x][y]	alf_ctb_flag[cIdx][xL][y]	alf_ctb_flag[cIdx][x][yA]	condL + condA

Table 26B

[0090] With respect to alf_ctb_flag, split_qt_flag, split_cu_flag, cu_skip_flag, amvr_flag, merge_subblock_flag, merge_triangle_flag, and inter_affine_flag the derivation of ctxInc may be as follows:

Input to this process is the luma location (x0, y0) specifying the top-left luma sample of the current luma block relative to the top-left sample of the current picture, the colour component cIdx, the current coding quadtree depth cqDepth, and the width and the height of the current coding block in luma samples cbWidth and cbHeight.

Output of this process is ctxInc.

The location (xNbL, yNbL) is set equal to (x0 − 1, y0) and the variable availableL, specifying the availability of the block located directly to the left of the current block, is derived by invoking an availability derivation process for a block in z-scan order with the location (xCurr, yCurr) set equal to (x0, y0) and the neighbouring location (xNbY, yNbY) set equal to (xNbL, yNbL) as inputs, and the output is assigned to availableL.

The location (xNbA, yNbA) is set equal to (x0, y0 − 1) and the variable availableA specifying the availability of the coding block located directly above the current block, is derived by invoking an availability derivation process for a block in z-scan order with the location (xCurr, yCurr) set equal to (x0, y0) and the neighbouring location (xNbY, yNbY) set equal to (xNbA, yNbA) as inputs, and the output is assigned to availableA.

The variable splitOffset is derived as follows invoking the split avaliability derivation process for a block as specified by above semantics:

splitOffset = (allowSplitBtVer + allowSplitBtHor + allowSplitTtVer + allowSplitTtHor + 2 * allowSplitQt - 1) / 3

mttOffset = (allowSplitBtVer + allowSplitBtHor == allowSplitTtVer + allowSplitTtHor) ? 0 : ((allowSplitBtVer + allowSplitBtHor < allowSplitTtVer + allowSplitTtHor) ? 3 : 4)

The assignment of ctxInc is specified as follows with condL and condA for the syntax elements alf_ctb_flag[x0][y0][cIdx], split_qt_flag[x0][y0], split_cu_flag[x0][y0], pred_mode_flag, cu_skip_flag[x0][y0], amvr_flag[x0][y0], inter_affine_flag[x0][y0], merge_triangle_flag[x0][y0] and merge_subblock_flag[x0][y0] specified in Table 27:

ctxInc = (condL && availableL) + (condA && availableA) + ctxOffset

The variables sizeC, sizeTh2 and sizeTh1 are derived as follows:

sizeTh2 = (MaxBtSizeY = = 128) ? 1024 : ((MaxBtSizeY = = 64) ? 512 : 256)

sizeTh1 = (MaxBtSizeY = = 128) ? 128 : 64

sizeC = cbWidth * cbHeight

Syntax element	condL	condA	ctxOffset
alf_ctb_flag[x0][y0][cIdx]	alf_ctb_flag[xNbL][yNbL][cIdx]	alf_ctb_flag[xNbA][yNbA][cIdx]	3*cIdx
split_qt_flag	cqtDepth[xNbL][yNbL] > cqtDepth	cqtDepth[xNbA][yNbA] > cqtDepth	(cqtDepth < 2) ? 0 : 3
split_cu_flag	cbHeight[xNbL][yNbL] < cbHeight	cbWidth[xNbA][yNbA] < cbWidth	3 * splitOffset
split_cu_flag	((cbHeight / cbHeight[xNbL][yNbL] ! = cbWidth / cbWidth[xNbA][yNbA]) && availableA && mttOffset = = 0) ? 1 : 0	((cbHeight / cbHeight[xNbL][yNbL] < cbWidth / cbWidth[xNbA][yNbA]) && availableL && mttOffset = = 0) ? 1 : 0	mttOffset
cu_skip_flag[x0][y0]	cu_skip_flag[xNbL][yNbL]	cu_skip_flag[xNbA][yNbA]	0
pred_mode_flag	CuPredMode[xNbL][yNbL] == MODE_INTRA ? 1 : 0	CuPredMode[xNbA][yNbA] == MODE_INTRA ? 1 : 0	(condL && availableL && condA && availableA) ? 1 : 0
amvr_flag[x0][y0]	amvr_flag[xNbL][yNbL]	amvr_flag[xNbA][yNbA]	0
merge_subblock_flag[x0][y0]	merge_subblock_flag[xNbL][yNbL] | inter_affine_flag[xNbL][yNbL]	merge_subblock_flag[xNbA][yNbA] | inter_affine_flag[xNbA][yNbA]	0
merge_triangle_flag[x0][y0]	merge_triangle_flag [xNbL][yNbL]	merge_triangle_flag [xNbA][yNbA]	0
inter_affine_flag [x0][y0]	merge_subblock_flag[xNbL][yNbL] | inter_affine_flag[xNbL][yNbL]	merge_subblock_flag[xNbA][yNbA] | inter_affine_flag[xNbA][yNbA]	0

Table 27

Syntax element	condL	condA	ctxSetIdx	ctxInc
qt_split_cu_flag[x0][y0]	cqtDepth[xNbL][yNbL] > cqtDepth	cqtDepth[xNbA][yNbA] > cqtDepth	(cqtDepth < 2) ? 0 : 1	(condL && availableL) + (condA && availableA) + ctxSetIdx * 3
mtt_split_cu_flag [x0][y0]	cbHeight[xNbL][yNbL] < cbHeight	cbWidth[xNbA] [yNbA] < cbWidth	(treeType = = DUAL_TREE_CHROMA) ? 3 : ((sizeC > sizeTh2) ? 0 : ((sizeC > sizeTh1) ? 1 : 2)))	(condL && availableL) + (condA && availableA) + ctxSetIdx * 3
cu_skip_flag[x0][y0]	cu_skip_flag[xNbL][yNbL]	cu_skip_flag[xNbA][yNbA]	na	(condL && availableL) + (condA && availableA)

Table 27A

[0091] With respect to merge_affine_flag, inter_affine_flag, and amvr_mode in Table 19, the derivation of ctxInc may be as provided in Table 27B.

Syntax element	condL	condA	ctxInc
merge_affine_flag[x0] [y0]	merge_affine_flag[xNbL][yNbL]	merge_affine_flag [xNbA][yNbA]	(condL && availableL) + (condA && availableA)
inter_affine_flag[x0][y0]	inter_affine_flag [xNbL][yNbL]	inter_affine_flag [xNbA][yNbA]	(condL && availableL) + (condA && availableA)
amvr_mode[x0][y0]	amvr_mode [xNbL][yNbL]	amvr_mode [xNbA][yNbA]	(condL && availableL) + (condA && availableA)

Table 27B

With respect to tu_cbf_luma in Table 19, the derivation of ctxInc may be as follows:

If trafoDepth is 0

ctxInc = 1

Otherwise

ctxInc = 0

[0092] With respect to last_sig_coeff_x_prefix and last_sig_coeff_y_prefix in Table 26, the derivation of ctxInc may be as follows:

Inputs to this process are the variable binIdx, the colour component index cIdx, the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight.
Output of this process is the variable ctxInc.
The variable log2TbSize is derived as follows:

- If the syntax element to be parsed is last_sig_coeff_x_prefix, log2TbSize is set equal to log2TbWidth.
- Otherwise (the syntax element to be parsed is last_sig_coeff_y_prefix), log2TbSize is set equal to log2TbHeight.

The variables ctxOffset and ctxShift are derived as follows:

- If cIdx is equal to 0, ctxOffset is set equal to $3 * (log2TbSize - 2) + ((log2TbSize - 1) >> 2)$ and ctxShift is set equal to $(log2TbSize + 1) >> 2$.
- Otherwise (cIdx is greater than 0), ctxOffset is set equal to 21 and ctxShift is set equal to $Clip3(0, 2, 2^{log2TbSize} >> 3)$.

The variable ctxInc is derived as follows:

ctxInc = (binIdx >> ctxShift) + ctxOffset

[0093] With respect to tu_cbf_luma in Table 26, the derivation of ctxInc may be as follows:

Inputs to this process are the variable binIdx, the colour component index cIdx, the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight and the current transform tree depth trDepth.
Output of this process is the variable ctxInc.
The variable ctxInc is derived as follows:

- If IntraSubpartitionSplitType is equal to ISP_NO_SPLIT or cIdx is not equal to 0, the following applies:

  ctxInc = trDepth = = 0 ? 1 : 0

- Otherwise (IntraSubpartitionSplitType is not equal to ISP_NO_SPLIT and cIdx is equal to 0), the following applies:
  - The variable prevTuCbfY is derived as follows:
    - If the current transform unit is the first one to be parsed in a coding unit, prevTuCbfY is set equal to 0.
    - Otherwise, prevTuCbfY is set equal to the value of tu_cbf_luma of the previous luma transform unit in the current coding unit.
  - The variable ctxInc is derived as follows:
    ctxInc = 2 + prevTuCbfY

[0094] With respect to coded_sub_block_flag in Table 26, the derivation of ctxInc may be as follows:

Inputs to this process are the colour component index cIdx, the current sub-block scan location (xS, yS), the previously decoded bins of the syntax element coded_sub_block_flag and the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable csbfCtx is derived using the current location (xS, yS), two previously decoded bins of the syntax element coded_sub_block_flag in scan order, log2TbWidth and log2TbHeight, as follows:

- The variables log2SbWidth and log2SbHeight are dervied as follows:

  log2SbWidth = (Min(log2TbWidth, log2TbHeight) < 2 ? 1 : 2)

  log2SbHeight = log2SbWidth

- The variables log2SbWidth and log2SbHeight are modifed as follows:
  - If log2TbWidth is less than 2 and cIdx is equal to 0, the following applies

    log2SbWidth = log2TbWidth

    log2SbHeight = 4 − log2SbWidth
  - Otherwise, if log2TbHeight is less than 2 and cIdx is equal to 0, the following applies

    log2SbHeight = log2TbHeight

    log2SbWidth = 4 − log2SbHeight
- csbfCtx is initialized with 0 as follows:

  csbfCtx = 0
- When xS is less than (1 << (log2TbWidth − log2SbWidth)) − 1, csbfCtx is modified as follows:

  csbfCtx += coded_sub_block_flag[xS + 1][yS]
- When yS is less than (1 << (log2TbHeight − log2SbHeight)) − 1, csbfCtx is modified as follows:

  csbfCtx += coded_sub_block_flag[xS][yS + 1]

The context index increment ctxInc is derived using the colour component index cIdx and csbfCtx as follows:

- If cIdx is equal to 0, ctxInc is derived as follows:

  ctxInc = Min(csbfCtx, 1)
  - Otherwise (cIdx is greater than 0), ctxInc is derived as follows:
    ctxInc = 2 + Min(csbfCtx, 1)

[0095] With respect to sig_coeff_flag, par_level_flag, abs_level_gt1_flag, and abs_level_gt3_flag in Table 26, the derivation of ctxInc may be based on the variables locNumSig and locSumAbsPass1 which may be derived as follows:

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 ]
        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 ]
        }
}
```

[0096] With respect to sig_coeff_flag, par_level_flag, rem_abs_gt1_flag, and rem_abs_gt2_flag, in Table 26A, the derivation of ctxInc may be as provided in "CE7: Transform coefficient coding with reduced number of regular-coded bins (tests 7.1.3a, 7.1.3b)," 12th Meeting of ISO/IEC JTC1/SC29/WG11 3-12 October 2018, Macao, CN, document JVET-L0274-v4, which for the sake of brevity is not repeated herein.

[0097] With respect to mts_idx, in Table 26A, the derivation of ctxInc may be as follows:

```
If CuPredMode[x][y] is MODE_INTRA
        ctxInc = binIdx
Otherwise
        ctxInc = binIdx + 2
```

[0098] With respect to sig_coeff_flag in Table 26, the derivation of ctxInc may be as follows:

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 as specified above 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))
- 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)

[0099] With respect to par_level_flag, abs_level_gt1_flag, and abs_level_gt3_flag in Table 26, the derivation of ctxInc may be as follows:

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 locNumSig and locSumAbsPass1 is derived by invoking the derivation process specified above 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)
- 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))))
- Otherwise (cIdx is greater than 0), ctxInc is derived as follows:
  ctxInc = 22 + ctxOffset + (d == 0 ? 5 : 0)

[0100] As provided above, for each non-bypass bin in Table 26, a ctxInc value is specified. Table 28 provides a ctxIdxOffest value for each ctxInc specified in Table 26. Where the ctxIdxOffset value is the lowest value specified in an entry for an initType. The derivation of initType depends on the value of the cabac_init_flag syntax element and initType is derived as follows:

```
if( tile_group_type = = I )
        initType = 0
else if( tile_group_type = = P )
        initType = cabac_init_flag ? 2 : 1
else
        initType = cabac_init_flag ? 1 : 2
```

[0101] As provided above, for each non-bypass bin in Table 26A, a ctxInc value is specified. Table 28A provides a ctxIdxOffest value for each ctxInc specified in Table 26A. Where the ctxIdxOffset value is the lowest value specified in an entry for an initType. The derivation of initType depends on the value of the cabac_init_flag syntax element and initType is derived as follows:

```
if( slice_type = = I )
        initType = 0
else if( slice_type = = P )
        initType = cabac_init_flag ? 2 : 1
else
        initType = cabac_init_flag ? 1 : 2
```

[0102] For each for each non-bypass bin in Table 26, the sum of ctxInc derived from Table 26 and ctxIdxOffset derived from Table 28 provide an entry in one of respective at which an initValue is provided.

Syntax Structure	Syntax Element	initType		
		0	1	2
coding_tree_unit()	alf_ctb_flag	0..8	9..17	18..26
sao()	sao_merge_left_flag	0	1	2
	sao_merge_up_flag	0	1	2
	sao_type_idx_luma	0	1	2
	sao_type_idx_chroma	0	1	2
coding_tree()	split_cu_flag	0..12	13..25	26..38
	split_qt_flag	0..5	6..11	12..17
	mtt_split_cu_vertical_flag	0..2	3..5	6..8
	mtt_split_cu_binary_flag	0	1	2
	cu_skip_flag		0..2	3..5
coding_unit()	pred_mode_flag		0	1
	pred_mode_cpr_flag	0	1	2
	intra_luma_ref_idx	0	1	2
	intra_subpartitions_mode_flag	0	1	2
	intra_subpartition_split_flag	0	1	2
	intra_luma_mpm_flag	0	1	2
	intra_chroma_pred_mode	0..2	3..5	6..8
	merge_flag		0	1
	inter_pred_idc		0..4	5..9
	inter_affine_flag		0..2	3..5
	cu_affine_type_flag	0	1	2
	sym_mvd_flag	0	1	2
	ref_idx_l0		0..1	2..3
	ref_idx_l1		0..1	2..3
	mvp_l0_flag	0	1	2
	mvp_l1_flag	0	1	2
	amvr_flag	0..3	4..7	8.11
	amvr_4pel_flag	0..1	2..3	4..5
	gbi_idx		0..3	4..7
	cu_sbt_flag	0	1	2
	cu_sbt_quad_flag	0	1	2
	cu_sbt_horizontal_flag	0..2	3..5	6..8
	cu_sbt_pos_flag	0	1	2
	cu_cbf	0	1	2
	mmvd_flag		0	1
merge_data()	mmvd_merge_flag		0	1
	mmvd_distance_idx		0	1
	merge_subblock_flag	0	1	2
	merge_subblock_idx	0	1	2
	merge_triangle_flag		0..2	3..4
	merge_triangle_idx		0	1
	merge_idx		0	1
	ciip_flag		0	1
	ciip_luma_mpm_flag		0	1
mvd_coding()	abs_mvd_greater0_flag	0	1	2
	abs_mvd_greater1_flag	0	1	2
transform_unit()	tu_cbf_luma	0..1	2..3	4..5
	tu_cbf_cb	0..4	5..9	10..14
	tu_cbf_cr	0..4	5..9	10..14

	cu_qp_delta_abs	0	1	2
residual_coding()	last_sig_coeff_x_prefix	0..22	23..45	46..68
	last_sig_coeff_y_prefix	0..22	23..45	46..68
	coded_sub_block_flag	0..3	4..7	8..11
	sig_coeff_flag	0..89	90..179	180..269
	par_level_flag	0..31	32..63	64..95
	abs_level_gt1_flag	0..31	32..63	64..95
	abs_level_gt3_flag	0..31	32..63	64..95
	mts_idx[ ][ ][ ]	0..9	10..19	20..29

Table 28

[0103] For each for each non-bypass bin in Table 26A, the sum of ctxInc derived from Table 26A and ctxIdxOffest derived from Table 28A provided an entry in one of respective tables 24C-63C at which a initValue is provided.

Syntax Structure	Syntax Element	initType		
		0	1	2
coding_tree_unit()	alf_ctb_flag	0..8	9..17	18..26
sao()	sao_merge_left_flag	0	1	2
	sao_merge_up_flag	0	1	2
	sao_type_idx_luma	0	1	2
	sao_type_idx_chroma	0	1	2
coding_quadtree()	qt_split_cu_flag	0..5	6..11	12..17
multi_type_tree()	mtt_split_cu_flag, bin1, bin2	0..12 13..14 15	16..28 29..30 31	32..44 45..46 47
	mtt_split_cu_vertical_flag	0..2	3..5	6..8
	mtt_split_cu_binary_flag	0	1	2
coding_unit()	cu_skip_flag		0..2	3..5
	pred_mode_flag		0	1
	intra_luma_mpm_flag	0	1	2
	intra_chroma_pred_mode	0	1	2
	mvp_l0_flag		0	1
	mvp_l1_flag		0	1
	merge_flag		0	1
	merge_idx		0	1
	inter_pred_idc		0..4	5..9
	ref_idx_l0		0..1	2..3
	ref_idx_l1		0..1	2..3
	merge_affine_flag		0..2	3..5
	inter_affine_flag		0..2	3..5
	cu_affine_type_flag	0	1	2
	amvr_mode, bin1		0..3	4..7
	cu_cbf	0	1	2
mvd_coding	abs_mvd_greater0_flag	0	1	2
	abs_mvd_greater1_flag	0	1	2
transform_unit	tu_cbf_luma	0..1	2..3	4..5
	tu_cbf_cb	0..4	5..9	10..14
	tu_cbf_cr	0..4	5..9	10..14
	tu_mts_flag	0..5	6..11	12..17
	transform_skip_flag	0..1	2..3	4..5
residual_coding	last_sig_coeff_x_prefix	0..22	23..45	46..68
	last_sig_coeff_y_prefix	0..22	23..45	46..68
	coded_sub_block_flag	0..3	4..7	8..11
	sig_coeff_flag	0..89	90..179	180..269
	par_level_flag	0..31	32..63	64..95
	rem_abs_gt1_flag	0..31	32..63	64..95
	rem_abs_gt2_flag	0..31	32..63	64..95
	mts_idx	0..1	2..3	4..5

Table 28A

[0104] With respect to the context coded bins, Tables 24C-63C provide the value of initValue specified in JVET-L1001 for each context index.

Initialization variable	ctxIdx of alf_ctb_flag								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	100	153	200	100	153	200	100	153	200
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	100	153	200	100	153	200	100	153	200
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
initValue	100	153	200	100	153	200	100	153	200

Table 24C

Initialization variable	ctxIdx of sao_merge_left_flag		
	0	**1**	**2**
initValue	153	153	153

Table 25C

Initialization variable	ctxIdx of sao_merge_up_flag		
	0	**1**	**2**
initValue	153	153	153

Table 26C

Initialization variable	ctxIdx of sao_type_idx_luma		
	0	**1**	**2**
initValue	200	185	160

Table 27C

Initialization variable	ctxIdx of sao_type_idx_chroma		
	0	**1**	**2**
initValue	200	185	160

Table 28C

Initialization variable	ctxIdx of qt_split_cu_flag					
	0	**1**	**2**	**3**	**4**	**5**
initValue	139	141	157	139	141	157
	6	**7**	**8**	**9**	**10**	**11**
initValue	107	139	126	107	139	126
	12	**13**	**14**	**15**	**16**	**17**
initValue	107	139	126	107	139	126

Table 29C

Initialization variable	ctxIdx of mtt_split_cu_flag															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	139	141	157	139	141	157	139	141	157	139	141	157	154	154	154	154
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	107	139	126	107	139	126	107	139	126	107	139	154	154	154	154	154
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	107	139	126	107	139	126	107	139	126	107	139	126	154	154	154	154

Table 30C

Initialization variable	ctxIdx of **mtt_split_cu_vertical_flag**								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	154	154	154	154	154	154	154	154	154

Table 31C

Initialization variable	ctxIdx of **mtt_split_cu_binary_flag**		
	0	**1**	**2**
initValue	154	154	154

Table 32C

Initialization variable	ctxIdx of **cu_skip_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	197	185	201	197	185	201

Table 33C

Initialization variable	ctxIdx of **pred_mode_flag**	
	0	**1**
initValue	149	134

Table 34C

Initialization variable	ctxIdx of **intra_luma_mpm_flag**		
	0	**1**	**2**
initValue	184	154	183

Table 35C

Initialization variable	ctxIdx of **intra_chroma_pred_mode**		
	0	**1**	**2**
initValue	63	152	152

Table 36C

Initialization variable	ctxIdx of **mvp_l0_flag**	
	0	**1**
initValue	168	168

Table 37C

Initialization variable	ctxIdx of **mvp_l1_flag**	
	0	**1**
initValue	168	168

Table 38C

Initialization variable	ctxIdx of **merge_flag**	
	0	**1**
initValue	110	154

Table 39C

Initialization variable	ctxIdx of **merge_idx**	
	0	**1**
initValue	122	137

Table 40C

Initialization variable	ctxIdx of **inter_pred_idc**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
initValue	95	79	63	31	31	95	79	63	31	31

Table 41C

Initialization variable	ctxIdx of **ref_idx_l0**			
	0	**1**	**2**	**3**
initValue	153	153	153	153

Table 42C

Initialization variable	ctxIdx of **ref_idx_l1**			
	0	**1**	**2**	**3**
initValue	153	153	153	153

Table 43C

Initialization variable	ctxIdx of **merge_affine_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	197	185	201	197	185	201

Table 44C

Initialization variable	ctxIdx of **inter_affine_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	197	185	201	197	185	201

Table 45C

Initialization variable	ctxIdx of **cu_affine_type_flag**	
	0	**1**
initValue	77	92

Table 46C

Initialization variable	ctxIdx of **amvr_mode**			
	0	**1**	**2**	**3**
initValue	197	185	201	185
	4	**5**	**6**	**7**
initValue	197	185	201	185

Table 47C

Initialization variable	ctxIdx of **cu_cbf**		
	0	**1**	**2**
initValue	154	79	79

Table 48C

Initialization variable	ctxIdx of **abs_mvd_greater0_flag**		
	0	**1**	**2**
initValue	154	140	169

Table 49C

Initialization variable	ctxIdx of abs_mvd_greater1_flag		
	0	1	2
initValue	154	198	198

Table 50C

Initialization variable	ctxIdx of tu_cbf_luma					
	0	1	2	3	4	5
initValue	111	141	153	111	153	111

Table 51C

Initialization variable	ctxIdx of tu_cbf_cb														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initValue	94	138	182	154	154	149	107	167	154	154	149	92	167	154	154

Table 52C

Initialization variable	ctxIdx of tu_cbf_cr														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initValue	94	138	182	154	154	149	107	167	154	154	149	92	167	154	154

Table 53C

Initialization variable	ctxIdx of tu_mts_flag					
	0	1	2	3	4	5
initValue	154	154	154	154	154	154
	6	7	8	9	10	11
initValue	154	154	154	154	154	154
	12	13	14	15	16	17
initValue	154	154	154	154	154	154

Table 54C

Initialization variable	ctxIdx of transform_skip_flag					
	0	1	2	3	4	5
initValue	139	139	139	139	139	139

Table 55C

Initialization variable	ctxIdx of last_sig_coeff_x_prefix																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
initValue	110	110	124	125	140	153	125	127	140	109	111	143	127	111	79	143	127	111
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
initValue	79	108	123	63	154	125	110	94	110	95	79	125	111	110	78	110	111	111
	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
initValue	95	94	111	111	95	94	108	123	108	154	125	110	124	110	95	94	125	111
	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68			
initValue	111	79	125	126	111	111	79	126	111	111	79	108	123	93	154			

Table 56C

Initialization variable	ctxIdx of last_sig_coeff_y_prefix																	
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	110	110	124	125	140	153	125	127	140	109	111	143	127	111	79	143	127	111
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**
initValue	79	108	123	63	154	125	110	94	110	95	79	125	111	110	78	110	111	111
	36	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**
initValue	95	94	111	111	95	94	108	123	108	154	125	110	124	110	95	94	125	111
	54	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**			
initValue	111	79	125	126	111	111	79	126	111	111	79	108	123	93	154			

Table 57C

Initialization variable	ctxIdx of coded_sub_block_flag											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	91	171	134	141	121	140	61	154	121	140	61	154

Table 58C

Initialization variable	ctxIdx of sig_coeff_flag														
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
initValue	120	138	153	154	140	126	120	139	154	155	155	142	137	185	169
	15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
initValue	185	171	159	119	168	153	140	140	141	167	200	155	172	142	158
	30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**
initValue	108	157	173	173	218	189	123	175	159	175	190	251	79	223	223
	45	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
initValue	223	223	223	152	158	157	187	204	175	170	223	223	237	223	223
	60	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**
initValue	107	143	219	188	233	190	63	250	205	252	220	251	63	223	223
	75	**76**	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**
initValue	223	223	253	166	159	158	232	158	174	183	238	223	223	223	223
	90	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**	**103**	**104**
initValue	149	152	167	168	183	140	149	182	168	183	169	170	195	213	183
	105	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**
initValue	198	184	156	134	168	168	139	169	155	166	229	198	229	185	157
	120	**121**	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**	**132**	**133**	**134**
initValue	123	142	202	172	157	203	138	173	218	188	173	175	168	223	223
	135	**136**	**137**	**138**	**139**	**140**	**141**	**142**	**143**	**144**	**145**	**146**	**147**	**148**	**149**
initValue	223	223	223	182	141	158	186	142	173	183	223	223	223	222	223
	150	**151**	**152**	**153**	**154**	**155**	**156**	**157**	**158**	**159**	**160**	**161**	**162**	**163**	**164**
initValue	123	172	175	158	158	233	138	175	190	175	188	175	196	223	223
	165	**166**	**167**	**168**	**169**	**170**	**171**	**172**	**173**	**174**	**175**	**176**	**177**	**178**	**179**
initValue	223	223	223	167	141	175	143	172	159	182	223	223	223	223	223
	180	**181**	**182**	**183**	**184**	**185**	**186**	**187**	**188**	**189**	**190**	**191**	**192**	**193**	**194**
initValue	120	152	167	153	168	169	119	167	197	183	183	170	209	213	183
	195	**196**	**197**	**198**	**199**	**200**	**201**	**202**	**203**	**204**	**205**	**206**	**207**	**208**	**209**
initValue	183	169	185	148	167	153	139	154	140	166	199	183	184	184	157
	210	**211**	**212**	**213**	**214**	**215**	**216**	**217**	**218**	**219**	**220**	**221**	**222**	**223**	**224**
initValue	152	127	173	201	187	173	226	188	188	217	188	174	182	223	223
	225	**226**	**227**	**228**	**229**	**230**	**231**	**232**	**233**	**234**	**235**	**236**	**237**	**238**	**239**
initValue	223	223	223	196	156	143	158	172	216	168	223	223	223	191	223
	240	**241**	**242**	**243**	**244**	**245**	**246**	**247**	**248**	**249**	**250**	**251**	**252**	**253**	**254**
initValue	137	142	189	173	187	174	241	175	175	174	174	204	210	223	223
	255	**256**	**257**	**258**	**259**	**260**	**261**	**262**	**263**	**264**	**265**	**266**	**267**	**268**	**269**
initValue	223	223	223	167	185	159	158	159	189	196	223	223	223	223	223

Table 59C

Initialization variable	ctxIdx of par_level_flag															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	135	150	152	138	153	153	151	123	153	168	139	152	153	153	139	139
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	138	168	139	154	139	135	121	167	168	138	153	167	139	154	139	154
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	120	119	151	167	138	168	135	152	153	153	139	136	153	153	168	139
	48	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**
initValue	137	168	168	139	139	134	150	152	153	153	153	166	168	168	139	139
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**
initValue	105	119	151	152	153	153	135	152	182	153	168	136	182	153	168	139
	80	**91**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**
initValue	166	168	139	168	154	105	135	152	167	153	124	151	168	169	153	124

Table 60C

Initialization variable	ctxIdx of rem_abs_gt1_flag															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	134	161	149	121	122	138	88	120	107	108	109	105	107	123	109	124
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	151	138	139	154	140	179	72	90	121	122	123	75	76	123	139	170
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	88	0	102	104	150	122	101	89	150	151	138	88	120	122	152	153
	48	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**
initValue	105	107	123	153	154	101	0	105	121	107	93	118	106	108	124	154
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**
initValue	73	0	58	119	150	137	42	73	120	136	123	58	149	151	152	153
	80	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**
initValue	134	136	152	153	125	87	57	90	107	107	63	119	91	152	124	140

Table 61C

Initialization variable	ctxIdx of rem_abs_gt2_flag															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	164	149	137	153	124	125	151	138	139	125	125	152	139	140	140	111
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	153	154	155	170	127	165	121	138	139	139	125	138	154	156	171	127
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	118	0	136	152	153	154	134	152	153	139	140	150	138	139	154	155
	48	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**
initValue	151	153	169	140	200	132	135	152	139	139	125	151	154	155	141	142
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**
initValue	89	103	121	137	138	139	119	137	138	139	125	135	167	168	154	140
	80	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**
initValue	136	153	183	155	185	27	149	137	153	139	125	151	154	170	127	127

Table 62C

Initialization variable	ctxIdx of mtx_idx					
	0	**1**	**2**	**3**	**4**	**5**
initValue	154	154	154	154	154	154

Table 63C

[0105] The values of initValue specified in JVET-L1001 for syntax element may be less than ideal.

[0106] With respect to the context coded bins, the value of initValue specified in JVET-M1001 for each context index may be less than ideal or yet to be defined.

[0107] FIG. 3 is a block diagram illustrating an example of a system that may be configured to code (i.e., encode and/or decode) video data according to one or more techniques of this disclosure. System 100 represents an example of a system that may perform video coding using partitioning techniques described according to one or more techniques of this disclosure. As illustrated in FIG. 3, system 100 includes source device 102, com-

munications medium 110, and destination device 120. In the example illustrated in FIG. 3, source device 102 may include any device configured to encode video data and transmit encoded video data to communications medium 110. Destination device 120 may include any device configured to receive encoded video data via communications medium 110 and to decode encoded video data. Source device 102 and/or destination device 120 may include computing devices equipped for wired and/or wireless communications and may include set top boxes, digital video recorders, televisions, desktop, laptop, or tablet computers, gaming consoles, mobile devices, including, for example, "smart" phones, cellular telephones, personal gaming devices, and medical imagining devices.

[0108] Communications medium 110 may include any combination of wireless and wired communication media, and/or storage devices. Communications medium 110 may include coaxial cables, fiber optic cables, twisted pair cables, wireless transmitters and receivers, routers, switches, repeaters, base stations, or any other equipment that may be useful to facilitate communications between various devices and sites. Communications medium 110 may include one or more networks. For example, communications medium 110 may include a network configured to enable access to the World Wide Web, for example, the Internet. A network may operate according to a combination of one or more telecommunication protocols. Telecommunications protocols may include proprietary aspects and/or may include standardized telecommunication protocols. Examples of standardized telecommunications protocols include Digital Video Broadcasting (DVB) standards, Advanced Television Systems Committee (ATSC) standards, Integrated Services Digital Broadcasting (ISDB) standards, Data Over Cable Service Interface Specification (DOCSIS) standards, Global System Mobile Communications (GSM) standards, code division multiple access (CDMA) standards, 3rd Generation Partnership Project (3GPP) standards, European Telecommunications Standards Institute (ETSI) standards, Internet Protocol (IP) standards, Wireless Application Protocol (WAP) standards, and Institute of Electrical and Electronics Engineers (IEEE) standards.

[0109] Storage devices may include any type of device or storage medium capable of storing data. A storage medium may include a tangible or non-transitory computer-readable media. A computer readable medium may include optical discs, flash memory, magnetic memory, or any other suitable digital storage media. In some examples, a memory device or portions thereof may be described as non-volatile memory and in other examples portions of memory devices may be described as volatile memory. Examples of volatile memories may include random access memories (RAM), dynamic random access memories (DRAM), and static random access memories (SRAM). Examples of non-volatile memories may include magnetic hard discs, optical

discs, floppy discs, flash memories, or forms of electrically programmable memories (EPROM) or electrically erasable and programmable (EEPROM) memories. Storage device(s) may include memory cards (e.g., a Secure Digital (SD) memory card), internal/external hard disk drives, and/or internal/external solid state drives. Data may be stored on a storage device according to a defined file format.

[0110] Referring again to FIG. 3, source device 102 includes video source 104, video encoder 106, and interface 108. Video source 104 may include any device configured to capture and/or store video data. For example, video source 104 may include a video camera and a storage device operably coupled thereto. Video encoder 106 may include any device configured to receive video data and generate a compliant bitstream representing the video data. A compliant bitstream may refer to a bitstream that a video decoder can receive and reproduce video data therefrom. Aspects of a compliant bitstream may be defined according to a video coding standard. When generating a compliant bitstream video encoder 106 may compress video data. Compression may be lossy (discernible or indiscernible) or lossless. Interface 108 may include any device configured to receive a compliant video bitstream and transmit and/or store the compliant video bitstream to a communications medium. Interface 108 may include a network interface card, such as an Ethernet card, and may include an optical transceiver, a radio frequency transceiver, or any other type of device that can send and/or receive information. Further, interface 108 may include a computer system interface that may enable a compliant video bitstream to be stored on a storage device. For example, interface 108 may include a chipset supporting Peripheral Component Interconnect (PCI) and Peripheral Component Interconnect Express (PCIe) bus protocols, proprietary bus protocols, Universal Serial Bus (USB) protocols, I^2C, or any other logical and physical structure that may be used to interconnect peer devices.

[0111] Referring again to FIG. 3, destination device 120 includes interface 122, video decoder 124, and display 126. Interface 122 may include any device configured to receive a compliant video bitstream from a communications medium. Interface 108 may include a network interface card, such as an Ethernet card, and may include an optical transceiver, a radio frequency transceiver, or any other type of device that can receive and/or send information. Further, interface 122 may include a computer system interface enabling a compliant video bitstream to be retrieved from a storage device. For example, interface 122 may include a chipset supporting PCI and PCIe bus protocols, proprietary bus protocols, USB protocols, I^2C, or any other logical and physical structure that may be used to interconnect peer devices. Video decoder 124 may include any device configured to receive a compliant bitstream and/or acceptable variations thereof and reproduce video data therefrom. Display 126 may include any device configured to display video data. Display 126 may comprise one of a variety of

display devices such as a liquid crystal display (LCD), a plasma display, an organic light emitting diode (OLED) display, or another type of display. Display 126 may include a High Definition display or an Ultra High Definition display. It should be noted that although in the example illustrated in FIG. 3, video decoder 124 is described as outputting data to display 126, video decoder 124 may be configured to output video data to various types of devices and/or sub-components thereof. For example, video decoder 124 may be configured to output video data to any communication medium, as described herein.

[0112] FIG. 4 is a block diagram illustrating an example of video encoder 200 that may implement the techniques for encoding video data described herein. It should be noted that although example video encoder 200 is illustrated as having distinct functional blocks, such an illustration is for descriptive purposes and does not limit video encoder 200 and/or sub-components thereof to a particular hardware or software architecture. Functions of video encoder 200 may be realized using any combination of hardware, firmware, and/or software implementations. In one example, video encoder 200 may be configured to encode video data according to the techniques described herein. Video encoder 200 may perform intra prediction coding and inter prediction coding of picture areas, and, as such, may be referred to as a hybrid video encoder. In the example illustrated in FIG. 4, video encoder 200 receives source video blocks. In some examples, source video blocks may include areas of picture that has been divided according to a coding structure. For example, source video data may include macroblocks, CTUs, CBs, sub-divisions thereof, and/or another equivalent coding unit. In some examples, video encoder 200 may be configured to perform additional sub-divisions of source video blocks. It should be noted that some techniques described herein may be generally applicable to video coding, regardless of how source video data is partitioned prior to and/or during encoding. In the example illustrated in FIG. 4, video encoder 200 includes summer 202, transform coefficient generator 204, coefficient quantization unit 206, inverse quantization/transform processing unit 208, summer 210, intra prediction processing unit 212, inter prediction processing unit 214, filter unit 216, and entropy encoding unit 218.

[0113] As illustrated in FIG. 4, video encoder 200 receives source video blocks and outputs a bitstream. Video encoder 200 may generate residual data by subtracting a predictive video block from a source video block. Summer 202 represents a component configured to perform this subtraction operation. In one example, the subtraction of video blocks occurs in the pixel domain. Transform coefficient generator 204 applies a transform, such as a discrete cosine transform (DCT), a discrete sine transform (DST), or a conceptually similar transform, to the residual block or sub-divisions thereof (e.g., four 8x8 transforms may be applied to a 16x16 array of residual values) to produce a

set of residual transform coefficients. Transform coefficient generator 204 may be configured to perform any and all combinations of the transforms included in the family of discrete trigonometric transforms. As described above, in ITU-T H.265, TBs are restricted to the following sizes 4x4, 8x8, 16x16, and 32x32. In one example, transform coefficient generator 204 may be configured to perform transformations according to arrays having sizes of 4x4, 8x8, 16x16, and 32x32. In one example, transform coefficient generator 204 may be further configured to perform transformations according to arrays having other dimensions. In particular, in some cases, it may be useful to perform transformations on rectangular arrays of difference values. In one example, transform coefficient generator 204 may be configured to perform transformations according to the following sizes of arrays: 2x2, 2x4N, 4Mx2, and/or 4Mx4N. In one example, a 2-dimensional (2D) MxN inverse transform may be implemented as 1-dimensional (1D) M-point inverse transform followed by a 1D N-point inverse transform. In one example, a 2D inverse transform may be implemented as a 1D N-point vertical transform followed by a 1D N-point horizontal transform. In one example, a 2D inverse transform may be implemented as a 1D N-point horizontal transform followed by a 1D N-point vertical transform. Transform coefficient generator 204 may output transform coefficients to coefficient quantization unit 206.

[0114] Coefficient quantization unit 206 may be configured to perform quantization of the transform coefficients. As described above, the degree of quantization may be modified by adjusting a quantization parameter. Coefficient quantization unit 206 may be further configured to determine quantization parameters and output QP data (e.g., data used to determine a quantization group size and/or delta QP values) that may be used by a video decoder to reconstruct a quantization parameter to perform inverse quantization during video decoding. It should be noted that in other examples, one or more additional or alternative parameters may be used to determine a level of quantization (e.g., scaling factors). The techniques described herein may be generally applicable to determining a level of quantization for transform coefficients corresponding to a component of video data based on a level of quantization for transform coefficients corresponding another component of video data.

[0115] Referring again to FIG. 4, quantized transform coefficients are output to inverse quantization/transform processing unit 208. Inverse quantization/transform processing unit 208 may be configured to apply an inverse quantization and an inverse transformation to generate reconstructed residual data. As illustrated in FIG. 4, at summer 210, reconstructed residual data may be added to a predictive video block. In this manner, an encoded video block may be reconstructed and the resulting reconstructed video block may be used to evaluate the encoding quality for a given prediction, transformation, and/or quantization. Video encoder 200 may be configured to perform

multiple coding passes (e.g., perform encoding while varying one or more of a prediction, transformation parameters, and quantization parameters). The rate-distortion of a bitstream or other system parameters may be optimized based on evaluation of reconstructed video blocks. Further, reconstructed video blocks may be stored and used as reference for predicting subsequent blocks.

[0116] As described above, a video block may be coded using an intra prediction. Intra prediction processing unit 212 may be configured to select an intra prediction mode for a video block to be coded. Intra prediction processing unit 212 may be configured to evaluate a frame and/or an area thereof and determine an intra prediction mode to use to encode a current block. As illustrated in FIG. 4, intra prediction processing unit 212 outputs intra prediction data (e.g., syntax elements) to entropy encoding unit 218 and transform coefficient generator 204. As described above, a transform performed on residual data may be mode dependent. As described above, possible intra prediction modes may include planar prediction modes, DC prediction modes, and angular prediction modes. Further, in some examples, a prediction for a chroma component may be inferred from an intra prediction for a luma prediction mode. Inter prediction processing unit 214 may be configured to perform inter prediction coding for a current video block. Inter prediction processing unit 214 may be configured to receive source video blocks and calculate a motion vector for PUs of a video block. A motion vector may indicate the displacement of a PU (or similar coding structure) of a video block within a current video frame relative to a predictive block within a reference frame. Inter prediction coding may use one or more reference pictures. Further, motion prediction may be uni-predictive (use one motion vector) or bi-predictive (use two motion vectors). Inter prediction processing unit 214 may be configured to select a predictive block by calculating a pixel difference determined by, for example, sum of absolute difference (SAD), sum of square difference (SSD), or other difference metrics. As described above, a motion vector may be determined and specified according to motion vector prediction. Inter prediction processing unit 214 may be configured to perform motion vector prediction, as described above. Inter prediction processing unit 214 may be configured to generate a predictive block using the motion prediction data. For example, inter prediction processing unit 214 may locate a predictive video block within a frame buffer (not shown in FIG. 4). It should be noted that inter prediction processing unit 214 may further be configured to apply one or more interpolation filters to a reconstructed residual block to calculate sub-integer pixel values for use in motion estimation. Inter prediction processing unit 214 may output motion prediction data for a calculated motion vector to entropy encoding unit 218. As illustrated in FIG. 4, inter prediction processing unit 214 may receive reconstructed video block via filter unit 216. Filter unit 216 may be configured to perform

deblocking and/or Sample Adaptive Offset (SAO) filtering. Deblocking refers to the process of smoothing the boundaries of reconstructed video blocks (e.g., make boundaries less perceptible to a viewer). SAO filtering is a non-linear amplitude mapping that may be used to improve reconstruction by adding an offset to reconstructed video data.

[0117] Referring again to FIG. 4, entropy encoding unit 218 receives quantized transform coefficients and predictive syntax data (i.e., intra prediction data, motion prediction data, QP data, etc.). It should be noted that in some examples, coefficient quantization unit 206 may perform a scan of a matrix including quantized transform coefficients before the coefficients are output to entropy encoding unit 218. In other examples, entropy encoding unit 218 may perform a scan. Entropy encoding unit 218 may be configured to perform entropy encoding according to one or more of the techniques described herein. Entropy encoding unit 218 may be configured to output a compliant bitstream, i.e., a bitstream that a video decoder can receive and reproduce video data therefrom.

[0118] FIG. 5 is a block diagram illustrating an example of an entropy encoder that may be configured to encode values of syntax elements according to one or more techniques of this disclosure. Entropy encoding unit 300 may include a context adaptive entropy encoding unit, e.g., a CABAC encoder. As illustrated in FIG. 5, entropy encoder 300 includes binarization unit 302, binary arithmetic encoding unit 304, and context modeling unit 310. Entropy encoding unit 300 may receive one or more syntax elements values and output a compliant bitstream. As described above, binarization includes representing a value of syntax element in to a bin string. Binarization unit 302 may be configured to receive a value for a syntax element and produce a bin string according to one or more binarization techniques. Binarization unit 302 may use, for example, any one or combination of the following techniques to produce a bin string: fixed length coding, unary coding, truncated unary coding, truncated Rice coding, Golomb coding, exponential Golomb coding, and Golomb-Rice coding.

[0119] As described above, binary arithmetic encoding codes a series of 0's and 1's based on the principle of recursive interval subdivision, where for a received binary string (b_1 ,...,b_N), an interval R_0 is recursively divided based on the estimated probability of b_i being the LPS, and bits are written to a bitstream according to a renormalization process. As further described above, the estimated probability of b_i being the LPS is based on a context index. Binary arithmetic encoding unit 304 is configured to receive a bin string from binarization unit 302 and a context index corresponding to a bin from context modeling unit 306, and perform binary arithmetic encoding on the bin string. That is, binary arithmetic encoding unit 304 is configured to write bits to a bitstream according to a renormalization process and further indicate an observed value of a bin

such that a context model may be updated. The context models may be defined according to a video coding standard, such as for example, ITU-T H.265. The context models may be stored in a memory. For example, context modeling unit 306 may store a series of indexed tables and/or utilize mapping functions to determine a context model for a particular bin. It should be noted that the functions of binary coding are not limited to particular function blocks and the example of binary arithmetic encoding unit 304 and context modeling unit 306 in the example FIG. 5 should not be construed as limiting.

[0120] As described above, the values of initValue specified in JVET-M1001 for syntax elements may be less than ideal. According to the techniques herein, in one example, the values of initValue for respective syntax element may be as provided in Tables 29-87 and Tables 29A-87A. It should be noted that in Tables 29-87, initValues correspond to the example initialization of variables pStateL and pStateH as provided above in paragraphs and in Tables 29A-87A initValues correspond to the example initialization of variables pStateL and pStateH as provided above in paragraphs.

Initialization variable	ctxIdx of **alf_ctb_flag**								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	219	236	238	232	249	235	246	234	251
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	139	186	203	183	247	249	183	232	249
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
	154	186	174	183	233	250	168	248	250
	shift [ctxInc]								
	0	0	4	0	0	1	0	0	1

Table 29

Initialization variable	ctxIdx of **sao_merge_left_flag**
	0
initValue	199
	1
initValue	244
	2
initValue	47
	shift [ctxInc]
	0

Table 30

Initialization variable	ctxIdx of sao_merge_up_flag
	0
initValue	199
	1
initValue	244
	2
initValue	47
	shift [ctxInc]
	0

Table 31

Initialization variable	ctxIdx of sao_type_idx_luma
	0
initValue	95
	1
initValue	95
	2
initValue	47
	shift [ctxInc]
	0

Table 32

Initialization variable	ctxIdx of sao_type_idx_chroma
	0
initValue	95
	1
initValue	95
	2
initValue	47
	shift [ctxInc]
	0

Table 33

Initialization variable	ctxIdx of **split_qt_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	139	125	127	136	153	126
	6	**7**	**8**	**9**	**10**	**11**
initValue	139	126	142	107	138	125
	12	**13**	**14**	**15**	**16**	**17**
initValue	138	140	142	136	138	140
	shift [ctxInc]					
	0	8	8	12	12	8

Table 34

Initialization variable	ctxIdx of **split_cu_flag**								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	138	154	172	124	140	142	154	127	175
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	93	139	171	124	125	141	139	141	158
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
initValue	122	124	141	108	125	156	138	126	143
	shift [ctxInc]								
	9	13	8	8	13	12	5	10	12

Table 35

Initialization variable	ctxIdx of **mtt_split_cu_vertical_flag**				
	0	**1**	**2**	**3**	**4**
initValue	154	168	140	153	169
	5	**6**	**7**	**8**	**9**
initValue	169	168	170	153	170
	10	**11**	**12**	**13**	**14**
initValue	154	168	155	139	155
	shift [ctxInc]				
	10	9	9	8	8

Table 36

Initialization variable	ctxIdx of **mtt_split_cu_binary_flag**			
	0	**1**	**2**	**3**
initValue	154	170	154	170
	4	**5**	**6**	**7**
initValue	169	155	154	140
	8	**9**	**10**	**11**
initValue	154	140	154	140
	shift [ctxInc]			
	12	12	12	12

Table 37

Initialization variable	ctxIdx of **cu_skip_flag**		
	0	**1**	**2**
initValue	40	138	154
	3	**4**	**5**
initValue	197	198	185
	6	**7**	**8**
initValue	197	214	216
	shift [ctxInc]		
	5	8	8

Table 38

Initialization variable	ctxIdx of **pred_mode_flag**	
	0	**1**
initValue	154	154
	2	**3**
initValue	165	139
	2	**3**
initValue	192	168
	shift [ctxInc]	
	5	2

Table 39

Initialization variable	ctxIdx of pred_mode_cpr_flag		
	0	**1**	**2**
initValue	132	153	125
	3	**4**	**5**
initValue	0	153	140
	6	**7**	**8**
initValue	0	154	141
	shift [ctxInc]		
	5	5	8

Table 40

Initialization variable	ctxIdx of intra_luma_ref_idx		
	0	**1**	**2**
initValue	119	169	154
	3	**4**	**5**
initValue	118	212	154
	6	**7**	**8**
initValue	90	212	154
	shift [ctxInc]		
	8	8	8

Table 41

Initialization variable	ctxIdx of intra_subpartitions_mode_flag
	0
initValue	152
	1
initValue	166
	2
initValue	152
	shift [ctxInc]
	8

Table 42

Initialization variable	ctxIdx of **intra_subpartitions_split_flag**
	0
initValue	154
	1
initValue	154
	2
initValue	154
	shift [ctxInc]
	5

Table 43

Initialization variable	ctxIdx of **intra_luma_mpm_flag**
	0
initValue	170
	1
initValue	154
	2
initValue	154
	shift [ctxInc]
	6

Table 44

Initialization variable	ctxIdx of **intra_chroma_pred_mode**		
	0	**1**	**2**
initValue	154	139	154
	3	**4**	**5**
initValue	138	139	169
	6	**7**	**8**
initValue	137	139	140
	shift [ctxInc]		
	5	8	9

Table 45

Initialization variable	ctxIdx of **merge_subblock_idx**
	0
initValue	154
	1
initValue	95
	2
initValue	109
	shift [ctxInc]
	0

Table 46

Initialization variable	ctxIdx of **merge_flag**
	0
initValue	153
	1
initValue	111
	2
initValue	111
	shift [ctxInc]
	5

Table 47

Initialization variable	ctxIdx of **inter_pred_idc**				
	0	**1**	**2**	**3**	**4**
initValue	154	154	154	154	154
	5	**6**	**7**	**8**	**9**
initValue	126	111	110	94	208
	10	**11**	**12**	**13**	**14**
initValue	111	125	110	94	192
	shift [ctxInc]				
	0	0	4	5	0

Table 48

Initialization variable	ctxIdx of **inter_affine_flag**		
	0	**1**	**2**
initValue	154	154	154
	3	**4**	**5**
initValue	180	168	155
	6	**7**	**8**
initValue	179	169	171
	shift [ctxInc]		
	8	5	4

Table 49

Initialization variable	ctxIdx of **cu_affine_type_flag**
	0
initValue	154
	1
initValue	153
	2
initValue	138
	shift [ctxInc]
	4

Table 50

Initialization variable	ctxIdx of **sym_mvd_flag**
	0
initValue	154
	1
initValue	125
	2
initValue	154
	shift [ctxInc]
	5

Table 51

Initialization variable	ctxIdx of **ref_idx_l0**	
	0	**1**
initValue	138	168
	2	**3**
initValue	125	139
	shift [ctxInc]	
	4	5

Table 52

Initialization variable	ctxIdx of **ref_idx_l1**	
	0	**1**
initValue	138	168
	2	**3**
initValue	125	139
	shift [ctxInc]	
	4	5

Table 53

Initialization variable	ctxIdx of **mvp_l0_flag**
	0
initValue	168
	1
initValue	168
	2
initValue	153
	shift [ctxInc]
	10

Table 54

Initialization variable	ctxIdx of **mvp_l1_flag**
	0
initValue	168
	1
initValue	168
	2
initValue	153
	shift [ctxInc]
	10

Table 55

Initialization variable	ctxIdx of **amvr_flag**			
	0	**1**	**2**	**3**
initValue	154	154	154	152
	5	**6**	**7**	**8**
initValue	213	229	244	166
	10	**11**	**12**	**13**
initValue	212	199	215	180
	shift [ctxInc]			
	1	4	4	5

Table 56

Initialization variable	ctxIdx of **amvr_4pel_flag**	
	0	**1**
initValue	154	154
	2	**3**
initValue	198	244
	4	5
initValue	183	242
	shift [ctxInc]	
	1	0

Table 57

Initialization variable	ctxIdx of **gbi_idx**						
	0	**1**	**2**	**3**	**4**	**5**	**6**
initValue	154	154	154	154	154	154	154
	9	**10**	**11**	**12**	**13**	**14**	**15**
initValue	242	154	154	154	154	170	237
	18	**19**	**20**	**21**	**22**	**23**	**24**
initValue	228	154	154	154	125	155	175
	shift [ctxInc]						
	4	8	8	8	4	0	0

Table 58

Initialization variable	ctxIdx of **cu_cbf**
	0
initValue	110
	1
initValue	95
	2
initValue	109
	shift [ctxInc]
	4

Table 59

Initialization variable	ctxIdx of **cu_sbt_flag**	
	0	**1**
initValue	154	154
	2	**3**
initValue	197	183
	4	**5**
initValue	168	183
	shift [ctxInc]	
	4	8

Table 60

Initialization variable	ctxIdx of **cu_sbt_quad_flag**
	0
initValue	154
	1
initValue	168
	2
initValue	168
	shift [ctxInc]
	9

Table 61

Initialization variable	ctxIdx of **cu_sbt_horizontal_flag**		
	0	**1**	**2**
initValue	154	154	154
	3	**4**	**5**
initValue	139	154	139
	6	**7**	**8**
initValue	139	154	139
	shift [ctxInc]		
	8	5	4

Table 62

Initialization variable	ctxIdx of **cu_sbt_pos_flag**
	0
initValue	154
	1
initValue	154
	2
initValue	154
	shift [ctxInc]
	13

Table 63

Initialization variable	ctxIdx of **mmvd_flag**
	0
initValue	154
	1
initValue	122
	2
initValue	120
	shift [ctxInc]
	8

Table 64

Initialization variable	ctxIdx of **mmvd_merge_flag**
	0
initValue	154
	1
initValue	154
	2
initValue	154
	shift [ctxInc]
	10

Table 65

Initialization variable	ctxIdx of **mmvd_distance_idx**
	0
initValue	154
	1
initValue	244
	2
initValue	213
	shift [ctxInc]
	1

Table 66

Initialization variable	ctxIdx of **merge_triangle_flag**		
	0	**1**	**2**
initValue	154	154	154
	3	**4**	**5**
initValue	151	152	138
	6	**7**	**8**
initValue	149	123	123
	shift [ctxInc]		
	8	12	9

Table 67

Initialization variable	ctxIdx of **merge_triangle_idx**
	0
initValue	154
	1
initValue	154
	2
initValue	154
	shift [ctxInc]
	8

Table 68

Initialization variable	ctxIdx of **merge_idx**
	0
initValue	153
	1
initValue	154
	2
initValue	138
	shift [ctxInc]
	8

Table 69

Initialization variable	ctxIdx of **ciip_flag**
	0
initValue	154
	1
initValue	197
	2
initValue	225
	shift [ctxInc]
	1

Table 70

Initialization variable	ctxIdx of **ciip_luma_mpm_flag**			
	0	**1**	**2**	**3**
initValue	154	154	154	154
	4	**5**	**6**	**7**
initValue	156	154	154	154
	8	**9**	**10**	**11**
initValue	156	154	154	154
	shift [ctxInc]			
	9	8	8	8

Table 71

Initialization variable	ctxIdx of **abs_mvd_greater0_flag**
	0
initValue	141
	1
initValue	155
	2
initValue	169
	shift [ctxInc]
	9

Table 72

Initialization variable	ctxIdx of abs_mvd_greater1_flag
	0
initValue	156
	1
initValue	154
	2
initValue	183
	shift [ctxInc]
	5

Table 73

Initialization variable	ctxIdx of tu_cbf_luma			
	0	**1**	**2**	**3**
initValue	154	111	124	111
	4	**5**	**6**	**7**
initValue	142	127	139	140
	8	**9**	**10**	**11**
initValue	141	127	139	140
	shift [ctxInc]			
	1	5	9	8

Table 74

Initialization variable	ctxIdx of tu_cbf_cb				
	0	**1**	**2**	**3**	**4**
initValue	109	154	154	154	154
	5	**6**	**7**	**8**	**9**
initValue	164	154	154	154	154
	10	**11**	**12**	**13**	**14**
initValue	163	154	154	154	154
	shift [ctxInc]				
	5	8	8	8	8

Table 75

Initialization variable	ctxIdx of tu_cbf_cr	
	0	**1**
initValue	151	155
	2	**3**
initValue	192	154
	4	**5**
initValue	161	154
	shift [ctxInc]	
	5	5

Table 76

Initialization variable	ctxIdx of cu_qp_delta_abs		
	0	**1**	**2**
initValue	154	154	154
	3	**4**	**5**
initValue	154	154	154
	6	**7**	**8**
initValue	154	154	154
	shift [ctxInc]		
	8	8	8

Table 77

Initialization variable	ctxIdx of last_sig_coeff_x_prefix											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	125	140	124	111	111	109	111	126	125	123	111	141
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**		
initValue	111	125	79	155	142	170	140	138	108	47		
	22	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**
initValue	125	110	109	125	125	123	111	111	95	123	126	111
	34	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**		
initValue	110	95	169	154	140	139	139	138	123	92		
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**
initValue	111	111	110	111	111	139	111	126	111	139	126	126
	56	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**		
initValue	111	111	169	154	111	110	110	122	124	63		
	shift [ctxInc]											
	8	5	5	5	4	4	5	4	4	0	5	1
	0	0	0	1	1	0	0	2	1	1		

Table 80

Initialization variable	ctxIdx of last_sig_coeff_y_prefix											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	110	110	109	125	111	123	111	126	95	108	111	127
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**		
initValue	111	95	78	169	157	141	125	123	123	91		
	22	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**
initValue	95	95	109	110	110	108	125	111	124	123	140	111
	34	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**		
initValue	110	124	139	125	126	110	124	108	123	121		
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**
initValue	125	125	139	125	111	139	111	111	110	110	140	126
	56	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**		
initValue	125	125	140	139	111	110	124	122	124	123		
	shift [ctxInc]											
	8	5	8	5	5	4	5	5	4	0	5	5
	1	0	0	1	4	1	0	2	2	2		

Table 81

Initialization variable	ctxIdx of **coded_sub_block_flag**			
	0	**1**	**2**	**3**
initValue	107	158	76	127
	4	**5**	**6**	**7**
initValue	106	156	90	141
	8	**9**	**10**	**11**
initValue	105	155	91	155
	shift [ctxInc]			
	8	5	5	8

Table 82

Initialization variable	ctxIdx of **sig_coeff_flag**														
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
initValue	89	138	153	139	154	140	134	139	139	140	140	141	137	170	169
	15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
initValue	170	141	157	43	153	168	169	154	155	152	215	155	201	171	143
	30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**
initValue	108	142	157	143	188	175	138	238	205	238	253	237	139	223	223
	45	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
initValue	223	223	253	152	173	157	187	204	253	170	223	223	223	223	223
	60	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**
initValue	107	174	223	207	206	223	63	223	223	238	223	238	12	223	223
	75	**76**	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**
initValue	223	223	253	166	234	223	236	248	223	108	223	223	223	223	223
	90	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**	**103**	**104**
initValue	132	152	167	168	183	140	177	182	168	154	169	155	180	213	183
	105	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**
initValue	169	184	156	133	138	153	139	154	140	181	229	169	229	170	157
	120	**121**	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**	**132**	**133**	**134**
initValue	123	142	157	172	172	218	138	249	248	248	219	223	139	223	223
	135	**136**	**137**	**138**	**139**	**140**	**141**	**142**	**143**	**144**	**145**	**146**	**147**	**148**	**149**
initValue	223	223	223	168	156	173	216	172	219	169	223	223	223	223	223
	150	**151**	**152**	**153**	**154**	**155**	**156**	**157**	**158**	**159**	**160**	**161**	**162**	**163**	**164**
initValue	123	175	223	175	218	223	138	223	223	223	222	223	196	223	223
	165	**166**	**167**	**168**	**169**	**170**	**171**	**172**	**173**	**174**	**175**	**176**	**177**	**178**	**179**
initValue	223	223	223	167	171	223	175	248	223	152	223	223	223	223	223
	180	**181**	**182**	**183**	**184**	**185**	**186**	**187**	**188**	**189**	**190**	**191**	**192**	**193**	**194**
initValue	88	166	152	182	168	154	0	167	182	168	183	155	193	213	183
	195	**196**	**197**	**198**	**199**	**200**	**201**	**202**	**203**	**204**	**205**	**206**	**207**	**208**	**209**
initValue	183	169	185	72	167	153	168	154	155	180	199	183	199	199	186
	210	**211**	**212**	**213**	**214**	**215**	**216**	**217**	**218**	**219**	**220**	**221**	**222**	**223**	**224**
initValue	152	156	201	186	186	187	182	248	188	232	188	205	182	223	223
	225	**226**	**227**	**228**	**229**	**230**	**231**	**232**	**233**	**234**	**235**	**236**	**237**	**238**	**239**
initValue	223	223	223	182	171	143	158	172	189	183	223	223	223	223	223

	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254
initValue	123	173	223	191	232	251	212	223	223	236	206	223	192	223	223
	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269
initValue	223	223	223	167	201	223	248	219	223	181	223	223	223	223	223
	shift [ctxInc]														
	12	9	9	9	9	10	9	9	9	9	9	9	8	8	8
	8	8	9	9	9	12	9	13	13	5	5	8	8	8	9
	9	12	8	8	8	8	8	8	8	8	8	5	8	0	0
	0	0	0	8	9	12	8	8	8	4	0	2	2	2	2
	8	8	4	8	8	8	8	0	0	4	8	5	4	2	2
	2	2	1	8	8	5	8	8	8	5	1	2	2	2	2

Table 83

Initialization variable	ctxIdx of par_level_flag											
	0	1	2	3	4	5	6	7	8	9	10	11
initValue	121	135	137	152	138	153	91	137	138	153	139	151
	12	13	14	15	16	17	18	19	20			
initValue	138	153	139	139	138	168	139	154	139			
	21	22	23	24	25	26	27	28	29	30	31	
initValue	136	135	152	153	138	153	136	168	154	139	154	
	32	33	34	35	36	37	38	39	40	41	42	43
initValue	121	119	136	137	138	153	104	122	138	153	139	106
	44	45	46	47	48	49	50	51	52			
initValue	138	153	168	139	137	153	168	139	139			
	53	54	55	56	57	58	59	60	61	62	63	
initValue	135	120	137	138	138	153	136	153	168	139	154	
	64	65	66	67	68	69	70	71	72	73	74	75
initValue	121	105	136	152	138	183	90	122	167	153	168	135
	76	77	78	79	80	81	82	83	84			
initValue	152	153	168	139	151	153	139	168	154			
	85	86	87	88	89	90	91	92	93	94	95	
initValue	151	120	152	138	153	153	136	168	154	168	154	
	shift [ctxInc]											
	8	9	12	13	13	13	13	13	13	13	13	10
	13	13	13	13	10	13	13	13	13			
	8	10	12	12	13	13	10	10	13	13	13	

Table 84

Initialization variable	ctxIdx of abs_level_gt1_flag											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	90	72	119	135	137	138	43	60	106	137	109	58
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**			
initValue	106	108	109	124	121	138	139	154	155			
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	
initValue	194	40	120	122	122	138	103	121	153	154	155	
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**
initValue	14	116	86	119	106	152	0	72	120	151	138	116
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**			
initValue	90	107	152	153	104	107	123	153	154			
	53	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	
initValue	117	0	90	106	92	93	147	136	138	154	140	
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**
initValue	31	73	118	75	152	109	42	44	105	107	109	0
	76	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**			
initValue	119	136	152	124	118	136	138	153	140			
	85	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	
initValue	119	101	134	151	107	123	118	122	124	140	155	
	shift [ctxInc]											
	4	1	8	8	4	2	5	9	9	8	9	9
	9	9	8	9	9	8	9	8	8			
	2	5	8	8	8	6	6	8	8	8	7	

Table 85

Initialization variable	ctxIdx of **abs_level_gt3_flag**											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	134	120	123	153	139	140	92	124	154	125	111	138
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**			
initValue	154	140	155	141	154	140	185	171	143			
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	
initValue	163	136	153	154	125	140	183	170	201	187	174	
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**
initValue	101	133	137	153	139	140	134	138	139	154	155	0
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**			
initValue	153	154	140	170	138	154	155	170	186			
	53	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	
initValue	0	135	153	139	125	140	182	155	156	142	159	
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**
initValue	43	177	181	168	154	170	133	167	139	154	155	164
	76	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**			
initValue	153	154	169	155	181	183	169	185	186			
	85	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	
initValue	0	178	153	154	140	140	196	170	186	157	188	
	shift [ctxInc]											
	8	5	9	9	12	9	9	10	13	12	10	9
	10	10	10	10	8	9	8	8	10			
	6	9	10	12	12	10	5	9	8	8	9	

Table 86

Initialization variable	ctxIdx of **tu_mts_idx**										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
initValue	219	154	154	140	140	140	154	138	153	9	
	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
initValue	233	154	155	155	140	140	154	167	153	0	
	20	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	
initValue	216	155	155	140	140	154	155	153	153	0	
	shift [ctxInc]										
	4	8	8	8	8	8	8	8	9	3	

Table 87

Initialization variable	ctxIdx of **alf_ctb_flag**								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	136	168	171	209	245	233	179	231	233
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	107	168	185	165	243	231	165	243	231
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
	201	249	175	184	246	247	184	246	233
	shift [ctxInc]								
	0	0	4	0	0	4	0	0	4

Table 29A

Initialization variable	ctxIdx of **sao_merge_left_flag**
	0
initValue	28
	1
initValue	240
	2
initValue	181
	shift [ctxInc]
	0

Table 30A

Initialization variable	ctxIdx of **sao_merge_up_flag**
	0
initValue	28
	1
initValue	240
	2
initValue	181
	shift [ctxInc]
	0

Table 31A

Initialization variable	ctxIdx of sao_type_idx_luma
	0
initValue	14
	1
initValue	63
	2
initValue	92
	shift [ctxInc]
	0

Table 32A

Initialization variable	ctxIdx of sao_type_idx_chroma
	0
initValue	14
	1
initValue	63
	2
initValue	92
	shift [ctxInc]
	0

Table 33A

Initialization variable	ctxIdx of split_qt_flag					
	0	**1**	**2**	**3**	**4**	**5**
initValue	105	122	139	118	120	137
	6	**7**	**8**	**9**	**10**	**11**
initValue	121	108	95	89	120	107
	12	**13**	**14**	**15**	**16**	**17**
initValue	106	107	94	118	135	108
	shift [ctxInc]					
	0	8	8	12	12	8

Table 34A

Initialization variable	ctxIdx of **split_cu_flag**								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	104	106	94	75	107	138	120	108	110
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	75	121	153	106	107	123	121	123	126
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
initValue	120	136	154	106	122	124	136	124	143
	shift [ctxInc]								
	12	13	8	8	13	12	5	10	12

Table 35A

Initialization variable	ctxIdx of **mtt_split_cu_vertical_flag**				
	0	**1**	**2**	**3**	**4**
initValue	136	150	137	121	137
	5	**6**	**7**	**8**	**9**
initValue	151	150	152	135	152
	10	**11**	**12**	**13**	**14**
initValue	136	150	122	135	166
	shift [ctxInc]				
	10	9	9	8	8

Table 36A

Initialization variable	ctxIdx of **mtt_split_cu_binary_flag**			
	0	**1**	**2**	**3**
initValue	136	137	136	122
	4	**5**	**6**	**7**
initValue	136	123	136	137
	8	**9**	**10**	**11**
initValue	136	152	136	152
	shift [ctxInc]			
	12	12	12	13

Table 37A

Initialization variable	ctxIdx of **cu_skip_flag**		
	0	**1**	**2**
initValue	179	196	198
	3	**4**	**5**
initValue	193	180	196
	6	**7**	**8**
initValue	112	120	151
	shift [ctxInc]		
	5	8	8

Table 38A

Initialization variable	ctxIdx of **pred_mode_flag**	
	0	**1**
initValue	144	150
	2	**3**
initValue	161	121
	2	**3**
initValue	136	136
	shift [ctxInc]	
	5	2

Table 39A

Initialization variable	ctxIdx of **pred_mode_cpr_flag**		
	0	**1**	**2**
initValue	6	136	138
	3	**4**	**5**
initValue	97	150	152
	6	**7**	**8**
initValue	114	135	107
	shift [ctxInc]		
	5	5	8

Table 40A

Initialization variable	ctxIdx of **intra_luma_ref_idx**		
	0	**1**	**2**
initValue	57	194	136
	3	**4**	**5**
initValue	114	179	136
	6	**7**	**8**
initValue	86	151	136
	shift [ctxInc]		
	8	8	8

Table 41A

Initialization variable	ctxIdx of **intra_subpartitions_mode_flag**
	0
initValue	147
	1
initValue	132
	2
initValue	133
	shift [ctxInc]
	9

Table 42A

Initialization variable	ctxIdx of **intra_subpartitions_split_flag**
	0
initValue	136
	1
initValue	136
	2
initValue	136
	shift [ctxInc]
	4

Table 43A

Initialization variable	ctxIdx of **intra_luma_mpm_flag**
	0
initValue	136
	1
initValue	136
	2
initValue	1123
	shift [ctxInc]
	9

Table 44A

Initialization variable	ctxIdx of **intra_chroma_pred_mode**		
	0	**1**	**2**
initValue	104	121	122
	3	**4**	**5**
initValue	120	121	151
	6	**7**	**8**
initValue	136	121	136
	shift [ctxInc]		
	5	9	9

Table 45A

Initialization variable	ctxIdx of **merge_subblock_idx**
	0
initValue	91
	1
initValue	77
	2
initValue	136
	shift [ctxInc]
	0

Table 46A

Initialization variable	ctxIdx of **merge_flag**
	0
initValue	93
	1
initValue	93
	2
initValue	120
	shift [ctxInc]
	5

Table 47A

Initialization variable	ctxIdx of **inter_pred_idc**				
	0	**1**	**2**	**3**	**4**
initValue	93	107	92	76	144
	5	**6**	**7**	**8**	**9**
initValue	94	93	92	76	161
	10	**11**	**12**	**13**	**14**
initValue	136	136	136	136	136
	shift [ctxInc]				
	0	0	4	5	0

Table 48A

Initialization variable	ctxIdx of **inter_affine_flag**		
	0	**1**	**2**
initValue	161	166	168
	3	**4**	**5**
initValue	162	150	137
	6	**7**	**8**
initValue	136	136	136
	shift [ctxInc]		
	5	5	4

Table 49A

Initialization variable	ctxIdx of **cu_affine_type_flag**
	0
initValue	120
	1
initValue	135
	2
initValue	136
	shift [ctxInc]
	4

Table 50A

Initialization variable	ctxIdx of **sym_mvd_flag**
	0
initValue	136
	1
initValue	136
	2
initValue	136
	shift [ctxInc]
	8

Table 51A

Initialization variable	ctxIdx of **ref_idx_l0**	
	0	**1**
initValue	107	121
	2	**3**
initValue	106	150
	shift [ctxInc]	
	4	5

Table 52A

Initialization variable	ctxIdx of **ref_idx_l1**	
	0	**1**
initValue	107	121
	2	**3**
initValue	106	105
	shift [ctxInc]	
	4	5

Table 53A

Initialization variable	ctxIdx of **mvp_l0_flag**
	0
initValue	135
	1
initValue	168
	2
initValue	135
	shift [ctxInc]
	10

Table 54A

Initialization variable	ctxIdx of **mvp_l1_flag**
	0
initValue	135
	1
initValue	150
	2
initValue	135
	shift [ctxInc]
	10

Table 55A

Initialization variable	ctxIdx of amvr_flag			
	0	**1**	**2**	**3**
initValue	194	181	197	162
	5	**6**	**7**	**8**
initValue	195	196	211	162
	10	**11**	**12**	**13**
initValue	136	136	136	119
	shift [ctxInc]			
	1	4	4	5

Table 56A

Initialization variable	ctxIdx of amvr_4pel_flag	
	0	**1**
initValue	165	224
	2	**3**
initValue	194	240
	4	5
initValue	136	136
	shift [ctxInc]	
	1	0

Table 57A

Initialization variable	ctxIdx of gbi_idx						
	0	**1**	**2**	**3**	**4**	**5**	**6**
initValue	210	136	136	136	107	137	159
	9	**10**	**11**	**12**	**13**	**14**	**15**
initValue	224	136	136	136	136	152	204
	18	**19**	**20**	**21**	**22**	**23**	**24**
initValue	136	136	136	136	136	136	136
	shift [ctxInc]						
	4	8	8	8	4	0	0

Table 58A

Initialization variable	ctxIdx of **cu_cbf**
	0
initValue	62
	1
initValue	63
	2
initValue	92
	shift [ctxInc]
	4

Table 59A

Initialization variable	ctxIdx of **cu_sbt_flag**	
	0	**1**
initValue	164	165
	2	**3**
initValue	193	179
	4	**5**
initValue	136	136
	shift [ctxInc]	
	4	8

Table 60A

Initialization variable	ctxIdx of **cu_sbt_quad_flag**
	0
initValue	150
	1
initValue	150
	2
initValue	136
	shift [ctxInc]
	9

Table 61A

Initialization variable	ctxIdx of **cu_sbt_horizontal_flag**		
	0	**1**	**2**
initValue	121	136	121
	3	**4**	**5**
initValue	121	136	106
	6	**7**	**8**
initValue	136	136	136
	shift [ctxInc]		
	8	4	4

Table 62A

Initialization variable	ctxIdx of **cu_sbt_pos_flag**
	0
initValue	136
	1
initValue	136
	2
initValue	136
	shift [ctxInc]
	13

Table 63A

Initialization variable	ctxIdx of **mmvd_flag**
	0
initValue	116
	1
initValue	104
	2
initValue	136
	shift [ctxInc]
	8

Table 64A

Initialization variable	ctxIdx of **mmvd_merge_flag**
	0
initValue	136
	1
initValue	136
	2
initValue	136
	shift [ctxInc]
	10

Table 65A

Initialization variable	ctxIdx of **mmvd_distance_idx**
	0
initValue	180
	1
initValue	240
	2
initValue	136
	shift [ctxInc]
	1

Table 66A

Initialization variable	ctxIdx of **merge_triangle_flag**		
	0	**1**	**2**
initValue	116	120	120
	3	**4**	**5**
initValue	119	120	120
	6	**7**	**8**
initValue	136	136	136
	shift [ctxInc]		
	8	12	8

Table 67A

Initialization variable	ctxIdx of **merge_triangle_idx**
	0
initValue	136
	1
initValue	136
	2
initValue	136
	shift [ctxInc]
	8

Table 68A

Initialization variable	ctxIdx of **merge_idx**
	0
initValue	120
	1
initValue	136
	2
initValue	135
	shift [ctxInc]
	8

Table 69A

Initialization variable	ctxIdx of **ciip_flag**
	0
initValue	192
	1
initValue	193
	2
initValue	136
	shift [ctxInc]
	1

Table 70A

Initialization variable	ctxIdx of **ciip_luma_mpm_flag**			
	0	**1**	**2**	**3**
initValue	138	136	136	136
	4	**5**	**6**	**7**
initValue	138	136	136	136
	8	**9**	**10**	**11**
initValue	136	136	136	136
	shift [ctxInc]			
	9	8	8	8

Table 71A

Initialization variable	ctxIdx of **abs_mvd_greater0_flag**
	0
initValue	151
	1
initValue	137
	2
initValue	138
	shift [ctxInc]
	9

Table 72A

Initialization variable	ctxIdx of **abs_mvd_greater1_flag**
	0
initValue	165
	1
initValue	138
	2
initValue	153
	shift [ctxInc]
	5

Table 73A

Initialization variable	ctxIdx of tu_cbf_luma			
	0	**1**	**2**	**3**
initValue	109	109	121	122
	4	**5**	**6**	**7**
initValue	111	109	121	93
	8	**9**	**10**	**11**
initValue	136	79	106	93
	shift [ctxInc]			
	1	5	9	8

Table 74 A

Initialization variable	ctxIdx of tu_cbf_cb				
	0	**1**	**2**	**3**	**4**
initValue	144	136	136	136	136
	5	**6**	**7**	**8**	**9**
initValue	117	121	136	136	136
	10	**11**	**12**	**13**	**14**
initValue	91	136	136	136	136
	shift [ctxInc]				
	5	8	8	8	8

Table 75A

Initialization variable	ctxIdx of tu_cbf_cr	
	0	**1**
initValue	128	136
	2	**3**
initValue	144	136
	4	**5**
initValue	133	137
	shift [ctxInc]	
	5	5

Table 76A

Initialization variable	ctxIdx of **cu_qp_delta_abs**		
	0	**1**	**2**
initValue	136	136	136
	3	**4**	**5**
initValue	136	136	136
	6	**7**	**8**
initValue	136	136	136
	shift [ctxInc]		
	8	8	8

Table 77A

Initialization variable	ctxIdx of **last_sig_coeff_x_prefix**											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	79	93	92	79	79	121	93	79	93	121	108	79
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**		
initValue	79	79	151	136	79	92	107	104	77	45		
	22	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**
initValue	107	92	91	107	78	120	93	93	92	120	108	93
	34	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**		
initValue	92	92	151	107	122	121	121	105	120	74		
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**
initValue	122	122	106	93	108	91	93	79	107	76	93	94
	56	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**		
initValue	93	122	76	137	139	152	122	120	105	29		
	shift [ctxInc]											
	8	5	5	5	4	4	5	4	4	0	5	1
	0	0	0	1	1	0	0	2	1	1		

Table 80A

Initialization variable	ctxIdx of **last_sig_coeff_y_prefix**											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	107	107	136	107	93	121	93	93	107	107	122	79
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**		
initValue	107	122	122	121	79	92	106	104	121	120		
	22	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**
initValue	77	92	91	92	78	105	107	93	77	120	122	93
	34	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**		
initValue	92	77	136	107	108	92	106	90	105	88		
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**
initValue	92	92	91	78	93	105	93	79	77	90	93	109
	56	**57**	**58**	**59**	**60**	**61**	**62**	**63**	**64**	**65**		
initValue	78	62	90	151	154	123	107	105	105	88		
	shift [ctxInc]											
	8	5	8	5	5	4	5	5	4	0	5	5
	1	0	0	1	4	1	0	2	2	2		

Table 81A

Initialization variable	ctxIdx of **coded_sub_block_flag**			
	0	**1**	**2**	**3**
initValue	108	137	58	152
	4	**5**	**6**	**7**
initValue	73	138	72	123
	8	**9**	**10**	**11**
initValue	89	111	58	94
	shift [ctxInc]			
	8	5	5	8

Table 82A

Initialization variable	ctxIdx of **sig_coeff_flag**														
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
initValue	99	148	163	149	135	136	6	149	164	150	165	137	160	195	165
	15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
initValue	165	151	167	113	149	135	150	150	137	147	181	165	181	181	168
	30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**
initValue	134	138	168	153	153	154	164	245	170	214	170	202	164	220	220
	45	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
initValue	207	191	207	164	153	125	140	154	172	165	207	207	207	175	206
	60	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**
initValue	105	141	207	207	229	234	194	207	207	207	250	207	128	207	207
	75	**76**	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**
initValue	207	207	207	149	183	175	245	231	159	148	205	207	190	207	207
	90	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**	**103**	**104**
initValue	128	134	149	150	150	122	144	164	150	136	136	137	147	195	165
	105	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**
initValue	151	166	138	100	120	135	121	136	122	148	196	151	211	152	139
	120	**121**	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**	**132**	**133**	**134**
initValue	105	124	139	154	154	200	120	247	245	245	202	207	121	207	207
	135	**136**	**137**	**138**	**139**	**140**	**141**	**142**	**143**	**144**	**145**	**146**	**147**	**148**	**149**
initValue	207	207	207	150	153	155	213	154	231	151	207	207	207	251	236
	150	**151**	**152**	**153**	**154**	**155**	**156**	**157**	**158**	**159**	**160**	**161**	**162**	**163**	**164**
initValue	105	158	207	158	215	221	120	207	207	207	207	207	192	207	207
	165	**166**	**167**	**168**	**169**	**170**	**171**	**172**	**173**	**174**	**175**	**176**	**177**	**178**	**179**
initValue	207	207	207	149	168	207	158	245	190	134	175	207	207	207	207
	180	**181**	**182**	**183**	**184**	**185**	**186**	**187**	**188**	**189**	**190**	**191**	**192**	**193**	**194**
initValue	71	120	135	121	136	122	101	121	121	122	122	123	148	152	151
	195	**196**	**197**	**198**	**199**	**200**	**201**	**202**	**203**	**204**	**205**	**206**	**207**	**208**	**209**
initValue	152	123	139	55	135	150	151	136	137	134	197	137	183	153	125
	210	**211**	**212**	**213**	**214**	**215**	**216**	**217**	**218**	**219**	**220**	**221**	**222**	**223**	**224**
initValue	90	124	124	125	170	157	120	221	248	220	235	250	121	223	222
	225	**226**	**227**	**228**	**229**	**230**	**231**	**232**	**233**	**234**	**235**	**236**	**237**	**238**	**239**
initValue	206	190	236	134	155	139	169	186	221	152	222	222	207	223	237
	240	**241**	**242**	**243**	**244**	**245**	**246**	**247**	**248**	**249**	**250**	**251**	**252**	**253**	**254**

initValue	89	157	223	252	219	237	45	191	221	207	223	222	7	252	222
	255	**256**	**257**	**258**	**259**	**260**	**261**	**262**	**263**	**264**	**265**	**266**	**267**	**268**	**269**
initValue	207	223	191	148	246	223	234	201	237	75	223	223	223	223	223
	shift [ctxInc]														
	12	9	9	9	9	10	9	9	9	9	9	9	8	8	8
	8	8	9	9	9	12	9	13	13	4	5	8	8	8	9
	9	12	8	8	8	8	8	8	8	8	8	4	8	1	0
	0	0	0	8	9	12	8	8	8	4	1	3	3	3	3
	8	8	4	8	8	8	8	0	1	4	8	5	4	3	3
	3	3	2	8	8	5	8	8	8	4	2	3	3	3	3

Table 83A

Initialization variable	ctxIdx of **par_level_flag**											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	103	102	103	119	120	165	72	89	149	135	150	117
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**			
initValue	134	135	150	121	133	135	121	150	136			
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	
initValue	133	102	134	120	135	135	118	135	136	150	136	
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**
initValue	88	101	103	119	105	135	71	89	105	135	121	88
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**			
initValue	105	135	150	121	104	135	150	121	121			
	53	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	
initValue	117	87	119	120	120	135	118	135	150	121	136	
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**
initValue	103	117	119	134	120	135	73	119	120	135	121	133
	76	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**			
initValue	120	135	121	121	120	150	121	136	121			
	85	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	
initValue	118	102	134	135	120	135	118	150	136	121	136	
	shift [ctxInc]											
	8	9	12	13	13	13	10	13	13	13	13	13
	13	13	13	13	10	13	13	13	13			
	9	10	12	12	13	13	10	10	13	13	13	

Table 84A

Initialization variable	ctxIdx of **abs_level_gt1_flag**											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	28	55	55	57	105	91	113	55	72	104	91	68
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**			
initValue	101	118	134	106	100	118	120	135	122			
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	
initValue	101	52	101	133	89	105	100	104	106	122	152	
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**
initValue	40	0	98	71	88	134	0	24	87	118	120	7
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**			
initValue	72	89	134	135	71	89	105	135	136			
	53	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	
initValue	54	0	72	73	74	75	99	118	120	136	137	
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**
initValue	72	39	86	117	119	120	10	42	88	119	91	55
	76	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**			
initValue	88	90	91	106	103	120	121	136	108			
	85	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	
initValue	176	37	102	104	119	120	55	103	135	136	152	
	shift [ctxInc]											
	4	1	8	8	4	2	5	9	9	8	9	9
	9	9	8	9	9	8	9	8	9			
	1	5	8	8	8	6	6	8	8	9	5	

Table 85A

Initialization variable	ctxIdx of abs_level_gt3_flag											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	113	144	178	165	136	167	115	149	121	136	137	146
	12	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**			
initValue	135	136	151	152	163	165	151	167	139			
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	
initValue	113	160	135	136	122	122	178	152	168	124	170	
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**
initValue	68	115	119	135	121	122	101	120	121	136	137	118
	44	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**			
initValue	135	136	122	152	120	136	137	152	168			
	53	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**	
initValue	0	117	135	121	107	122	164	137	124	110	156	
	64	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**	**75**
initValue	101	102	105	106	136	122	74	106	136	107	93	120
	76	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**			
initValue	136	122	137	123	136	122	167	153	140			
	85	**86**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**94**	**95**	
initValue	130	118	135	136	107	122	165	152	183	184	171	
	shift [ctxInc]											
	8	5	9	9	12	9	9	10	13	12	10	9
	10	10	10	10	8	9	8	8	10			
	6	9	10	12	12	10	5	9	8	8	9	

Table 86A

Initialization variable	ctxIdx of tu_mts_idx									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
initValue	136	138	138	108	108	136	198	120	135	0
	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
initValue	136	138	138	123	123	136	230	134	135	0
	20	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
initValue	136	136	137	108	108	136	247	105	150	0
	shift [ctxInc]									
	8	8	8	8	8	8	0	9	9	10

Table 87A

[0121] As described above, the values of initValue specified in JVET-L1001 for syntax elements may be less than ideal. According to the techniques herein, in one example, the values of initValue for respective syntax element may be as provided in Tables 64B-103B.

Initialization variable	ctxIdx of alf_ctb_flag								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	155	205	253	168	187	234	168	187	220
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
initValue	153	156	188	137	185	218	152	185	218
	18	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
initValue	138	141	173	122	170	203	151	155	203

Table 64B

Initialization variable	ctxIdx of sao_merge_left_flag		
	0	**1**	**2**
initValue	184	214	92

Table 65B

Initialization variable	ctxIdx of sao_merge_up_flag		
	0	**1**	**2**
initValue	184	214	92

Table 66B

Initialization variable	ctxIdx of sao_type_idx_luma		
	0	**1**	**2**
initValue	110	111	77

Table 67B

Initialization variable	ctxIdx of sao_type_idx_chroma		
	0	**1**	**2**
initValue	110	111	77

Table 68B

Initialization variable	ctxIdx of qt_split_cu_flag					
	0	**1**	**2**	**3**	**4**	**5**
initValue	138	141	158	151	124	126
	6	**7**	**8**	**9**	**10**	**11**
initValue	138	111	143	107	138	140
	12	**13**	**14**	**15**	**16**	**17**
initValue	137	125	127	107	138	140

Table 69B

Initialization variable	ctxIdx of mtt_split_cu_flag															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
initValue	139	141	157	139	155	142	153	125	141	154	154	154	154	154	154	140
	16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
initValue	123	140	156	138	125	141	122	124	140	154	154	154	169	155	139	169
	32	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
initValue	137	125	141	123	125	141	78	124	140	154	154	154	169	155	154	154

Table 70B

Initialization variable	ctxIdx of mtt_split_cu_vertical_flag								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
initValue	154	154	154	169	155	139	169	155	154

Table 71B

Initialization variable	ctxIdx of **mtt_split_cu_binary_flag**		
	0	**1**	**2**
initValue	140	169	154

Table 72B

Initialization variable	ctxIdx of **cu_skip_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	168	199	200	183	185	186

Table 73B

Initialization variable	ctxIdx of **pred_mode_flag**	
	0	**1**
initValue	194	178

Table 74B

Initialization variable	ctxIdx of **intra_luma_mpm_flag**		
	0	**1**	**2**
initValue	156	154	183

Table 75B

Initialization variable	ctxIdx of **intra_chroma_pred_mode**		
	0	**1**	**2**
initValue	109	138	152

Table 76B

Initialization variable	ctxIdx of **mvp_l0_flag**	
	0	**1**
initValue	168	168

Table 77B

Initialization variable	ctxIdx of **mvp_l1_flag**	
	0	**1**
initValue	168	168

Table 78B

Initialization variable	ctxIdx of **merge_flag**	
	0	**1**
initValue	110	125

Table 79

Initialization variable	ctxIdx of **merge_idx**	
	0	**1**
initValue	138	167

Table 80B

Initialization variable	ctxIdx of **inter_pred_idc**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
initValue	126	111	95	93	194	111	110	95	78	193

Table 81B

Initialization variable	ctxIdx of **ref_idx_l0**			
	0	**1**	**2**	**3**
initValue	138	168	139	139

Table 82B

Initialization variable	ctxIdx of **ref_idx_l1**			
	0	**1**	**2**	**3**
initValue	138	168	139	139

Table 83B

Initialization variable	ctxIdx of **merge_affine_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	181	169	185	196	184	171

Table 84B

Initialization variable	ctxIdx of **inter_affine_flag**					
	0	**1**	**2**	**3**	**4**	**5**
initValue	181	169	185	196	184	171

Table 85B

Initialization variable	ctxIdx of **cu_affine_type_flag**		
	0	**1**	**2**
initValue	154	138	123

Table 86B

Initialization variable	ctxIdx of **amvr_mode**			
	0	**1**	**2**	**3**
initValue	212	214	230	182
	4	**5**	**6**	**7**
initValue	212	214	230	182

Table 87B

Initialization variable	ctxIdx of **cu_cbf**		
	0	**1**	**2**
initValue	154	95	94

Table 88B

Initialization variable	ctxIdx of **abs_mvd_greater0_flag**		
	0	**1**	**2**
initValue	154	155	169

Table 89B

Initialization variable	ctxIdx of **abs_mvd_greater1_flag**		
	0	**1**	**2**
initValue	154	198	183

Table 90B

Initialization variable	ctxIdx of tu_cbf_luma					
	0	1	2	3	4	5
initValue	154	126	155	127	140	141

Table 91B

Initialization variable	ctxIdx of tu_cbf_cb														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initValue	109	154	154	154	154	164	154	154	154	154	149	168	154	154	154

Table 92B

Initialization variable	ctxIdx of tu_cbf_cr														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initValue	109	154	154	154	154	164	154	154	154	154	149	168	154	154	154

Table 93B

Initialization variable	ctxIdx of tu_mts_flag					
	0	1	2	3	4	5
initValue	154	154	140	155	155	154
	6	7	8	9	10	11
initValue	141	141	141	126	155	154
	12	13	14	15	16	17
initValue	155	141	155	155	140	154

Table 94B

Initialization variable	ctxIdx of transform_skip_flag				
	0	1	2	3	4	5
initValue	109	42	138	46	124	61

Table 95B

Initialization variable	ctxIdx of last_sig_coeff_x_prefix																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
initValue	140	140	124	140	126	109	140	141	125	94	111	127	111	140	93	141	186	141
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
initValue	125	197	123	108	62	125	110	109	111	125	123	111	111	95	123	140	126	125
	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
initValue	95	169	125	140	110	124	152	138	123	92	111	125	124	111	111	109	111	111
	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68			
initValue	125	109	140	126	111	111	139	140	111	125	95	138	123	109	63			

Table 96B

Initialization variable	ctxIdx of last_sig_coeff_y_prefix																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
initValue	110	110	109	125	111	123	111	141	95	108	111	142	111	95	63	140	157	141
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
initValue	110	152	108	123	77	110	95	94	125	110	123	140	111	95	123	125	111	110
	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
initValue	95	154	125	111	95	94	137	108	93	92	125	110	139	125	125	109	111	111
	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68			
initValue	110	109	140	126	110	110	154	140	111	125	109	181	108	94	122			

Table 97B

Initialization variable	ctxIdx of coded_sub_block_flag											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
initValue	107	158	105	126	121	141	105	155	106	170	91	140

Table 98B

Initialization variable	ctxIdx of sig_coeff_flag														
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
initValue	105	138	153	154	125	111	105	139	154	155	155	127	137	185	169
	15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
initValue	185	171	159	104	168	168	169	140	141	167	215	155	172	171	158
	30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**
initValue	108	157	158	158	218	189	123	191	159	190	205	236	79	223	253
	45	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
initValue	223	223	253	152	173	157	187	189	251	170	223	223	253	223	223
	60	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**70**	**71**	**72**	**73**	**74**
initValue	107	143	205	188	233	205	63	251	191	253	206	252	62	223	223
	75	**76**	**77**	**78**	**79**	**80**	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**
initValue	223	223	223	166	174	159	247	188	189	168	223	223	223	238	223
	90	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**	**103**	**104**
initValue	119	152	167	168	183	140	134	182	168	183	169	185	166	228	183
	105	**106**	**107**	**108**	**109**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**
initValue	198	184	156	134	168	168	169	169	170	196	244	184	244	200	172
	120	**121**	**122**	**123**	**124**	**125**	**126**	**127**	**128**	**129**	**130**	**131**	**132**	**133**	**134**
initValue	123	142	202	172	172	203	138	188	233	203	203	191	139	223	223
	135	**136**	**137**	**138**	**139**	**140**	**141**	**142**	**143**	**144**	**145**	**146**	**147**	**148**	**149**
initValue	223	223	223	168	156	173	201	157	203	198	223	223	223	223	223
	150	**151**	**152**	**153**	**154**	**155**	**156**	**157**	**158**	**159**	**160**	**161**	**162**	**163**	**164**
initValue	123	187	191	173	173	248	138	191	191	191	203	191	196	223	223
	165	**166**	**167**	**168**	**169**	**170**	**171**	**172**	**173**	**174**	**175**	**176**	**177**	**178**	**179**
initValue	223	223	223	167	156	237	158	188	205	182	223	223	223	223	223
	180	**181**	**182**	**183**	**184**	**185**	**186**	**187**	**188**	**189**	**190**	**191**	**192**	**193**	**194**
initValue	105	152	167	153	168	169	104	167	182	183	183	170	209	213	183
	195	**196**	**197**	**198**	**199**	**200**	**201**	**202**	**203**	**204**	**205**	**206**	**207**	**208**	**209**
initValue	183	169	185	148	167	153	168	154	140	166	199	183	199	199	172
	210	**211**	**212**	**213**	**214**	**215**	**216**	**217**	**218**	**219**	**220**	**221**	**222**	**223**	**224**
initValue	152	127	173	201	187	173	197	203	188	217	188	189	182	223	223
	225	**226**	**227**	**228**	**229**	**230**	**231**	**232**	**233**	**234**	**235**	**236**	**237**	**238**	**239**
initValue	223	223	223	182	171	143	158	172	202	168	223	223	223	223	223
	240	**241**	**242**	**243**	**244**	**245**	**246**	**247**	**248**	**249**	**250**	**251**	**252**	**253**	**254**
initValue	137	142	190	188	202	189	241	191	191	189	189	190	195	223	223
	255	**256**	**257**	**258**	**259**	**260**	**261**	**262**	**263**	**264**	**265**	**266**	**267**	**268**	**269**
initValue	223	223	223	167	200	175	188	174	175	196	223	223	223	223	223

Table 99B

Initialization variable	ctxIdx of par_level_flag															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	121	135	137	138	153	153	136	123	138	153	139	152	153	153	139	139
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	138	168	139	154	139	136	121	167	168	138	153	137	139	154	139	154
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	106	134	151	152	138	168	120	137	138	153	139	136	138	153	168	139
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
initValue	137	153	168	139	139	120	150	152	153	153	153	166	168	168	139	154
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
initValue	91	104	136	152	153	153	105	137	167	153	168	121	167	153	168	139
	80	91	82	83	84	85	86	87	88	89	90	91	92	93	94	95
initValue	151	153	139	168	154	135	135	152	138	153	124	151	168	169	153	139

Table 100B

Initialization variable	ctxIdx of rem_abs_gt1_flag															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	90	41	149	121	122	123	58	105	92	108	109	104	92	123	109	124
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	151	138	139	154	140	165	11	120	122	137	138	75	106	138	154	155
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	88	0	102	149	150	152	101	103	150	151	138	102	105	122	167	153
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
initValue	90	107	123	153	154	0	0	105	151	107	93	103	136	138	154	125
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
initValue	30	0	102	104	106	152	57	44	120	136	123	87	134	151	152	153
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
initValue	89	121	152	153	125	102	101	90	107	122	93	118	121	153	125	140

Table 101B

Initialization variable	ctxIdx of rem_abs_gt2_flag															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	135	120	108	153	139	140	151	153	139	125	140	123	154	140	155	126
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	139	140	170	156	142	165	151	153	154	125	126	168	155	186	172	143
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	118	101	137	138	139	140	149	138	139	154	155	136	153	154	140	170
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
initValue	152	139	140	155	186	132	136	153	154	140	155	167	155	156	142	173
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
initValue	89	132	151	138	124	125	119	152	153	154	140	135	153	139	169	155
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
initValue	151	168	169	170	171	102	164	138	139	154	140	181	155	171	157	143

Table 102B

Initialization variable	ctxIdx of mtx_idx					
	0	1	2	3	4	5
initValue	167	123	138	167	153	138

Table 103B

[0122] In this manner, video encoder 200 represents an example of a device configured to determine an initial probability state of bin of a syntax element and entropy encode the bin of the syntax element using the determined initial probability state.

[0123] FIG. 6 is a block diagram illustrating an example of a video decoder that may be configured to decode video data according to one or more techniques of this disclosure. In one example, video decoder 400 may be configured to reconstruct video data based on one or more of the techniques described above. That is, video decoder 400 may operate in a reciprocal manner to video encoder 200 described above. Video decoder 400 may be configured to perform intra prediction decoding and inter prediction

decoding and, as such, may be referred to as a hybrid decoder. In the example illustrated in FIG. 6 video decoder 400 includes an entropy decoding unit 402, inverse quantization unit 404, inverse transformation processing unit 406, intra prediction processing unit 408, inter prediction processing unit 410, summer 412, filter unit 414, and reference buffer 416. Video decoder 400 may be configured to decode video data in a manner consistent with a video encoding system, which may implement one or more aspects of a video coding standard. It should be noted that although example video decoder 400 is illustrated as having distinct functional blocks, such an illustration is for descriptive purposes and does not limit video decoder 400 and/or subcomponents thereof to a particular hardware or software architecture. Functions of video decoder 400 may be realized using any combination of hardware, firmware, and/or software implementations.

[0124] As illustrated in FIG. 6, entropy decoding unit 402 receives an entropy encoded bitstream. Entropy decoding unit 402 may be configured to decode quantized syntax elements and quantized coefficients from the bitstream according to a process reciprocal to an entropy encoding process. Entropy decoding unit 402 may be configured to perform entropy decoding according any of the entropy coding techniques described above. Entropy decoding unit 402 may parse an encoded bitstream in a manner consistent with a video coding standard. Video decoder 400 may be configured to parse an encoded bitstream where the encoded bitstream is generated based on the techniques described above.

[0125] FIG. 7 is a block diagram illustrating an example entropy decoding unit that may implement one or more of the techniques described in this disclosure. Entropy decoding unit 500 receives an entropy encoded bitstream and decodes syntax elements from the bitstream. As illustrated in FIG. 7, entropy decoding unit 500 includes a binary arithmetic decoding module 502, a context modeling unit 504 and a binarization unit 506. Entropy decoding unit 500 may perform reciprocal functions to entropy encoding unit 300 described above with respect to FIG. 5.

[0126] As shown in FIG. 7, context modeling unit 508 and a binarization unit 506 receive a request for a syntax element value. Context modeling unit 504 determines a context index for the syntax element. Further, context modeling unit 504 updates a context index based on the determination made by binary arithmetic decoding module 502, for example, according to the probability estimate techniques described above. Binary arithmetic decoding module 502 receives n bits from the bitstream, i.e., the arithmetic code, and outputs a sequence of parsed bins based on the arithmetic code and the calculated sub-intervals. Binarization unit 506 determines possible valid binarization values for a syntax element and uses a bin matching function to determine if a series parsed bin values corresponds to a valid value for the syntax element. When a series

bin values corresponds to a valid value for the syntax element, the value of the syntax element is output. That is, entropy decoding unit 500 is configured to determine the value of a bin based on the current sub-interval and bits from the bitstream, where the current sub-interval is determined based on techniques described herein, for example, probability estimation techniques described herein.

[0127] In this manner, video decoder 400 represents an example of a device configured to determine an initial probability state of bin of a syntax element and entropy decode the bin of the syntax element using the determined initial probability state.

[0128] Referring again to FIG. 6, inverse quantization unit 404 receives quantized transform coefficients (i.e., level values) and quantization parameter data from entropy decoding unit 402. Quantization parameter data may include any and all combinations of delta QP values and/or quantization group size values and the like described above. Video decoder 400 and/or inverse quantization unit 404 may be configured to determine QP values used for inverse quantization based on values signaled by a video encoder and/or through video properties and/or coding parameters. That is, inverse quantization unit 404 may operate in a reciprocal manner to coefficient quantization unit 206 described above. For example, inverse quantization unit 404 may be configured to infer predetermined values), allowed quantization group sizes, and the like, according to the techniques described above. Inverse quantization unit 404 may be configured to apply an inverse quantization. Inverse transform processing unit 406 may be configured to perform an inverse transformation to generate reconstructed residual data. The techniques respectively performed by inverse quantization unit 404 and inverse transform processing unit 406 may be similar to techniques performed by inverse quantization/transform processing unit 208 described above. Inverse transform processing unit 406 may be configured to apply an inverse DCT, an inverse DST, an inverse integer transform, Non-Separable Secondary Transform (NSST), or a conceptually similar inverse transform processes to the transform coefficients in order to produce residual blocks in the pixel domain. Further, as described above, whether a particular transform (or type of particular transform) is performed may be dependent on an intra prediction mode. As illustrated in FIG. 6, reconstructed residual data may be provided to summer 412. Summer 412 may add reconstructed residual data to a predictive video block and generate reconstructed video data. A predictive video block may be determined according to a predictive video technique (i.e., intra prediction and inter frame prediction).

[0129] Intra prediction processing unit 408 may be configured to receive intra prediction syntax elements and retrieve a predictive video block from reference buffer 416. Reference buffer 416 may include a memory device configured to store one or more frames of video data. Intra prediction syntax elements may identify an intra prediction

mode, such as the intra prediction modes described above. In one example, intra prediction processing unit 308 may reconstruct a video block using according to one or more of the intra prediction coding techniques described herein. Inter prediction processing unit 410 may receive inter prediction syntax elements and generate motion vectors to identify a prediction block in one or more reference frames stored in reference buffer 416. Inter prediction processing unit 410 may produce motion compensated blocks, possibly performing interpolation based on interpolation filters. Identifiers for interpolation filters to be used for motion estimation with sub-pixel precision may be included in the syntax elements. Inter prediction processing unit 410 may use interpolation filters to calculate interpolated values for sub-integer pixels of a reference block. Filter unit 414 may be configured to perform filtering on reconstructed video data. For example, filter unit 414 may be configured to perform deblocking and/or SAO filtering, as described above with respect to filter unit 216. Further, it should be noted that in some examples, filter unit 414 may be configured to perform proprietary discretionary filter (e.g., visual enhancements). As illustrated in FIG. 6, a reconstructed video block may be output by video decoder 400.

[0130] In one or more examples, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored on or transmitted over as one or more instructions or code on a computer-readable medium and executed by a hardware-based processing unit. Computer-readable media may include computer-readable storage media, which corresponds to a tangible medium such as data storage media, or communication media including any medium that facilitates transfer of a computer program from one place to another, e.g., according to a communication protocol. In this manner, computer-readable media generally may correspond to (1) tangible computer-readable storage media which is non-transitory or (2) a communication medium such as a signal or carrier wave. Data storage media may be any available media that can be accessed by one or more computers or one or more processors to retrieve instructions, code and/or data structures for implementation of the techniques described in this disclosure. A computer program product may include a computer-readable medium.

[0131] By way of example, and not limitation, such computer-readable storage media can comprise RAM, ROM, EEPROM, CD-ROM or other optical disk storage, magnetic disk storage, or other magnetic storage devices, flash memory, or any other medium that can be used to store desired program code in the form of instructions or data structures and that can be accessed by a computer. Also, any connection is properly termed a computer-readable medium. For example, if instructions are transmitted from a website, server, or other remote source using a coaxial cable, fiber optic cable, twisted pair, digital subscriber line (DSL), or wireless technologies such as infrared,

radio, and microwave, then the coaxial cable, fiber optic cable, twisted pair, DSL, or wireless technologies such as infrared, radio, and microwave are included in the definition of medium. It should be understood, however, that computer-readable storage media and data storage media do not include connections, carrier waves, signals, or other transitory media, but are instead directed to non-transitory, tangible storage media. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk and Blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above should also be included within the scope of computer-readable media.

[0132] Instructions may be executed by one or more processors, such as one or more digital signal processors (DSPs), general purpose microprocessors, application specific integrated circuits (ASICs), field programmable logic arrays (FPGAs), or other equivalent integrated or discrete logic circuitry. Accordingly, the term "processor," as used herein may refer to any of the foregoing structure or any other structure suitable for implementation of the techniques described herein. In addition, in some aspects, the functionality described herein may be provided within dedicated hardware and/or software modules configured for encoding and decoding, or incorporated in a combined codec. Also, the techniques could be fully implemented in one or more circuits or logic elements.

[0133] The techniques of this disclosure may be implemented in a wide variety of devices or apparatuses, including a wireless handset, an integrated circuit (IC) or a set of ICs (e.g., a chip set). Various components, modules, or units are described in this disclosure to emphasize functional aspects of devices configured to perform the disclosed techniques, but do not necessarily require realization by different hardware units. Rather, as described above, various units may be combined in a codec hardware unit or provided by a collection of interoperative hardware units, including one or more processors as described above, in conjunction with suitable software and/or firmware.

[0134] Moreover, each functional block or various features of the base station device and the terminal device used in each of the aforementioned embodiments may be implemented or executed by a circuitry, which is typically an integrated circuit or a plurality of integrated circuits. The circuitry designed to execute the functions described in the present specification may comprise a general-purpose processor, a digital signal processor (DSP), an application specific or general application integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic devices, discrete gates or transistor logic, or a discrete hardware component, or a combination thereof. The general-purpose processor may be a microprocessor, or alternatively, the processor may be a conventional processor, a controller, a microcontroller

or a state machine. The general-purpose processor or each circuit described above may be configured by a digital circuit or may be configured by an analogue circuit. Further, when a technology of making into an integrated circuit superseding integrated circuits at the present time appears due to advancement of a semiconductor technology, the integrated circuit by this technology is also able to be used.

[0135] Various examples have been described. These and other examples are within the scope of the following claims.

[0136] <Cross Reference>

This Nonprovisional application claims priority under 35 U.S.C. § 119 on provisional Application No. 62/757,054 on November 7, 2018, No. 62/757,669 on November 8, 2018, No. 62/790,763 on January 10, 2019, No. 62/803,239 on February 8, 2019, No. 62/815,980 on March 8, 2019, No. 62/803,734 on November 2, 2019 the entire contents of which are hereby incorporated by reference.

[0137] WHAT IS CLAIMED IS:

Claims

[Claim 1] A method of entropy encoding, the method comprising:
determining an initial probability state of a bin of a syntax element; and
entropy encoding the bin of the syntax element using the determined initial probability state.

[Claim 2] A method of entropy decoding, the method comprising:
determining an initial probability state of a bin of a syntax element; and
entropy decoding the bin of the syntax element using the determined initial probability state.

[Claim 3] The method of any of claims 1 or 2 where an initial probability state of a bin of a syntax element includes one of the initial probability states specified herein.

[Claim 4] The method of any of claims 1 or 2 where determining an initial probability state of a bin of a syntax element includes determining the initial probability state based on one of the tables specified herein.

[Claim 5] A device for coding video data, the device comprising one or more processors configured to perform any and all combinations of the steps of claims 1-4.

[Claim 6] The device of claim 5, wherein the device includes a video encoder.

[Claim 7] The device of claim 5, wherein the device includes a video decoder.

[Claim 8] A system comprising:
the device of claim 6; and
the device of claim 7.

[Claim 9] An apparatus for coding video data, the apparatus comprising means for performing any and all combinations of the steps of claims 1-4.

[Claim 10] A non-transitory computer-readable storage medium comprising instructions stored thereon that, when executed, cause one or more processors of a device for coding video data to perform any and all combinations of the steps of claims 1-4.

[Fig. 1]

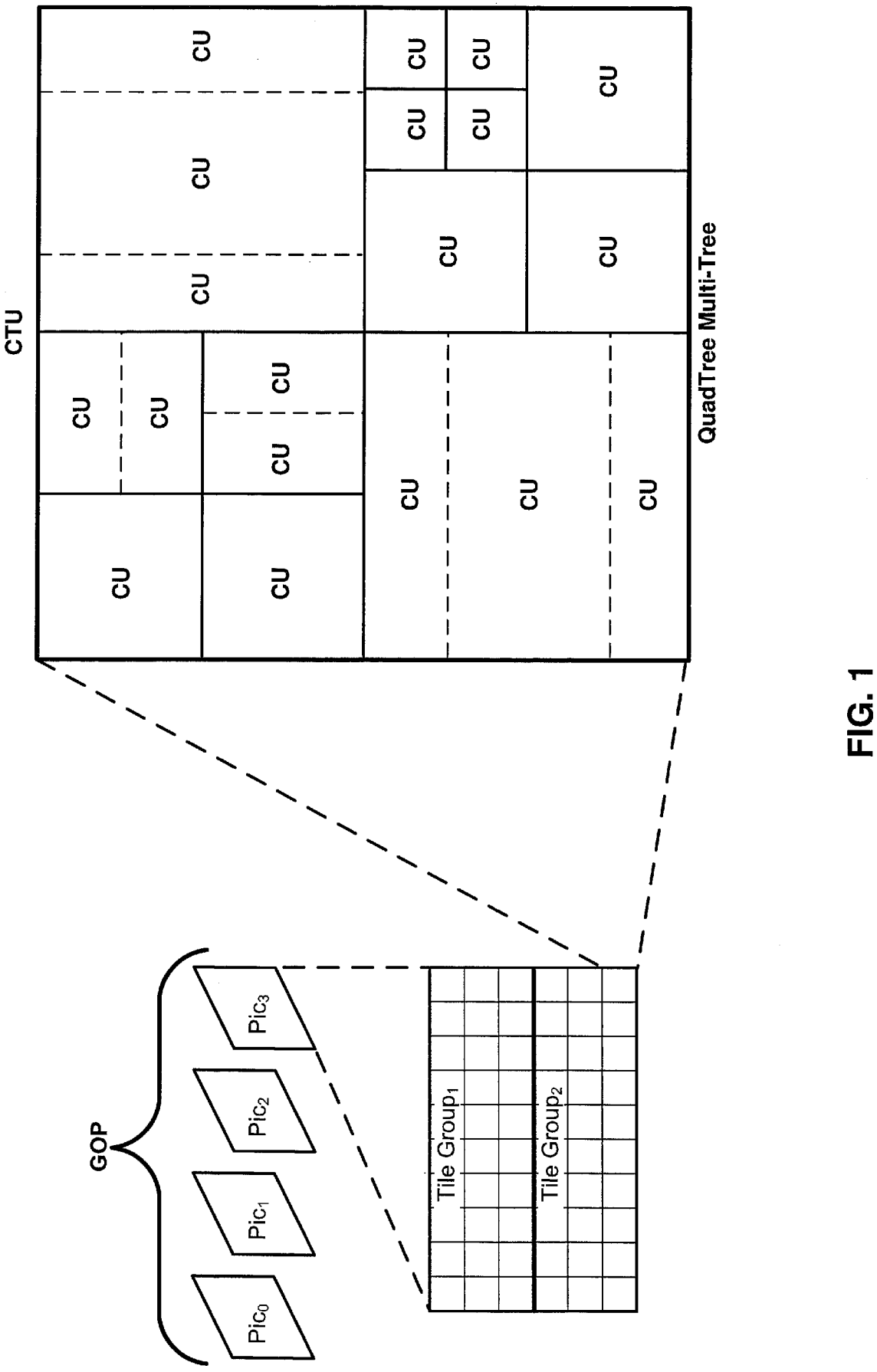
CTU
CU
QuadTree Multi-Tree
GOP
Pic0
Pic1
Pic2
Pic3
Tile Group1
Tile Group2

FIG. 1

[Fig. 2A]

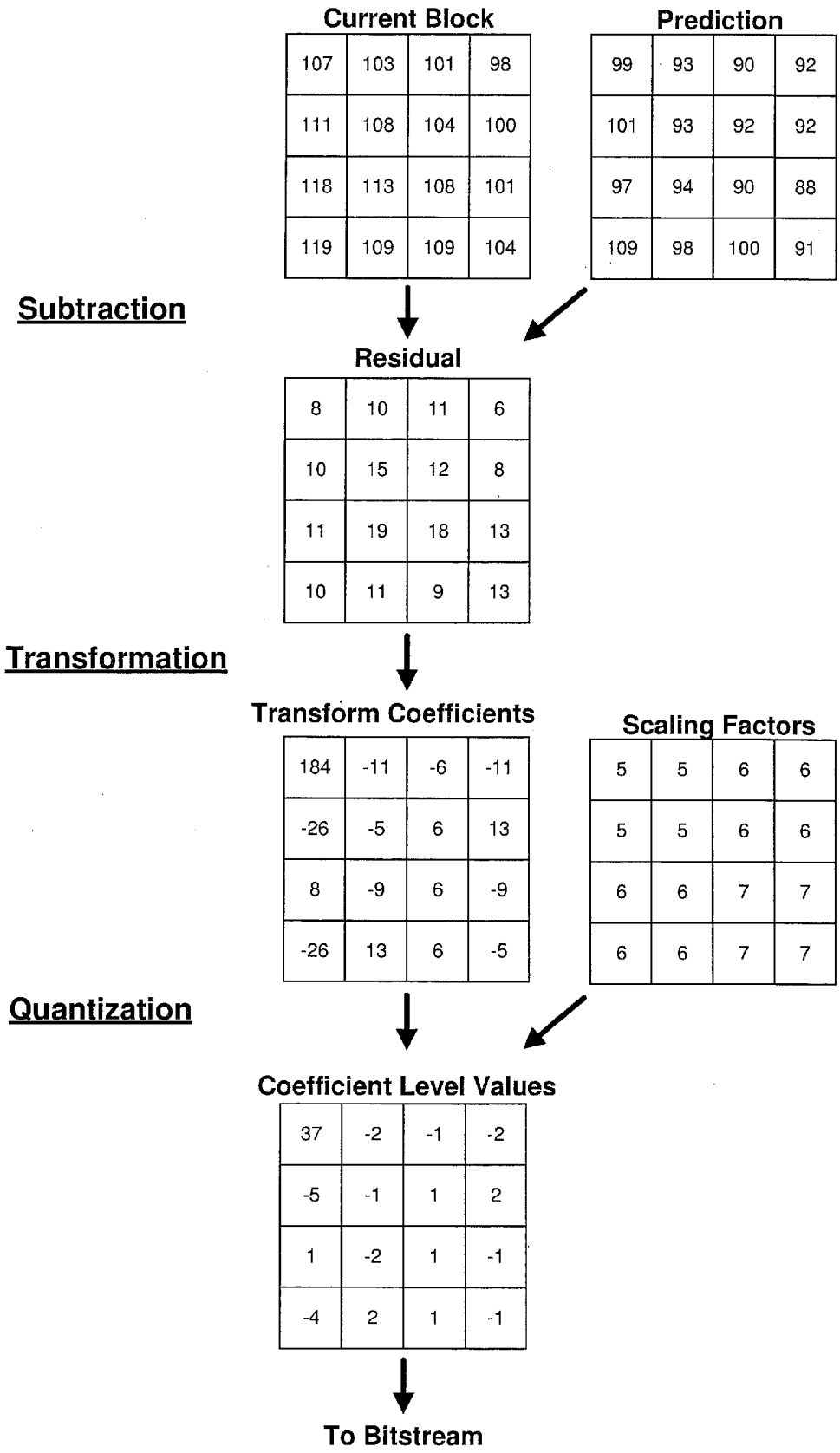
Current Block
107 103 101 98
111 108 104 100
118 113 108 101
119 109 109 104
Prediction
99 93 90 92
101 93 92 92
97 94 90 88
109 98 100 91
Subtraction
Residual
8 10 11 6
10 15 12 8
11 19 18 13
10 11 9 13
Transformation
Transform Coefficients
184 -11 -6 -11
-26 -5 6 13
8 -9 6 -9
-26 13 6 -5
Scaling Factors
5 5 6 6
5 5 6 6
6 6 7 7
6 6 7 7
Quantization
Coefficient Level Values
37 -2 -1 -2
-5 -1 1 2
1 -2 1 -1
-4 2 1 -1
To Bitstream

FIG. 2A

[Fig. 2B]

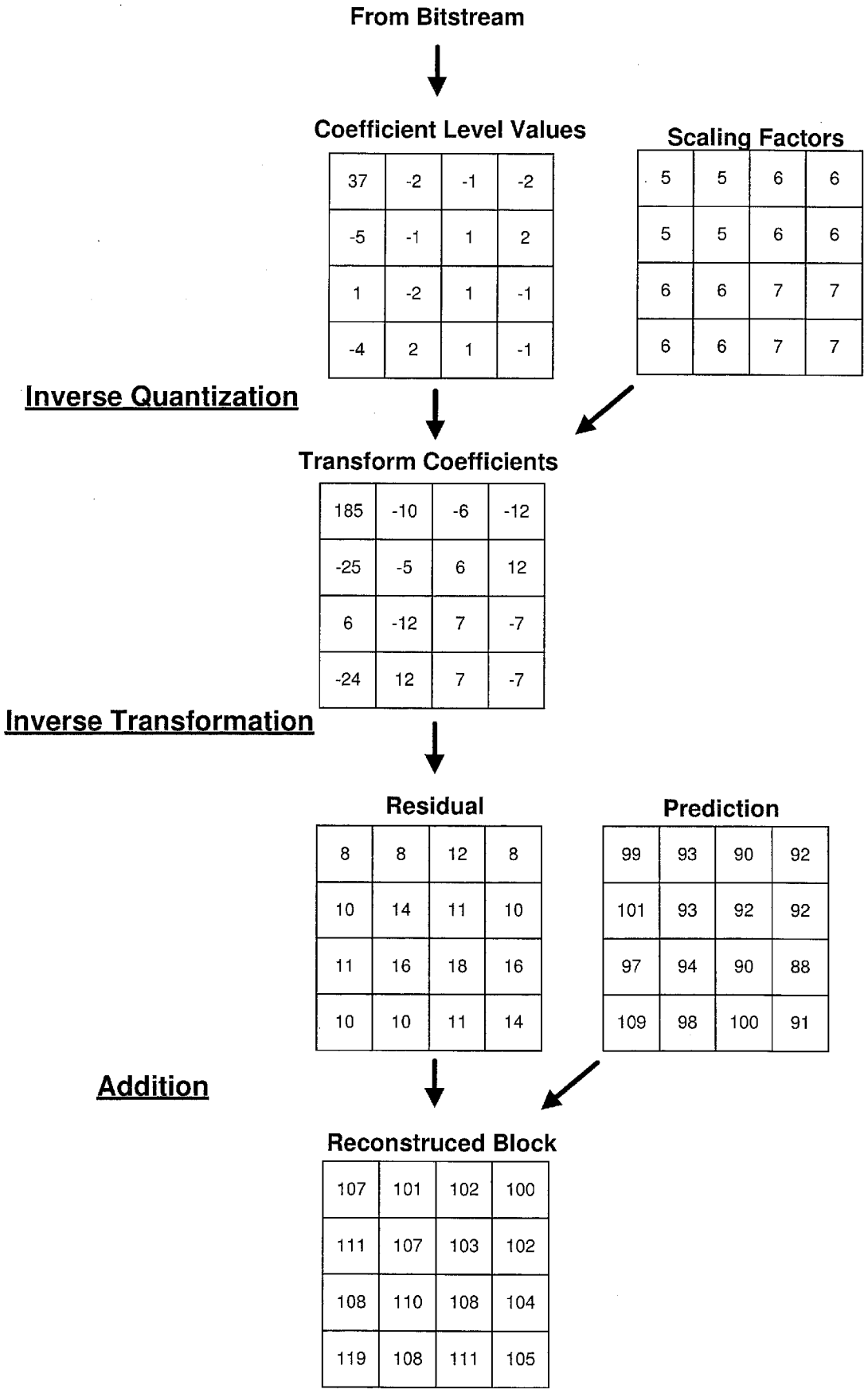
From Bitstream
Coefficient Level Values
37 -2 -1 -2
-5 -1 1 2
1 -2 1 -1
-4 2 1 -1
Scaling Factors
5 5 6 6
5 5 6 6
6 6 7 7
6 6 7 7
Inverse Quantization
Transform Coefficients
185 -10 -6 -12
-25 -5 6 12
6 -12 7 -7
-24 12 7 -7
Inverse Transformation
Residual
8 8 12 8
10 14 11 10
11 16 18 16
10 10 11 14
Prediction
99 93 90 92
101 93 92 92
97 94 90 88
109 98 100 91
Addition
Reconstruced Block
107 101 102 100
111 107 103 102
108 110 108 104
119 108 111 105

FIG. 2B

[Fig. 3]

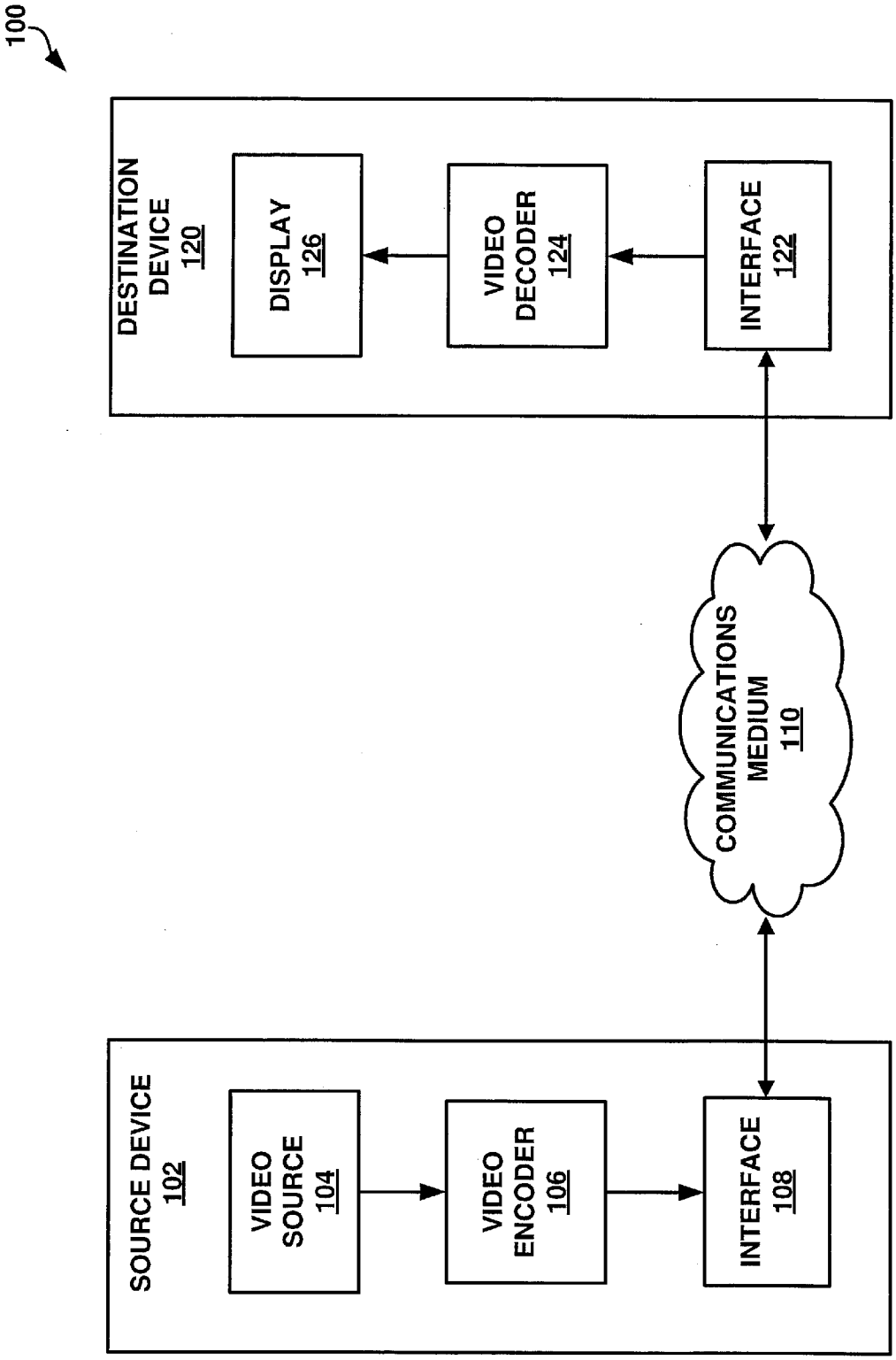
100
SOURCE DEVICE 102
VIDEO SOURCE 104
VIDEO ENCODER 106
INTERFACE 108
COMMUNICATIONS MEDIUM 110
DESTINATION DEVICE 120
INTERFACE 122
VIDEO DECODER 124
DISPLAY 126

FIG. 3

[Fig. 4]

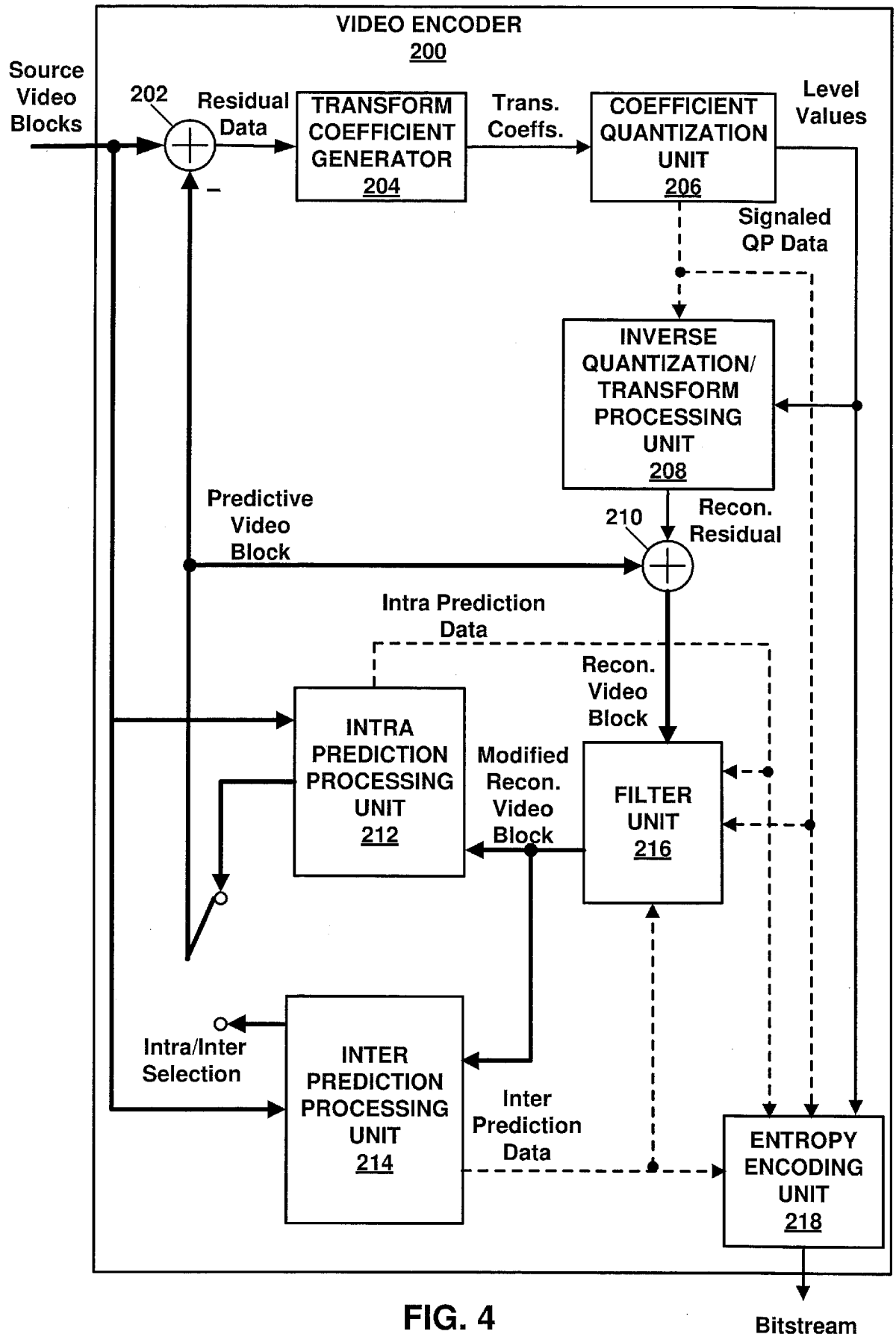
VIDEO ENCODER
200
Source Video Blocks
202
Residual Data
TRANSFORM COEFFICIENT GENERATOR
204
Trans. Coeffs.
COEFFICIENT QUANTIZATION UNIT
206
Level Values
Signaled QP Data
INVERSE QUANTIZATION/ TRANSFORM PROCESSING UNIT
208
Predictive Video Block
210
Recon. Residual
Intra Prediction Data
Recon. Video Block
INTRA PREDICTION PROCESSING UNIT
212
Modified Recon. Video Block
FILTER UNIT
216
Intra/Inter Selection
INTER PREDICTION PROCESSING UNIT
214
Inter Prediction Data
ENTROPY ENCODING UNIT
218
Bitstream

FIG. 4

[Fig. 5]

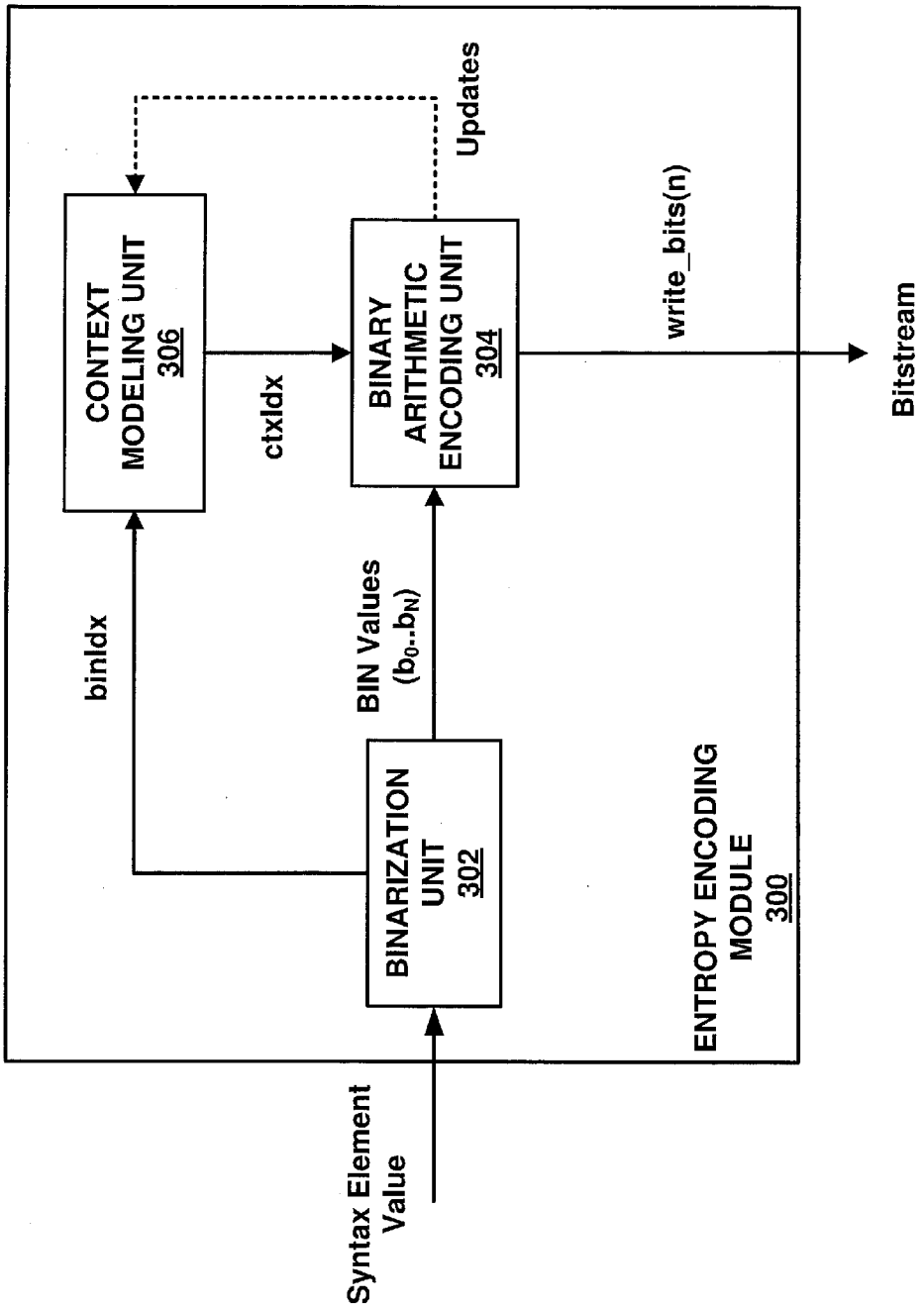
Updates
CONTEXT MODELING UNIT 306
BINARY ARITHMETIC ENCODING UNIT 304
write_bits(n)
Bitstream
ctxIdx
binIdx
BIN Values ($b_0..b_N$)
BINARIZATION UNIT 302
ENTROPY ENCODING MODULE 300
Syntax Element Value

FIG. 5

[Fig. 6]

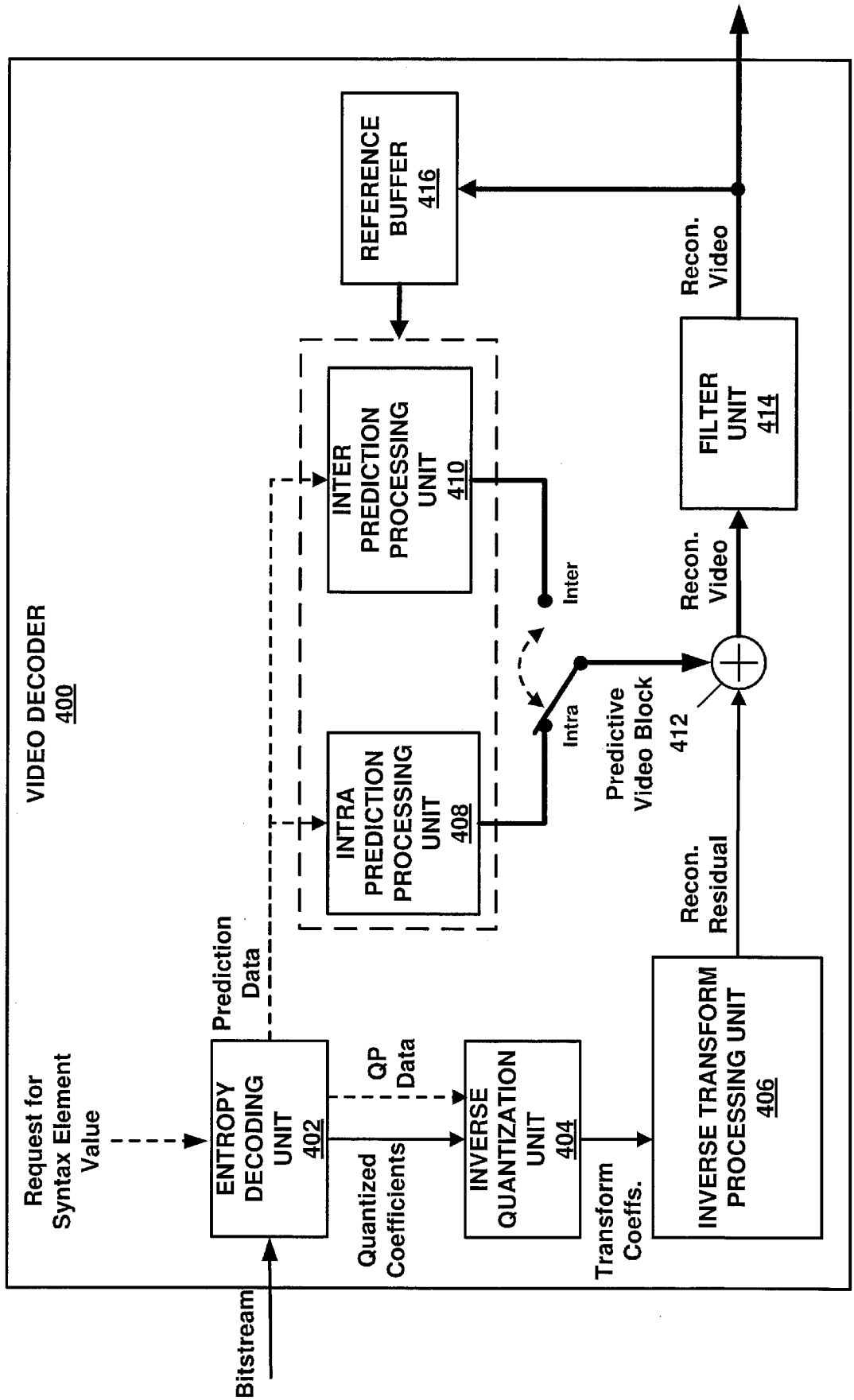
VIDEO DECODER
400
ENTROPY DECODING UNIT
402
INVERSE QUANTIZATION UNIT
404
INVERSE TRANSFORM PROCESSING UNIT
406
INTRA PREDICTION PROCESSING UNIT
408
INTER PREDICTION PROCESSING UNIT
410
FILTER UNIT
414
REFERENCE BUFFER
416
Predictive Video Block
412
Request for Syntax Element Value
Bitstream
Quantized Coefficients
QP Data
Transform Coeffs.
Recon. Residual
Prediction Data
Intra
Inter
Recon. Video
Recon. Video

FIG. 6

[Fig. 7]

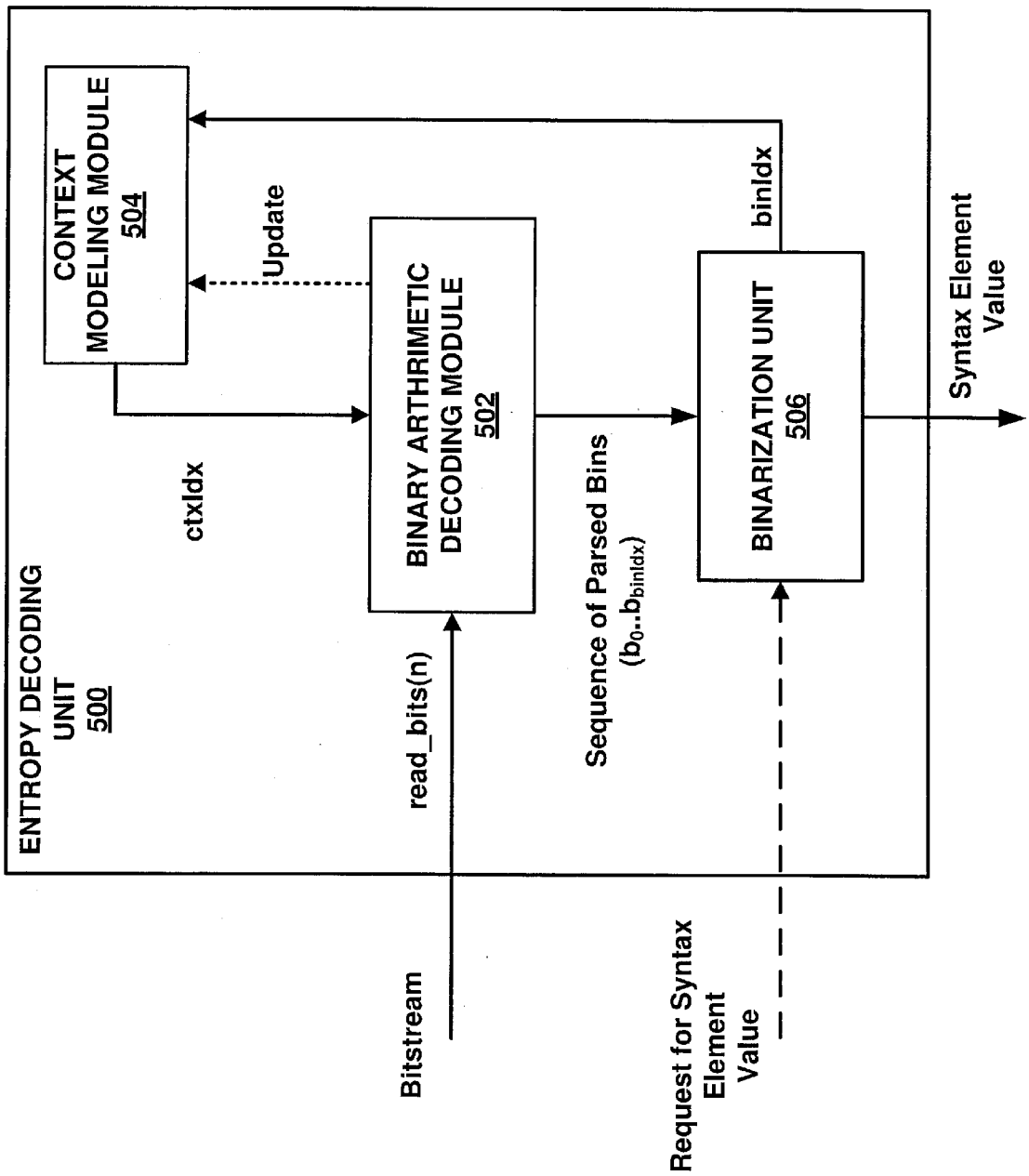
ENTROPY DECODING UNIT 500
CONTEXT MODELING MODULE 504
BINARY ARTHRIMETIC DECODING MODULE 502
BINARIZATION UNIT 506
Update
ctxIdx
binIdx
read_bits(n)
Sequence of Parsed Bins ($b_0..b_{binIdx}$)
Syntax Element Value
Bitstream
Request for Syntax Element Value

FIG. 7

[Fig. 8]

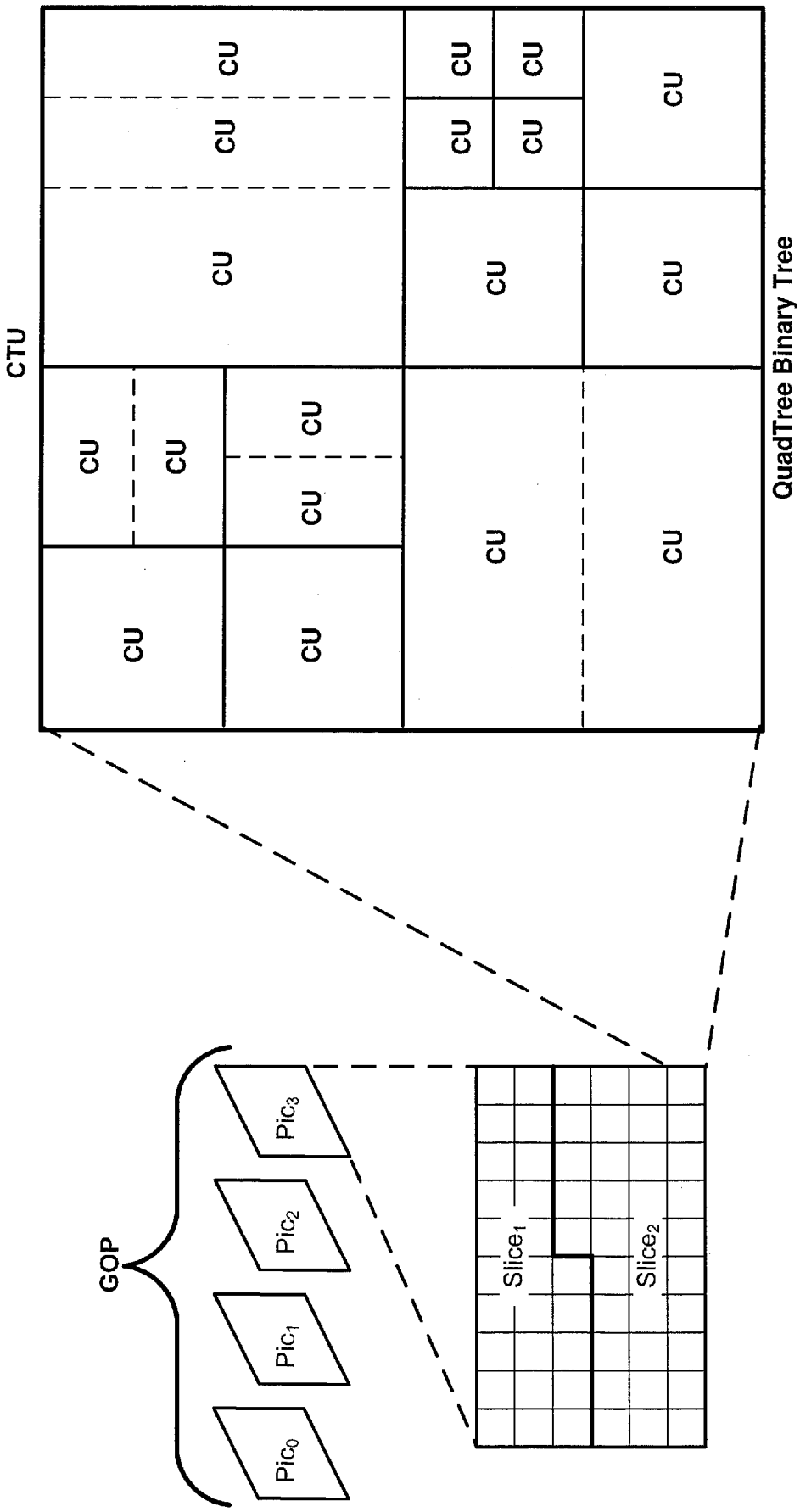
CTU
CU
QuadTree Binary Tree
GOP
Pic0
Pic1
Pic2
Pic3
Slice1
Slice2

FIG. 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/JP2019/043637

A. CLASSIFICATION OF SUBJECT MATTER

H04N 19/70(2014.01)i
FI: H04N19/70

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

H04N19/00 - H04N19/98

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Published examined utility model applications of Japan 1922-1996
Published unexamined utility model applications of Japan 1971-2020
Registered utility model specifications of Japan 1996-2020
Published registered utility model applications of Japan 1994-2020

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	ITU-T, H.265(02/2018) SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS, T-REC-H.265-201802-S!!PDF-E.pdf[online], 2018.05.11, https://www.itu.int/rec/T-REC-H.265-201802-S/en pp.203-216	1-10

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
23 January 2020	**04 February 2020**
Name and mailing address of the ISA/JP **Japan Patent Office** **3-4-3, Kasumigaseki, Chiyoda-ku, Tokyo** **100-8915, Japan**	Authorized officer **KATAOKA, Toshinobu 5C 4881** Telephone No. **+81-3-3581-1101 Ext. 3541**