# Syntax and semantics

## Syntax in tabular form

### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, log2CbSize ) { | Descriptor |
| CurrCbAddrTS = MinCbAddrZS[ x0 >> Log2MinCbSize ][ y0 >> Log2MinCbSize ] |  |
| if( slice\_type != I ) |  |
| **skip\_flag[** x0 **][** y0 **]** | ae(v) |
| if( skip\_flag[ x0 ][ y0 ] ) |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| else if( slice\_type != I | | log2CbSize = = Log2MinCbSize ) { |  |
| if( slice\_type != I ) |  |
| **pred\_mode\_flag** | ae(v) |
| if( PredMode != MODE\_INTRA | | log2CbSize = = Log2MinCbSize ) |  |
| **part\_mode** | ae(v) |
| x1 = x0 + ( ( 1 << log2CbSize ) >> 1 ) |  |
| y1 = y0 + ( ( 1 << log2CbSize ) >> 1 ) |  |
| x2 = x1 − ( ( 1 << log2CbSize ) >> 2 ) |  |
| y2 = y1 − ( ( 1 << log2CbSize ) >> 2 ) |  |
| x3 = x1 + ( ( 1 << log2CbSize ) >> 2 ) |  |
| y3 = y1 + ( ( 1 << log2CbSize ) >> 2 ) |  |
| if( PartMode = = PART\_2Nx2N ) |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| else if( PartMode = = PART\_2NxN ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y1 , log2CbSize ) |  |
| } else if( PartMode = = PART\_Nx2N ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x1, y0 , log2CbSize ) |  |
| } else if( PartMode = = PART\_2NxnU ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y2 , log2CbSize ) |  |
| } else if( PartMode = = PART\_2NxnD ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y3 , log2CbSize ) |  |
| } else if( PartMode = = PART\_nLx2N ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x2, y0 , log2CbSize ) |  |
| } else if( PartMode = = PART\_nRx2N ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x3, y0 , log2CbSize ) |  |
| } else { /\* PART\_NxN \*/ |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x1, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y1 , log2CbSize ) |  |
| prediction\_unit( x1, y1 , log2CbSize ) |  |
| } |  |
| if( !pcm\_flag ) { |  |
| if( PredMode != MODE\_INTRA &&   !(PartMode = = PART\_2Nx2N && merge\_flag[x0][y0]) ) |  |
| **no\_residual\_data\_flag** | ae(v) |
| if( !no\_residual\_data\_flag ) { |  |
| MaxTrafoDepth = ( PredMode = = MODE\_INTRA ?   max\_transform\_hierarchy\_depth\_intra + IntraSplitFlag :   max\_transform\_hierarchy\_depth\_inter ) |  |
| transform\_tree( x0, y0, x0, y0, log2CbSize, log2CbSize, log2CbSize, 0, 0 ) |  |
| } |  |
| } |  |
| } |  |
| } |  |

### Transform tree syntax

|  |  |
| --- | --- |
| transform\_tree( x0, y0, xC, yC, log2CbSize, log2TrafoWidth, log2TrafoHeight,  trafoDepth, blkIdx ) { | Descriptor |
| log2TrafoSize = ( log2TrafoWidth + log2TrafoHeight ) >> 1 |  |
| xBase = x0 − ( x0 & ( 1 << log2TrafoWidth ) ) |  |
| yBase = y0 − ( y0 & ( 1 << log2TrafoHeight ) ) |  |
| if( log2TrafoSize <= Log2MaxTrafoSize &&   log2TrafoSize > Log2MinTrafoSize &&  trafoDepth < MaxTrafoDepth && !(IntraSplitFlag && trafoDepth = = 0) ) |  |
| **split\_transform\_flag**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| if( log2TrafoSize <= Log2MaxTrafoSize ) { |  |
| firstChromaCbf = ( log2TrafoSize = = Log2MaxTrafoSize | |  trafoDepth = = 0 ) ? 1 : 0 |  |
| if( firstChromaCbf | | log2TrafoSize > Log2MinTrafoSize ) { |  |
| if( firstChromaCbf | | cbf\_cb[ xBase ][ yBase ][ trafoDepth − 1 ] ) { |  |
| if( blkIdx < 3 | | log2TrafoSize = = Log2MaxTrafoSize | |   cbf\_cb[ xBase ][ yBase ][ trafoDepth ] | |   cbf\_cb[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] | |   cbf\_cb[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] ) |  |
| **cbf\_cb**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| } |  |
| if( firstChromaCbf | | cbf\_cr[ xBase ][ yBase ][ trafoDepth − 1 ] ) { |  |
| if( blkIdx < 3 | | log2TrafoSize = = Log2MaxTrafoSize | |   cbf\_cr[ xBase ][ yBase ][ trafoDepth ] | |   cbf\_cr[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] | |   cbf\_cr[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] ) |  |
| **cbf\_cr**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| } |  |
| } |  |
| } |  |
| if( split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] ) { |  |
| if( InterTUSplitDirection = = 2 ) { |  |
| x1 = x0 + ( ( 1 << log2TrafoWidth ) >> 1 ) |  |
| y1 = y0 |  |
| x2 = x0 |  |
| y2 = y0 + ( ( 1 << log2TrafoHeight ) >> 1 ) |  |
| x3 = x1 |  |
| y3 = y2 |  |
| } else { |  |
| x1 = x0 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y1 = y0 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| x2 = x1 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y2 = y1 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| x3 = x2 + ( ( 1 << log2TrafoWidth ) >> 2 ) \* InterTUSplitDirection |  |
| y3 = y2 + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − InterTUSplitDirection ) |  |
| log2TrafoHeight = log2TrafoHeight + 2 \* InterTUSplitDirection − 1 |  |
| log2TrafoWidth = log2TrafoWidth − 2 \* InterTUSplitDirection + 1 |  |
| } |  |
| transform\_tree( x0, y0, x0, y0, log2CbSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 0 ) |  |
| transform\_tree( x1, y1, x0, y0, log2CbSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 1 ) |  |
| transform\_tree( x2, y2, x0, y0, log2CbSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 2 ) |  |
| transform\_tree( x3, y3, x0, y0, log2CbSize, log2TrafoWidth − 1, log2TrafoHeight − 1,   trafoDepth + 1, 3 ) |  |
| } else { |  |
| if( ( PredMode = = MODE\_INTRA | | trafoDepth != 0 | |  cbf\_cb[ x0 ][ y0 ][ trafoDepth ] | | cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) &&   ( blkIdx < 3 | | PredMode = = MODE\_INTRA | |   ( ( log2CbSize > Log2MaxTrafoSize + 1 ) && ( log2TrafoSize = = Log2MaxTrafoSize ) ) | |  cbf\_luma[ xBase ][ yBase ][ trafoDepth ] | |   cbf\_luma[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] | |   cbf\_luma[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] | |   cbf\_cb[ xBase ][ yBase ][ trafoDepth − 1] | |   cbf\_cr[ xBase ][ yBase ][ trafoDepth − 1] ) ) |  |
| **cbf\_luma**[ x0 ][ y0 ][ trafoDepth ] | ae(v) |
| transform\_unit (x0, y0, xC, yC, log2TrafoWidth, log2TrafoHeight, trafoDepth, blkIdx) |  |
| } |  |
| } |  |

[Ed. (KS): Need to be updated on second depth for non square transform tree. In 16x16 CU, second TU depth should have four square transforms in a row or column. ]

### Transform unit syntax

|  |  |
| --- | --- |
| transform\_unit( x0, y0, xC, yC, log2TrafoWidth, log2TrafoHeight, trafoDepth, blkIdx ) { | Descriptor |
| if( cbf\_luma[ x0 ][ y0 ][ trafoDepth ] | | cbf\_cb[ x0 ][ y0 ][ trafoDepth ] | |  cbf\_cr[ x0 ][ y0 ][ trafoDepth ] { |  |
| if( (max\_cu\_qp\_delta\_depth > 0 ) && !IsCuQpDeltaCoded ) { |  |
| **cu\_qp\_delta** | ae(v) |
| IsCuQpDeltaCoded = 1 |  |
| } |  |
| log2TrafoSize = ( ( log2TrafoWidth + log2TrafoHeight ) >> 1 ) |  |
| if( PredMode = = MODE\_INTRA ) { |  |
| scanIdx = ScanType[ log2TrafoSize − 2 ][ IntraPredMode ] |  |
| scanIdxC = ScanType[ log2TrafoSize − 2 ][ IntraPredModeC ] |  |
| } else { |  |
| scanIdx = 0 |  |
| scanIdxC = 0 |  |
| } |  |
| if( cbf\_luma[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( x0, y0, log2TrafoWidth, log2TrafoHeight, scanIdx, 0 ) |  |
| if( log2TrafoSize > Log2MinTrafoSize ) { |  |
| if( cbf\_cb[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( x0, y0, log2TrafoWidth − 1, log2TrafoHeight − 1, scanIdxC, 1 ) |  |
| if( cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( x0, y0, log2TrafoWidth − 1, log2TrafoHeight − 1, scanIdxC, 2 ) |  |
| } else if( blkIdx = = 3 ) { |  |
| if( cbf\_cb[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( xC, yC, log2TrafoWidth, log2TrafoHeight, scanIdxC, 1 ) |  |
| if( cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) |  |
| residual\_coding( xC, yC, log2TrafoWidth, log2TrafoHeight, scanIdxC, 2 ) |  |
| } |  |
| } |  |
| } |  |

### Residual coding syntax

|  |  |
| --- | --- |
| residual\_coding( x0, y0, log2TrafoWidth, log2TrafoHeight, scanIdx, cIdx ) { | Descriptor |
| **last\_significant\_coeff\_x\_prefix** | ae(v) |
| **last\_significant\_coeff\_y\_prefix** | ae(v) |
| if( last\_significant\_coeff\_x\_prefix > 3 ) |  |
| **last\_significant\_coeff\_x\_suffix** | ae(v) |
| if( last\_significant\_coeff\_y\_prefix > 3 ) |  |
| **last\_significant\_coeff\_y\_suffix** | ae(v) |
| numCoeff = 0 |  |
| do { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ numCoeff ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ numCoeff ][ 1 ] |  |
| numCoeff++ |  |
| } while( ( xC != LastSignificantCoeffX ) | | ( yC != LastSignificantCoeffY ) ) |  |
| numLastSubset = (numCoeff − 1) >> 4 |  |
| for( i = numLastSubset; i >= 0; i− − ) { |  |
| offset = i << 4 |  |
| if( scanIdx = = 1 && log2TrafoWidth = = 3 && log2TrafoHeight = = 3 ) { |  |
| xCG = 0 |  |
| yCG = i |  |
| } else if( scanIdx = = 2 && log2TrafoWidth = = 3 && log2TrafoHeight = = 3 ) { |  |
| xCG = i |  |
| yCG = 0 |  |
| } else { |  |
| xCG = ScanOrder[ log2TrafoWidth − 2 ][ log2TrafoHeight − 2 ][ scanIdx ][ i ][ 0 ] |  |
| yCG = ScanOrder[ log2TrafoWidth − 2 ][ log2TrafoHeight − 2 ][ scanIdx ][ i ][ 1 ] |  |
| } |  |
| implicitNonZeroCoeff = 0 |  |
| if( (i < numLastSubset) && (i > 0) ) { |  |
| **significant\_coeff\_group\_flag**[ xCG ][ yCG ] | ae(v) |
| implicitNonZeroCoeff = 1 |  |
| } |  |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( (n + offset) < (numCoeff − 1) && significant\_coeff\_group\_flag[ xCG ][ yCG ] &&   ( n > 0 | | implicitNonZeroCoeff = = 0 ) ) { |  |
| **significant\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| if( significant\_coeff\_flag[ xC ][ yC ] = = 1 ) |  |
| implicitNonZeroCoeff = 0 |  |
| } |  |
| } |  |
| firstNZPosInCG = 16 |  |
| lastNZPosInCG = −1 |  |
| numSigCoeff = 0 |  |
| firstGreater1CoeffIdx = −1 |  |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( significant\_coeff\_flag[ xC ][ yC ] ) { |  |
| if( numSigCoeff < 8 ) { |  |
| **coeff\_abs\_level\_greater1\_flag[** n **]** | ae(v) |
| numSigCoeff++ |  |
| if( coeff\_abs\_level\_greater1\_flag[ n ] && firstGreater1CoeffIdx = = −1 ) |  |
| firstGreater1CoeffIdx = n |  |
| } |  |
| if( lastNZPosInCG = = −1) |  |
| lastNZPosInCG = n |  |
| firstNZPosInCG = n |  |
| } |  |
| } |  |
| signHidden = ( lastNZPosInCG – firstNZPosInCG >= sign\_hiding\_threshold) ? 1 : 0 |  |
| if( firstGreater1CoeffIdx != −1 ) |  |
| **coeff\_abs\_level\_greater2\_flag[** firstGreater1CoeffIdx**]** | ae(v) |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( significant\_coeff\_flag[ xC ][ yC ] &&  (!sign\_data\_hiding\_flag | | !signHidden | | n != firstNZPosInCG) ) |  |
| **coeff\_sign\_flag[** n **]** | ae(v) |
| } |  |
| numSigCoeff = 0 |  |
| sumAbs = 0 |  |
| for( n = 15; n >= 0; n− − ) { |  |
| xC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 0 ] |  |
| yC = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ n + offset ][ 1 ] |  |
| if( significant\_coeff\_flag[ xC ][ yC ] ) { |  |
| baseLevel = 1 + coeff\_abs\_level\_greater1\_flag[ n ] + coeff\_abs\_level\_greater2\_flag[ n ] |  |
| if( baseLevel = = ( ( numSigCoeff < 8 ) ? ( (n = = firstGreater1CoeffIdx) ? 3 : 2 ) : 1 ) ) |  |
| **coeff\_abs\_level\_remaining[** n **]** | ae(v) |
| transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   ( coeff\_abs\_level\_remaining[ n ] + baseLevel ) \* ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| if( sign\_data\_hiding\_flag && signHidden ) { |  |
| sumAbs += ( coeff\_abs\_level\_remaining[ n ] + baseLevel ) |  |
| if( n = = firstNZPosInCG && (sumAbs%2 = = 1) ) |  |
| transCoeffLevel[x0][y0][cIdx][xC][yC] = −  transCoeffLevel[x0][y0][cIdx][xC][yC] |  |
| } |  |
| numSigCoeff++ |  |
| } else |  |
| transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] = 0 |  |
| } |  |
| } |  |
| } |  |

## Semantics

### Raw byte sequence payloads and RBSP trailing bits semantics

#### Sequence parameter set RBSP semantics

**profile\_idc** and **level\_idc** indicate the profile and level to which the coded video sequence conforms.

**reserved\_zero\_8bits** shall be equal to 0. Decoders shall ignore the value of reserved\_zero\_8bits.

**seq\_parameter\_set\_id** identifies the sequence parameter set that is referred to by the picture parameter set. The value of seq\_parameter\_set\_id shall be in the range of 0 to 31, inclusive.

**chroma\_format\_idc** specifies the chroma sampling relative to the luma sampling as specified in subclause 6.2. The value of chroma\_format\_idc shall be in the range of 0 to 3, inclusive. When chroma\_format\_idc is not present, it is inferred to be equal to 1 (4:2:0 chroma format).

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

NOTE 4 – There is no dependency in decoding processes between the colour planes having different colour\_plane\_id values. For example, the decoding process of a monochrome picture with one value of colour\_plane\_id does not use any data from monochrome pictures having different values of colour\_plane\_id for inter prediction.

Depending on the value of separate\_colour\_plane\_flag, the value of the variable ChromaArrayType is assigned as follows:

– If separate\_colour\_plane\_flag is equal to 0, ChromaArrayType is set equal to chroma\_format\_idc.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), ChromaArrayType is set equal to 0.

**max\_temporal\_layers\_minus1** + 1 specifies the maximum number of temporal layers present in the sequence. The value of max\_temporal\_layers\_minus1 shall be in the range of 0 to 7, inclusive. [Ed. (GJS): Inconsistency in semantics between "+ 1 specifies" and "plus 1 specifies". Which is better?]

**pic\_width\_in\_luma\_samples** specifies the width of each decoded picture in units of luma samples. pic\_width\_in\_luma\_samples shall not be equal to 0 and shall be an integer multiple of ( 1 << Log2MinCbSize ).

**pic\_height\_in\_luma\_samples** specifies the height of each decoded picture in units of luma samples. pic\_height\_in\_luma\_samples shall not be equal to 0 and shall be an integer multiple of ( 1 << Log2MinCbSize ).

**pic\_cropping\_flag** equal to 1 indicates that the picture cropping offset parameters follow next in the sequence parameter set. pic\_cropping\_flag equal to 0 indicates that the picture cropping offset parameters are not present.

**pic\_crop\_left\_offset**, **pic\_crop\_right\_offset**, **pic\_crop\_top\_offset**, and **pic\_crop\_bottom\_offset** specify the samples of the pictures in the coded video sequence that are output from the decoding process, in terms of a rectangular region specified in picture coordinates for output.

The variables CropUnitX and CropUnitY are derived as follows:

– If ChromaArrayType is equal to 0, CropUnitX and CropUnitY are derived as:

CropUnitX = 1 (7-2)  
CropUnitY = 1 (7-3)

– Otherwise (ChromaArrayType is equal to 1, 2, or 3), CropUnitX and CropUnitY are derived as:

CropUnitX = SubWidthC (7-4)  
CropUnitY = SubHeightC (7-5)

The picture cropping rectangle contains luma samples with horizontal picture coordinates from CropUnitX \* pic\_crop\_left\_offset to PicWidthInSamplesL − ( CropUnitX \* pic\_crop\_right\_offset + 1 ) and vertical picture coordinates from CropUnitY \* pic\_crop\_top\_offset to PicHeightInSamplesL − ( CropUnitY \* pic\_crop\_bottom\_offset + 1 ), inclusive. The value of pic\_crop\_left\_offset shall be in the range of 0 to ( PicWidthInSamplesL / CropUnitX ) − ( pic\_crop\_right\_offset + 1 ), inclusive; and the value of pic\_crop\_top\_offset shall be in the range of 0 to ( PicHeightInSamplesL / CropUnitY ) − ( pic\_crop\_bottom\_offset + 1 ), inclusive.

When pic\_cropping\_flag is equal to 0, the values of pic\_crop\_left\_offset, pic\_crop\_right\_offset, pic\_crop\_top\_offset, and pic\_crop\_bottom\_offset is inferred to be equal to 0.

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

NOTE – The picture cropping parameters are only applied at the output. All internal decoding processes are applied to the uncropped picture size.

**bit\_depth\_luma\_minus8** + 8 specifies the bit depth of the samples of the luma array and the value of the luma quantization parameter range offset QpBdOffsetY, as specified by

BitDepthY = 8 + bit\_depth\_luma\_minus8 (‑)  
QpBdOffsetY = 6 \* bit\_depth\_luma\_minus8 (‑)

bit\_depth\_luma\_minus8 shall be in the range of 0 to 6, inclusive.

**bit\_depth\_chroma\_minus8** + 8specifies the bit depth of the samples of the chroma arrays and the value of the chroma quantization parameter range offset QpBdOffsetC, as specified by

BitDepthC = 8 + bit\_depth\_chroma\_minus8 (‑)  
QpBdOffsetC = 6 \* bit\_depth\_chroma\_minus8 (‑)

bit\_depth\_chroma\_minus8 shall be in the range of 0 to 6, inclusive.

**pcm\_enabled\_flag** equal to 0 specifies that PCM data shall not be present in the video sequence.

**pcm\_sample\_bit\_depth\_luma\_minus1** + 1 specifies the number of bits used to represent each of PCM sample values of luma component. The value of pcm\_sample\_bit\_depth\_luma\_minus1 + 1 shall be less than or equal to the value of BitDepthY.

PCMBitDepthY = 1 + pcm\_sample\_bit\_depth\_luma\_minus1 (‑)

**pcm\_sample\_bit\_depth\_chroma\_minus1** + 1 specifies the number of bits used to represent each of PCM sample values of chroma components. The value of pcm\_sample\_bit\_depth\_chroma\_minus1 + 1 shall be less than or equal to the value of BitDepthC.

PCMBitDepthC = 1 + pcm\_sample\_bit\_depth\_chroma\_minus1 (‑)

**qpprime\_y\_zero\_transquant\_bypass\_flag** equal to 1 specifies that, when QP′Y is equal to 0, a lossless coding process shall be applied. In lossless coding operation, the scaling and transform process as specified in subclause 8.6 and the in-lopp filter procecss as specified in subclause 8.7 are bypassed.

**log2\_max\_pic\_order\_cnt\_lsb\_minus4** specifies the value of the variable MaxPicOrderCntLsb that is used in the decoding process for picture order count as follows:

MaxPicOrderCntLsb = 2( log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 ) (‑)

The value of log2\_max\_pic\_order\_cnt\_lsb\_minus4 shall be in the range of 0 to 12, inclusive.

**max\_dec\_pic\_buffering[** i **]** specifies the required size of the HRD decoded picture buffer (DPB) in units of picture storage buffers for the bitstream subset as specified in subclause 10.1 with  i as input. The coded video sequence shall not require a decoded picture buffer with size of more than Max( 1, max\_dec\_pic\_buffering ) picture storage buffers to enable the output of decoded pictures at the output times specified by dpb\_output\_delay of the picture timing SEI messages. The value of max\_dec\_pic\_buffering[ i ] shall be in the range of 0 to MaxDpbSize (as specified in subclause A.4), inclusive.

**num\_reorder\_pics[** i **]** indicates the maximum number of pictures preceding any picture in decoding order and succeeding that picture in output order for the bitstream subset as specified in subclause 10.1 with  i as input. The value of num\_reorder\_pics[ i ] shall be in the range of 0 to max\_dec\_pic\_buffering[ i ], inclusive.

**max\_latency\_increase[** i **]** not equal to 0 is used to compute the value of MaxLatencyPictures[ i ] as specified by MaxLatencyPictures[ i ] = num\_reorder\_pics[ i ] + max\_latency\_increase[ i ] for the bitstream subset as specified in subclause 10.1 with  i as input. When max\_latency\_increase[ i ] is not equal to 0, the value of MaxLatencyPictures[ i ] specifies the maximum number of pictures that can precede any picture in the coded video sequence in output order and follow that picture in decoding order for the bitstream subset as specified in subclause 10.1 with  i as input. When max\_latency\_increase[ i ] is equal to 0, no corresponding limit is expressed. The value of max\_latency\_increase[ i ] shall be in the range of 0 to 232 − 1, inclusive.

**restricted\_ref\_pic\_lists\_flag** equal to 1 indicates that all slices of the same slice type that belong to the same picture have identical reference picture lists. restricted\_ref\_pic\_lists\_flag equal to 0 indicates that there may be slices of the same slice type that belong to the same picture that have different reference picture lists. num\_ref\_idx\_active\_override\_flag, num\_ref\_idx\_l0\_active\_minus1, num\_ref\_idx\_l1\_active\_minus1, ref\_pic\_list\_combination\_flag and num\_ref\_idx\_lc\_active\_minus1 shall when present be identical for all slices of the same slice type that belong to the same picture when restricted\_ref\_pic\_lists\_flag is equal to 1.

**lists\_modification\_present\_flag** equal to 0 specifies that the syntax structures ref\_pic\_list\_modification( ) and ref\_pic\_list\_combination( ) are not present in the slice header. lists\_modification\_present\_flag equal to 1 specifies that the syntax structures ref\_pic\_list\_modification( ) andref\_pic\_list\_combination( ) are present in the slice header.

**log2\_min\_coding\_block\_size\_minus3** specifies the minimum size of a coding block.

The variable Log2MinCbSize is set equal to log2\_min\_coding\_block\_size\_minus3 + 3.

**log2\_diff\_max\_min\_coding\_block\_size** specifies the difference between the maximum and minimum coding block size.

The variables Log2CtbSize, PicWidthInCtbs and PicHeightInCtbs are derived as follows.

Log2CtbSize = log2\_min\_coding\_block\_size\_minus 3 + 3 + log2\_diff\_max\_min\_coding\_block\_size (‑)  
PicWidthInCtbs = Ceil( pic\_width\_in\_luma\_samples ÷ ( 1 << Log2CtbSize ) ) (‑)  
PicHeightInCtbs = Ceil( pic\_height\_in\_luma\_samples ÷ ( 1 << Log2CtbSize ) ) (‑)  
PicWidthInMinCbs = PicWidthInSamplesL / ( 1 << Log2MinCbSize ) (‑)  
PicHeightInMinCbs = PicHeightInSamplesL / ( 1 << Log2MinCbSize ) (‑)

The variables CtbWidthC and CtbHeightC, which specify the width and height, respectively, of the chroma arrays for each coding treeblock, are derived as follows.

– If chroma\_format\_idc is equal to 0 (monochrome) or separate\_colour\_plane\_flag is equal to 1, CtbWidthC and CtbHeightC are both equal to 0.

– Otherwise, CtbWidthC and CtbHeightC are derived as

CtbWidthC = CtbSize / SubWidthC (7‑18)  
CtbHeightC = CtbSize / SubHeightC (7‑19)

**log2\_min\_transform\_block\_size\_minus2** specifies the minimum transform size.

The variable Log2MinTrafoSize is set equal to log2\_min\_transform\_block\_size\_minus2 + 2. The bitstream shall not contain data that result in Log2MinTrafoSize greater than or equal to Log2MinCbSize.

**log2\_diff\_max\_min\_transform\_block\_size** specifies the difference between the maximum and minimum transform size.

The variable Log2MaxTrafoSize is set equal to log2\_min\_transform\_block\_size\_minus 2 + 2 + log2\_diff\_max\_min\_transform\_block\_size.

The bitstream shall not contain data that result in Log2MaxTrafoSize greater than Log2CtbSize.

**log2\_min\_pcm\_coding\_block\_size\_minus3** + 3 specifies the minimum size of I\_PCM coding blocks.

The variable Log2MinIPCMCUSize is set equal to log2\_min\_pcm\_coding\_block\_size\_minus3 + 3. The variable Log2MinIPCMCUSize shall be equal or less than Min( Log2CtbSize, 5 ).

**log2\_diff\_max\_min\_pcm\_coding\_block\_size** specifies the difference between the maximum and minimum size of I\_PCM coding blocks.

The variable Log2MaxIPCMCUSize is set equal to log2\_min\_pcm\_coding\_block\_size\_minus3 + 3 + log2\_diff\_max\_min\_pcm\_coding\_block\_size. The variable Log2MaxIPCMCUSize shall be equal or less than Min( Log2CtbSize, 5 ).

**max\_transform\_hierarchy\_depth\_intra** specifies the maximum hierarchy depth for transform blocks of coding blocks coded in intra prediction mode.

**max\_transform\_hierarchy\_depth\_inter** specifies the maximum hierarchy depth for transform units of coding units coded in inter prediction mode.

**scaling\_list\_enable\_flag** equal to 1 specifies that scaling list is used for scaling process for transform coefficients in 8.6.3. scaling\_list\_enable\_flag equal to 0 specifies that scaling list is not used for scaling process for transform coefficients in 8.6.3.

**chroma\_pred\_from\_luma\_enabled\_flag** equal to 1 specifies the intra chroma prediction process using the reconstructed luma block is applied according to the intra chroma prediction mode.

**deblocking\_filter\_in\_aps\_enabled\_flag** equal to 0 specifies that deblocking parameters are present in the slice header. deblocking\_filter\_in\_aps\_enabled\_flag equal to 1 specifies that deblocking parameters are present in the APS.

**seq\_loop\_filter\_across\_slices\_enabled\_flag** equal to 1 specifies that the slice\_loop\_filter\_across\_slices\_enabled\_flag determines whether in-loop filtering operations are performed across slice boundaries; otherwise, the in-loop filtering operations are slice-independent and not applied across slice boundaries. The in-loop filtering operations include the deblocking filter, sample adaptive offset filter, and adaptive loop filter.

**asymmetric\_motion\_partitions\_enabled\_flag** equal to 1 specifies that asymmetric motion partitions may be used in treeblocks; asymmetric\_motion\_partitions\_enabled\_flag equal to 0 specifies that asymmetric motion partitions cannot be used in treeblocks.

**non\_square\_quadtree\_enabled\_flag** equal to 1 specifies that non-square quadtree partitions may be used in treeblocks; non\_square\_quadtree\_enabled\_flag equal to 0 specifies that non-square quadtree partitions cannot be used in treeblocks.

**sample\_adaptive\_offset\_enabled\_flag** equal to 1 specifies the sample adaptive offset process is applied to the reconstruced picture after the deblocking filter process.

**adaptive\_loop\_filter\_enabled\_flag** equal to 1 specifies the adaptive loop filter process is applied to the reconstructed picture after the deblocking filter process.

**alf\_coef\_in\_slice\_flag** equal to 1 specifies that the ALF parameters alf\_param() are present in the slice header; equal to 0 specifies that the ALF parameters alf\_param() are not present in the APS.

**pcm\_loop\_filter\_disable\_flag** specifies whether the loop filter process is disabled on reconstructed pixels of I\_PCM blocks. If the pcm\_loop\_filter\_disable\_flag value is equal to 1, deblocking filter, sample adaptive offset and adaptive loop filter processes on the reconstructed pixels of I\_PCM blocks are disabled; otherwise if the pcm\_loop\_filter\_disable\_flag value is equal to 0, deblocking filter, sample adaptive offset and adaptive loop filter processes on the reconstructed pixels of I\_PCM blocks are not disabled. When pcm\_loop\_filter\_disable\_flag is not present, it is inferred to be equal to 0.

[Ed. (WJ): select one expression – enabled\_flag or disable\_flag]

**temporal\_id\_nesting\_flag** specifies whether inter prediction is additionally restricted for the coded video sequence.

Dependent on temporal\_id\_nesting\_flag, the following applies.

– If temporal\_id\_nesting\_flag is equal to 0, additional constraints may not be obeyed.

– Otherwise (temporal\_id\_nesting\_flag is equal to 1), the following applies.

– For each access unit auA with temporal\_id equal to tIdA, an access unit auB with temporal\_id equal to tIdB and tIdB less than or equal to tIdA shall not be referenced by inter prediction when there exists an access unit auC with temporal\_id equal to tIdC and tIdC less than tIdB, which follows the access unit auB and precedes the access unit auA in decoding order.

NOTE 1 – The syntax element temporal\_id\_nesting\_flag is used to indicate that temporal up-switching, i.e., switching from decoding of up to a specific temporal\_id tIdN to decoding up to a temporal\_id tIdM > tIdN, is always possible.

**inter\_4x4\_enabled\_flag** specifies whether inter prediction can be applied to blocks having the size of 4 by 4 luma samples.

**num\_short\_term\_ref\_pic\_sets** specifies the number of short-term reference picture sets that are specified in the sequence parameter set. The value of num\_short\_term\_ref\_pic\_sets shall be in the range of 0 to 64, inclusive.

NOTE – A decoder must allocate space for a total number of num\_short\_term\_ref\_pic\_sets + 1 short-term reference picture sets since a coded video sequence may contain up to one short-term reference picture set explicitly signalled in the slice headers of a current picture. An explicitly signalled short-term reference picture set will always have an index equal to num\_short\_term\_ref\_pic\_sets in the list of short-term reference picture sets.

**long\_term\_ref\_pics\_present\_flag** equal to 0 specifies that no long-term reference picture is used for inter prediction of any coded picture in the coded video sequence. long\_term\_ref\_pics\_present\_flag equal to 1 specifies that long-term reference pictures may be used for inter prediction of one or more coded pictures in the coded video sequence.

**tiles\_or\_entropy\_coding\_sync\_idc** equal to 0 specifies that there is only one tile in each picture in the coded video sequence, and no specific synchronization process for context variables is invoked before decoding the first coding treeblock of a row of coding treeblocks.

tiles\_or\_entropy\_coding\_sync\_idcequal to 1 specifies that there may be more than one tile in each picture in the coded video sequence, and no specific synchronization process for context variables is invoked before decoding the first coding treeblock of a row of coding treeblocks.

tiles\_or\_entropy\_coding\_sync\_idcequal to 2 specifies that there is only one tile in each picture in the coded video sequence, a specific synchronization process for context variables is invoked before decoding the first coding treeblock of a row of coding treeblocks, and a specific memorization process for context variables is invoked after decoding two coding treeblocks of a row of coding treeblocks.

The value of tiles\_or\_entropy\_coding\_sync\_idc shall be in the range of 0 to 2, inclusive.

[Ed. (YK): All the following semantics, including the derivations of the vectors, may need to be moved to PPS semantics, and here the the syntax elements here in SPS only specify the default values fot the corresponding syntax elements in the PPS.]

**num\_tile\_columns\_minus1** plus 1 specifies the number of tile columns partitioning the picture.

**num\_tile\_rows\_minus1** plus 1 specifies the number of tile rows partitioning the picture.

When num\_tile\_columns\_minus1 is equal to 0, num\_tile\_rows\_minus1 shall not be equal to 0.

[Ed. (YK): The following should go to Annex A.]

One or both of the following conditions shall be fulfilled for each slice and tile:

– All coded blocks in a slice belong to the same tile.

– All coded blocks in a tile belong to the same slice.

NOTE 2 – Within the same picture, there may be both slices that contain multiple tiles and tiles that contain multiple slices.

**uniform\_spacing\_flag** equal to 1 specifies that column boundaries and likewise row boundaries are distributed uniformly across the picture. uniform\_spacing\_flag equal to 0 specifies that column boundaries and likewise row boundaries are not distributed uniformly across the picture but signalled explicitly using the syntax elements column\_width[ i ] and row\_height[ i ].

**column\_width[**i **]** specifies the width of the i-th tile column in units of coding treeblocks.

**row\_height[**i **]** specifies the height of the i-th tile row in units of coding treeblocks.

[Ed. (YK): The following should go to somewhere in Clause 8.]

Values of ColumnWidth[ i ], specifying the width of the i-th tile column in units of coding treeblocks, and the values of ColumnWidthInLumaSamples[ i ], specifying the width of the i-th tile column in units of luma samples, are derived as follows:

for( i = 0; i <= num\_tile\_columns\_minus1; i++ ) {  
 if( uniform\_spacing\_flag )  
 ColumnWidth[ i ] = ( ( i + 1 ) \* PicWidthInCtbs ) / ( num\_tile\_columns\_minus1 + 1 ) – (‑)  
 ( i \* PicWidthInCtbs ) / ( num\_tile\_columns\_minus1 + 1 )  
 else  
 ColumnWidth[ i ] = column\_width[ i ]  
 ColumnWidthInLumaSamples[ i ] = ColumnWidth[ i ] << Log2CtbSize  
}

Values of RowHeight[ i ], specifying the height of the i-th tile row in units of coding treeblocks, are derived as follows:

for( i = 0; i <= num\_tile\_rows\_minus1; i++ )  
 if( uniform\_spacing\_flag )  
 RowHeight[ i ] = ( ( i + 1 ) \* PicHeightInCtbs ) / ( num\_tile\_rows\_minus1 + 1 ) – (‑)  
 ( i \* PicHeightInCtbs ) / ( num\_tile\_rows\_minus1 + 1)   
 else  
 RowHeight[ i ] = row\_height[ i ]

Values of ColBd[ i ], specifying the location of the left column boundary of the i-th tile column in units of coding treeblocks, are derived as follows:

for( ColBd[ 0 ] = 0, i = 0; i <= num\_tile\_columns\_minus1; i++ ) (‑)  
 ColBd[ i + 1 ] = ColBd[ i ] + ColumnWidth[ i ]

Values of RowBd[ i ], specifying the location of the top row boundary of the i-th tile row in units of coding treeblocks, are derived as follows:

for( RowBd[ 0 ] = 0, i = 0; i <= num\_tile\_rows\_minus1; i++ ) (‑)  
 RowBd[ i + 1 ] = RowBd[ i ] + RowHeight[ i ]

**loop\_filter\_across\_tiles\_enabled\_flag** equal to 1 specifies that in-loop filtering operations are performed across tile boundaries. loop\_filter\_across\_tiles\_enabled\_flag equal to 0 specifies that in-loop filtering operations are not performed across tile boundaries. The in-loop filtering operations include the deblocking filter, sample adaptive offset, and adaptive loop filter operations. When not present, the value of loop\_filter\_across\_tiles\_enabled\_flag is inferred to be equal to 1.

**sps\_extension\_flag** equal to 0 specifies that no sps\_extension\_data\_flag syntax elements are present in the sequence parameter set RBSP syntax structure. sps\_extension\_flag shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. The value of 1 for sps\_extension\_flag is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore all data that follow the value 1 for sps\_extension\_flag in a sequence parameter set NAL unit.

**sps\_extension\_data\_flag** may have any value. Its value does not affect decoder conformance to profiles specified in this Recommendation | International Standard.

### Coding unit semantics

**skip\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, no more syntax elements except the motion vector predictor indices are parsed after skip\_flag[ x0 ][ y0 ]. 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 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 PredMode is derived as follows.

* If pred\_mode\_flag is equal to 0,
  + PredMode is set to MODE\_INTER.
* Otherwise (pred\_mode\_flag is equal to 1),
  + PredMode is set to MODE\_INTRA.

When pred\_mode\_flag is not present, the variables PredMode is derived as follows.

* If slice\_type is equal to I,
  + PredMode is inferred to be equal to MODE\_INTRA
* Otherwise (slice\_type is equal to P or B), if skip\_flag[ x0 ][ y0 ] is equal to 1,
  + PredMode is inferred to be equal to MODE\_SKIP

**part\_mode** specifies partitioning mode of the current coding unit. The semantics of part\_mode depend on PredMode. The variables PartMode and IntraSplitFlag are derived from the value of part\_mode as defined in Table 7‑10.

The value of part\_mode is restricted as follows.

* If PredMode is equal to MODE\_INTRA, part\_mode shall be equal to 0 or 1.
* Otherwise (PredMode is equal to MODE\_INTER), the following applies
* If log2CbSize is greater than Log2MinCbSize, part\_mode shall be in the range of 0 to 2, inclusive and in the range of 4 to 7, inclusive.
* Otherwise if log2CbSize is equal to 3 and inter\_4x4\_enabled\_flag is equal to 0, ,the value of part\_mode shall be in the range of 0 to 2, inclusive.
* Otherwise (log2CbSize is greater than 3 or inter\_4x4\_enabled\_flag is equal to q), the value of part\_mode shall be in the range of 0 to 3, inclusive.

When part\_mode is not present, the variables PartMode and IntraSplitFlag are derived as follows.

* PartMode is inferred to be equal to PART\_2Nx2N,
* IntraSplitFlag is inferred to be equal to 0.

**pcm\_flag** specifies whether the associated coding unit with PART\_2Nx2N is coded by I\_PCM: If the pcm\_flag is equal to 1, the associated coding unit with PART\_2Nx2N is coded by I\_PCM. When the pcm\_flag is not present, it shall be infered to be equal to 0.

**num\_subsequent\_pcm** specifies the number of subsequent I\_PCM coding units with the current log2CbSize that successively follow the current I\_PCM coding unit in the same depth of a TB. The values of pcm\_flags of the subsequent coding units are set equal to 1. It is a requirement of bitstream conformance that the immediate roots of the current and subsequent I\_PCM coding units are identical. The value num\_subsequent\_pcm shall be in the range of 0 to 3, inclusive.

**pcm\_alignment\_zero\_bit** is a bit equal to 0.

**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.

Table 7‑10 – Name association to prediction mode and partitioning type

|  |  |  |  |
| --- | --- | --- | --- |
| **PredMode** | **part\_mode** | **IntraSplitFlag** | **PartMode** |
| MODE\_INTRA | 0 | 0 | PART\_2Nx2N |
| 1 | 1 | PART\_NxN |
| MODE\_INTER | 0 | 0 | PART\_2Nx2N |
| 1 | 0 | PART\_2NxN |
| 2 | 0 | PART\_Nx2N |
| 3 | 0 | PART\_NxN |
| 4 | 0 | PART\_2NxnU |
| 5 | 0 | PART\_2NxnD |
| 6 | 0 | PART\_nLx2N |
| 7 | 0 | PART\_nRx2N |

**no\_residual\_data\_flag** equal to 1 specifies that no residual data are present for the current coding unit. no\_residual\_data\_flag equal to 0 specifies that residual data are present for the current coding unit.

When no\_residual\_data\_flag is not present, its value is inferred to be equal to 0.

### Transform tree semantics

The variable InterTUSplitDirection defines how a transform block is split into four blocks with smaller horizontal or vertical size for the purpose of transform coding. The blocks are half horizontal and vertical size when PredMode is equal to MODE\_INTRA or InterTUSplitDirection is equal to 2, full horizontal and quarter vertical size when InterTUSplitDirection is equal to 0, quarter horizontal and full vertical size when InterTUSplitDirection is equal to 1. InterTUSplitDirection is specified as follows.

if( ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_2NxN  
 | | PartMode = = PART\_2NxnU | | PartMode = = PART\_2NxnD ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth < log2TrafoHeight ) )   
{  
 InterTUSplitDirection = 0  
}  
else if( ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_Nx2N  
 | | PartMode = = PART\_nLx2N | | PartMode = = PART\_nRx2N ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth > log2TrafoHeight ) )   
{  
 InterTUSplitDirection = 1  
}  
else   
{  
 InterTUSplitDirection = 2  
}

[Ed. (KS): This process should be moved to somewhere else to be a derivation process. ]

**split\_transform\_flag**[ x0 ][ y0 ][ trafoDepth ] specifies whether a block is split into four blocks with smaller horizontal or vertical size for the purpose of transform coding. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered block relative to the top-left luma sample of the picture. The array index trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

The variable interSplitFlag is derived as follows.

* If max\_transform\_hierarchy\_depth\_inter is equal to 0 and PredMode is equal t oMODE\_INTER and PartMode is not equal to PART\_2Nx2N and trafoDepth is equal to 0, interSplitFlag is set to 1.
* Otherwise, interSplitFlag is set to 0.

When split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] is not present, it is inferred as follows:

* If one of the following conditions is true, the value of split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 1.
* log2TrafoSize is greater than Log2MaxTrafoSize
* IntraSplitFlag is equal to 1 and trafoDepth is equal to 0
* interSplitFlag is equal to 1
* Otherwise, the value of split\_transform\_flag[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 0.

**cbf\_luma**[ x0 ][ y0 ][ trafoDepth ] 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. The array index trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When cbf\_luma[ x0 ][ y0 ][ trafoDepth ] is not present, it is inferred to be equal to 1.

**cbf\_cb**[ x0 ][ y0 ][ trafoDepth ] 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 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 trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is not present, the value of cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is inferred as follows.

* If all of the following conditions are true, cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 1.
* blkIdx is equal to 3
* log2TrafoSize is less than Log2MaxTrafoSize
* cbf\_cb[ xBase ][ yBase ][ trafoDepth ] is equal to 0
* cbf\_cb[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] is equal to 0
* cbf\_cb[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] is equal to 0
* Otherwise if firstChromaCbf is equal to 0 and log2TrafoSize is equal to Log2MinTrafoSize, cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to cbf\_cb[ xBase ][ yBase ][ trafoDepth − 1 ]
* Otherwise, cbf\_cb[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 0.

**cbf\_cr**[ x0 ][ y0 ][ trafoDepth ] 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 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 trafoDepth specifies the current subdivision level of a coding unit into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding units.

When cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is not present, the value of cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is inferred as follows.

* If all of the following conditions are true, cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 1.
* blkIdx is equal to 3
* log2TrafoSize is less than Log2MaxTrafoSize
* cbf\_cr[ xBase ][ yBase ][ trafoDepth ] is equal to 0
* cbf\_cr[ xBase + ( 1 << log2TrafoWidth ) ][ yBase ][ trafoDepth ] is equal to 0
* cbf\_cr[ xBase ][ yBase + ( 1 << log2TrafoHeight ) ][ trafoDepth ] is equal to 0
* Otherwise if firstChromaCbf is equal to 0 and log2TrafoSize is equal to Log2MinTrafoSize, cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to cbf\_cr[ xBase ][ yBase ][ trafoDepth − 1 ]
* Otherwise, cbf\_cr[ x0 ][ y0 ][ trafoDepth ] is inferred to be equal to 0.

### Transform coefficient semantics

The transform coefficient levels are parsed into the arrays transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ]. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb, and equal to 2 for Cr. The array indices xC, yC specify the transform coefficient location ( xC, yC ) within the current transform block.

When PredMode is equal to MODE\_INTRA, different scanning orders are used. The array ScanType[ log2TrafoSize − 2 ][ IntraPredMode ], specifying the scanning order for various luma transform block sizes and intra prediction modes, is derived as specified in Table 7‑12.

Table 7‑12 – Specification of ScanType[ log2TrafoSize − 2 ][ IntraPredMode ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **IntraPredMode** | **log2TrafoSize − 2** | | | |
| **0** | **1** | **2** | **3** |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 2 – 5 | 0 | 0 | 0 | 0 |
| 6 – 14 | 2 | 2 | 0 | 0 |
| 15 – 21 | 0 | 0 | 0 | 0 |
| 22 – 30 | 1 | 1 | 0 | 0 |
| 31 – 35 | 0 | 0 | 0 | 0 |

**cu\_qp\_delta** specifies the difference between a luma quantization parameter to be used and its prediction for a coding unit quantization group where the coding unit quantization group is specified as follows:

* If a coding unit with the split\_coding\_unit\_flag[ x0 ][ y0 ] equal to 0 and the log2CbSize is greater than or equal to log2MinCUDQPSize, the coding unit quantization group includes this coding unit only.
* Otherwise, if a coding unit with the split\_coding\_unit\_flag[ x0 ][ y0 ] equal to 1 and the log2CbSize is equal to log2MinCUDQPSize, the coding unit quantization group includes all coding units split from this coding unit.

The decoded value of cu\_qp\_delta shall be in the range of –( 26+ QpBdOffsetY / 2 ) to +( 25+ QpBdOffsetY / 2 ), inclusive. cu\_qp\_delta is inferred to be equal to 0 when it is not present for any coding unit quantization group.

The luma quantization parameter QP’Y and the chroma quantization parameters QP’Cb and QP’Cr are derived by invoking the process specified in subclause 8.6.1 with ( x0 ,y0 ) as inputs.

### Residual coding semantics

The array ScanOrder[ log2BlockWidth ][ log2BlockHeight ][ scanIdx ][ sPos ][ sComp ] specifies the mapping of the scan position sPos, ranging from 0 to ( ( 1 << log2BlockWidth ) \* ( 1 << log2BlockHeight ) ) − 1, to horizontal and vertical components of the scan-order matrix. The array index scanIdx equal to 0 specifies an up-right diagonal scan, scanIdx equal to 1 specifies a horizontal scan, and scanIdx equal to 2 specifies a vertical scan. The array index sComp equal to 0 specifies the horizontal component and the array index sComp equal to 1 specifies the vertical component. The array ScanOrder is derived as follows.

The scanning order array initialization process as specified in 6.5.1 is invoked with 1 << log2TrafoWidth and 1 << log2TrafoHeight as input and the output is assigned to ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ].

**last\_significant\_coeff\_x\_prefix** specify the prefix of the column position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_x\_prefix shall be in the range from 0 to ( log2TrafoWidth << 1 ) − 1, inclusive.

**last\_significant\_coeff\_y\_prefix** specify the prefix of the row position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_y\_prefix shall be in the range from 0 to ( log2TrafoHeight << 1 ) − 1, inclusive.

**last\_significant\_coeff\_x\_suffix** specify the suffix of the column position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_x\_suffix shall be in the range from 0 to ( 1 << ( ( last\_significant\_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\_significant\_coeff\_x\_suffix is not present, the following applies.

LastSignificantCoeffX = last\_significant\_coeff\_x\_prefix (7‑71)

* Otherwise (last\_significant\_coeff\_x\_suffix is present), the following applies.

LastSignificantCoeffX = ( (1 << ((last\_significant\_coeff\_x\_prefix >> 1) − 1)) \*   
 (2 + (last\_significant\_coeff\_x\_prefix & 1)) + (7‑72)  
 last\_significant\_coeff\_x\_suffix

**last\_significant\_coeff\_y\_suffix** specify the suffix of the row position of the last significant coefficient in scanning order within a transform block. The values of last\_significant\_coeff\_y\_suffix shall be in the range from 0 to ( 1 << (  ( last\_significant\_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\_significant\_coeff\_y\_suffix is not present, the following applies.

LastSignificantCoeffY = last\_significant\_coeff\_y\_prefix (7‑73)

* Otherwise (last\_significant\_coeff\_y\_suffix is present), the following applies.

LastSignificantCoeffY = ( (1 << ((last\_significant\_coeff\_y\_prefix >> 1) − 1)) \*   
 (2 + (last\_significant\_coeff\_y\_prefix & 1)) + (7‑74)   
 last\_significant\_coeff\_y\_suffix

When scanIdx is equal to 2, the coordinates are swapped as follows.

temp = LastSignificantCoeffX  
LastSignificantCoeffX = LastSignificantCoeffY (7‑75)  
LastSignificantCoeffY = temp

**significant\_coeff\_group\_flag**[ xCG ][ yCG ] specifies for the coefficient group position ( xCG, yCG ) within the current transform block whether the corresponding coefficient group at location ( xCG, yCG ) has non-zero transform coefficient level. A coefficient group at location ( xCG, yCG ) is an array of 16 transform coefficient levels at locations ( xC, yC ).

* If significant\_coeff\_group\_flag[ xCG ][ yCG ] is equal to 0, the 16 transform coefficient levels of the coefficient group at location ( xCG, yCG ) are inferred to be equal to 0;
* Otherwise (significant\_coeff\_group\_flag[ xCG ][ yCG ] is equal to 1), the following applies.
* If significant\_coeff\_group\_flag[ xCG ][ yCG ] is present or ( xCG << 2, yCG << 2 ) is the last significant position ( LastSignificantCoeffX, LastSignificantCoeffY ), at least one of the 16 transform coefficient levels of the coefficient group at location ( xCG, yCG ) has a non zero value.
* Otherwise, at least one of the 16 significant\_coeff\_flag syntax elements is present for the coefficient group at location ( xCG, yCG )

When significant\_coeff\_group\_flag[ xCG ][ yCG ] is not present, it is inferred as follows.

* If one of the following conditions is true, significant\_coeff\_group\_flag[ xCG ][ yCG ] is inferred to be equal to 1.
* ( xCG, yCG ) is equal to ( LastSignificantCoeffX >> 2, LastSignificantCoeffY >> 2 )
* ( xCG, yCG ) is equal to ( 0, 0 )
* Otherwise, significant\_coeff\_group\_flag[ xCG ][ yCG ] is inferred to be equal to 0.

**significant\_coeff\_flag[** xC **][** yC **]** specifies for the transform coefficient position ( xC, yC ) within the current transform block whether the corresponding transform coefficient level at location ( xC, yC ) is non-zero as follows.

* If significant\_coeff\_flag[ xC ][ yC ] is equal to 0, the transform coefficient level at location ( xC, yC ) is set equal to 0.
* Otherwise (significant\_coeff\_flag[ xC ][ yC ] is equal to 1), the transform coefficient level at location ( xC, yC ) has a non‑zero value.

When significant\_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 both of the following conditions are true, significant\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 1
* ( xC, yC ) is equal to ( xCG << 2, yCG << 2 )
* implicitNonZeroCoeff is equal to 1
* Otherwise, significant\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 0.

**coeff\_abs\_level\_greater1\_flag[** n **]** specifies for the scanning position n whether there are transform coefficient levels greater than 1.

When coeff\_abs\_level\_greater1\_flag[ n ] is not present, it is inferred to be equal to 0.

**coeff\_abs\_level\_greater2\_flag[** n **]** specifies for the scanning position n whether there are transform coefficient levels greater than 2.

When coeff\_abs\_level\_greater2\_flag[ n ] is not present, it is inferred to be equal to 0.

**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.

**coeff\_abs\_level\_remaining[** n **]** is the remaining absolute value of a transform coefficient level that is coded with Golomb-Rice code at the scanning position n. The value of coeff\_abs\_level\_remaining is constrained by the limits in subclause 8.6.3.When coeff\_abs\_level\_remaining [ n ] is not present, it is inferred as 0.

# Decoding process

## Decoding process for coding units coded in inter prediction mode

### Decoding process for the residual signal of coding units coded in inter prediction mode

#### Decoding process for luma residual blocks

Inputs to this process are:

– a luma location ( xC, yC ) specifying the top-left luma sample of the current coding unit relative to the top‑left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left luma sample of the current block relative to the top‑left luma sample of the current coding unit,

– a variable log2TrafoWidth specifying the width of the current block,

– a variable log2TrafoHeight specifying the height of the current block,

– a variable trafoDepth specifying the hierarchy depth of the current block relative to the coding unit,

– a variable cIdx specifying the chroma component of the current block,

– a variable nCS specifying the size, in chroma samples, of the current coding unit,

– a (nCS)x(nCS) array resSamples of chroma residual samples.

Output of this process is:

– a modified version of the (nCS)x(nCS) array of chroma residual samples.

[Ed. (KS): Derivation process for InterTUSplitDirection should be invoked before applying this process. ]

The variable splitChromaFlag is derived as follows:

– If split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 1 and log2TrafoSize is greater than Log2MinTrafoSize, splitChromaFlag is set equal to 1.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0 or log2TrafoSize is equal to Log2MinTrafoSize), splitChromaFlag is set equal to 0.

Depending splitChromaFlag, the following applies:

– If splitChromaFlag is equal to 1, the following ordered steps apply:

1. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If InterTUSplitDirection is equal to 2, the following applies.
   * The variable xB1 is set equal to xB + ( ( 1 << log2TrafoWidth ) >> 1 ).
   * The variable yB1 is set equal to yB.
   * The variable xB2 is set equal to xB.
   * The variable yB2 is set equal to yB + ( ( 1 << log2TrafoHeight ) >> 1 ).
   * The variable xB3 is set equal to xB1.
   * The variable yB3 is set equal to yB2.
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 1.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 1.
   * Otherwise (InterTUSplitDirection is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* InterTUSplitDirection.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − InterTUSplitDirection).
   * The variable log2TrafoWidth1 is set equal to (log2TrafoWidth − 2) \* InterTUSplitDirection.
   * The variable log2TrafoHeight1 is set equal to (log2TrafoHeight − 2) \* (1 − InterTUSplitDirection).

[Ed. (KS): Need to be updated on second depth for non square transform tree. In 16x16 CU, second TU depth should have four square transforms in a row or column. ]

1. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
2. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB1, yB1 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
3. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB2, yB2 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.
4. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location ( xC, yC ), the luma location ( xB3, yB3 ), the variable log2TrafoWidth set equal to log2TrafoWidth1, the variable log2TrafoHeight set equal to log2TrafoHeight1, the variable trafoDepth set equal to trafoDepth + 1, the variable cIdx, the variable nCS, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.

– Otherwise (splitChromaFlag is equal to 0), the following ordered steps apply:

1. The variable nW is set equal to ( 1 << log2TrafoWidth ) >> 1.
2. The variable nH is set equal to (1 << log2TrafoHeight ) >> 1.
3. The scaling and transformation process as specified in subclause 8.6.1 is invoked with the luma location ( xC + xB, yC +yB ), the variable trafoDepth, the variable cIdx, the transform width trafoWidth set equal to nW, and the transform height trafoHeight set equal to nH as the inputs and the output is a (nW)x(nH) array resSamplesBlock.
4. The array construction process as specified in subclause 8.6.5 is invoked with the luma location ( xB, yB ), the variable cIdx, the variable inputArrayWidth set equal to nW, the variable inputArrayHeight set equal to nH, the variable outputArraySize set equal to nCS, the (nW)x(nH) array resSamplesBlock, and the (nCS)x(nCS) array resSamples as the inputs and the output is a modified version of the (nCS)x(nCS) array resSamples.