H.264 / MPEG-4 Part 10 White Paper

Overview of H.264

1. Introduction

Broadcast television and home entertainment have been revolutionised by the advent of digital TV and DVD-video. These applications and many more were made possible by the standardisation of video compression technology. The next standard in the MPEG series, MPEG4, is enabling a new generation of internet-based video applications whilst the ITU-T H.263 standard for video compression is now widely used in videoconferencing systems.

MPEG4 (Visual) and H.263 are standards that are based on video compression ("video coding") technology from circa. 1995. The groups responsible for these standards, the Motion Picture Experts Group and the Video Coding Experts Group (MPEG and VCEG) are in the final stages of developing a new standard that promises to significantly outperform MPEG4 and H.263, providing better compression of video images together with a range of features supporting high-quality, low-bitrate streaming video. The history of the new standard, "Advanced Video Coding" (AVC), goes back at least 7 years.

After finalising the original H.263 standard for videotelephony in 1995, the ITU-T Video Coding Experts Group (VCEG) started work on two further development areas: a "short-term" effort to add extra features to H.263 (resulting in Version 2 of the standard) and a "long-term" effort to develop a new standard for low bitrate visual communications. The long-term effort led to the draft "H.26L" standard, offering significantly better video compression efficiency than previous ITU-T standards. In 2001, the ISO Motion Picture Experts Group (MPEG) recognised the potential benefits of H.26L and the Joint Video Team (JVT) was formed, including experts from MPEG and VCEG. JVT's main task is to develop the draft H.26L "model" into a full International Standard. In fact, the outcome will be two identical) standards: ISO MPEG4 Part 10 of MPEG4 and ITU-T H.264. The "official" title of the new standard is Advanced Video Coding (AVC); however, it is widely known by its old working title, H.26L and by its ITU document number, H.264 [1].

2. H.264 CODEC

In common with earlier standards (such as MPEG1, MPEG2 and MPEG4), the H.264 draft standard does not explicitly define a CODEC (enCOder / DECoder pair). Rather, the standard defines the syntax of an encoded video bitstream together with the method of decoding this bitstream. In practice, however, a compliant encoder and decoder are likely to include the functional elements shown in Figure 2-1 and Figure 2-2. Whilst the functions shown in these Figures are likely to be necessary for compliance, there is scope for considerable variation in the structure of the CODEC. The basic functional elements (prediction, transform, quantization, entropy encoding) are little different from previous standards (MPEG1, MPEG2, MPEG4, H.261, H.263); the important changes in H.264 occur in the details of each functional element.

The Encoder (Figure 2-1) includes two dataflow paths, a "forward" path (left to right, shown in blue) and a "reconstruction" path (right to left, shown in magenta). The dataflow path in the Decoder (Figure 2-2) is shown from right to left to illustrate the similarities between Encoder and Decoder.

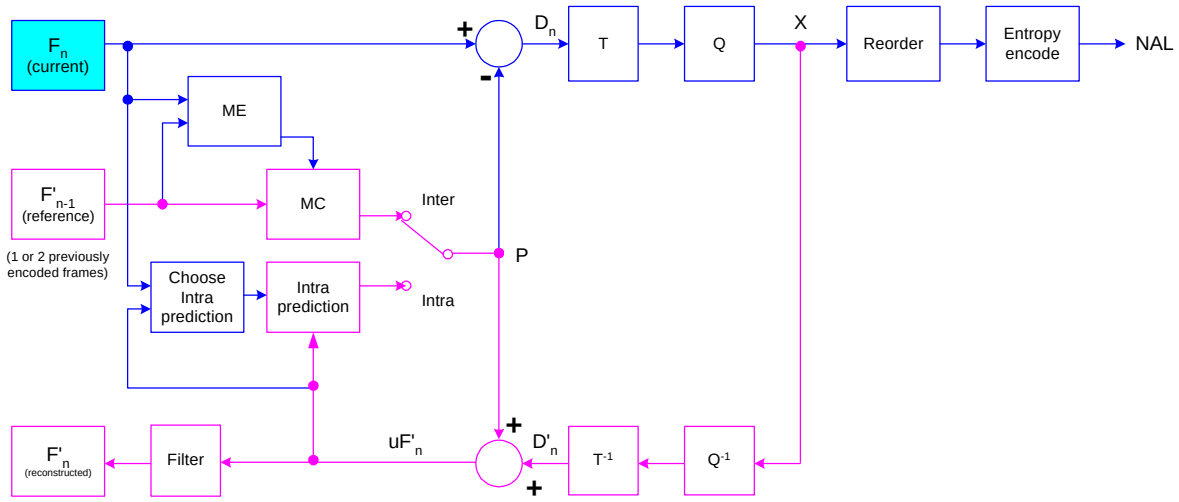


Figure 2-1 AVC Encoder

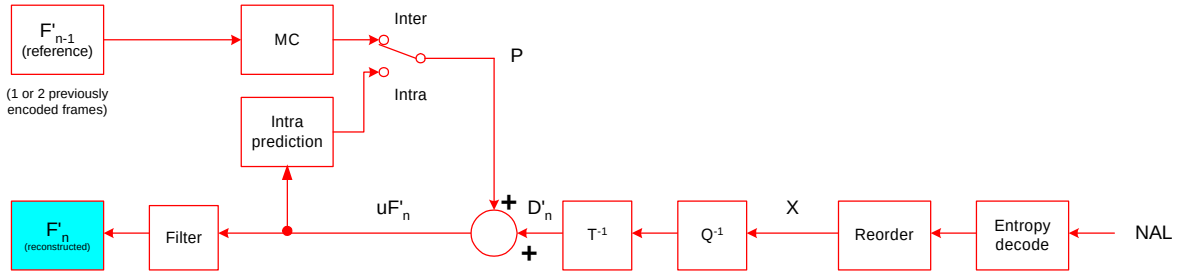


Figure 2-2 AVC Decoder

2.1 Encoder (forward path)

An input frame F_n is presented for encoding. The frame is processed in units of a macroblock (corresponding to 16x16 pixels in the original image). Each macroblock is encoded in **intra** or **inter** mode. In either case, a prediction macroblock P is formed based on a reconstructed frame. In Intra mode, P is formed from samples in the current frame n that have previously encoded, decoded and reconstructed (uF'_n in the Figures; note that the **unfiltered** samples are used to form P). In Inter mode, P is formed by motion-compensated prediction from one or more reference frame(s). In the Figures, the reference frame is shown as the previous encoded frame F'_{n-1} ; however, the predicton for each macroblock may be formed from one or two past or future frames (in time order) that have already been encoded and reconstructed.

The prediction P is subtracted from the current macroblock to produce a residual or difference macroblock D_n. This is transformed (using a block transform) and quantized to give X, a set of quantized transform coefficients. These coefficients are re-ordered and entropy encoded. The entropy-encoded coefficients, together with side information required to decode the macroblock (such as the macroblock prediction mode, quantizer step size, motion vector information describing how the macroblock was motion-compensated, etc) form the compressed bitstream. This is passed to a Network Abstraction Layer (NAL) for transmission or storage.

2.2 Encoder (reconstruction path)

The quantized macroblock coefficients X are decoded in order to reconstruct a frame for encoding of further macroblocks. The coefficients X are re-scaled (Q^{-1}) and inverse transformed (T^{-1}) to produce a difference macroblock D_n'. This is not identical to the original difference macroblock D_n ; the quantization process introduces losses and so D_n' is a distorted version of D_n.

The prediction macroblock P is added to D_n' to create a reconstructed macroblock uF'_n (a distorted version of the original macroblock). A filter is applied to reduce the effects of blocking distortion and reconstructed reference frame is created from a series of macroblocks F'_n.

2.3 Decoder

The decoder receives a compressed bitstream from the NAL. The data elements are entropy decoded and reordered to produce a set of quantized coefficients X. These are rescaled and inverse transformed to give D_n' (this identical to the D_n' shown in the Encoder). Using the header information decoded from the bitstream, the decoder creates a prediction macroblock P, identical to the original prediction P formed in the encoder. P is added to D_n' to produce uF'_n which this is filtered to create the decoded macroblock F'_n.

It should be clear from the Figures and from the discussion above that the purpose of the reconstruction path in the encoder is to ensure that both encoder and decoder use identical reference frames to create the prediction P. If this is not the case, then the predictions P in encoder and decoder will not be identical, leading to an increasing error or "drift" between the encoder and decoder.

3. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document JVT-E022, September 2002

H.264 / MPEG-4 Part 10 White Paper

Revised April 03

Prediction of Intra Macroblocks

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1,2] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". This document describes the methods of predicting intra-coded macroblocks in an H.264 CODEC.

If a block or macroblock is encoded in intra mode, a prediction block is formed based on previously encoded and reconstructed (but **un-filtered**) blocks. This prediction block P is subtracted from the current block prior to encoding. For the luminance (luma) samples, P may be formed for each 4x4 sub-block or for a 16x16 macroblock. There are a total of 9 optional prediction modes for each 4x4 luma block; 4 optional modes for a 16x16 luma block; and one mode that is always applied to each 4x4 chroma block.

2. 4x4 luma prediction modes

Figure 1 shows a luminance macroblock in a QCIF frame and a 4x4 luma block that is required to be predicted. The samples above and to the left have previously been encoded and reconstructed and are therefore available in the encoder and decoder to form a prediction reference. The prediction block P is calculated based on the samples labelled A-M in Figure 2, as follows. Note that in some cases, not all of the samples A-M are available within the current slice: in order to preserve independent decoding of slices, only samples within the current slice are available for prediction. DC prediction (mode 0) is modified depending on which samples A-M are available; the other modes (1-8) may only be used if all of the required prediction samples are available (except that, if E, F, G and H are not available, their value is copied from sample D).

The arrows in Figure 3 indicate the direction of prediction in each mode. For modes 3-8, the predicted samples are formed from a weighted average of the prediction samples A-Q. The encoder may select the prediction mode for each block that minimizes the residual between P and the block to be encoded.

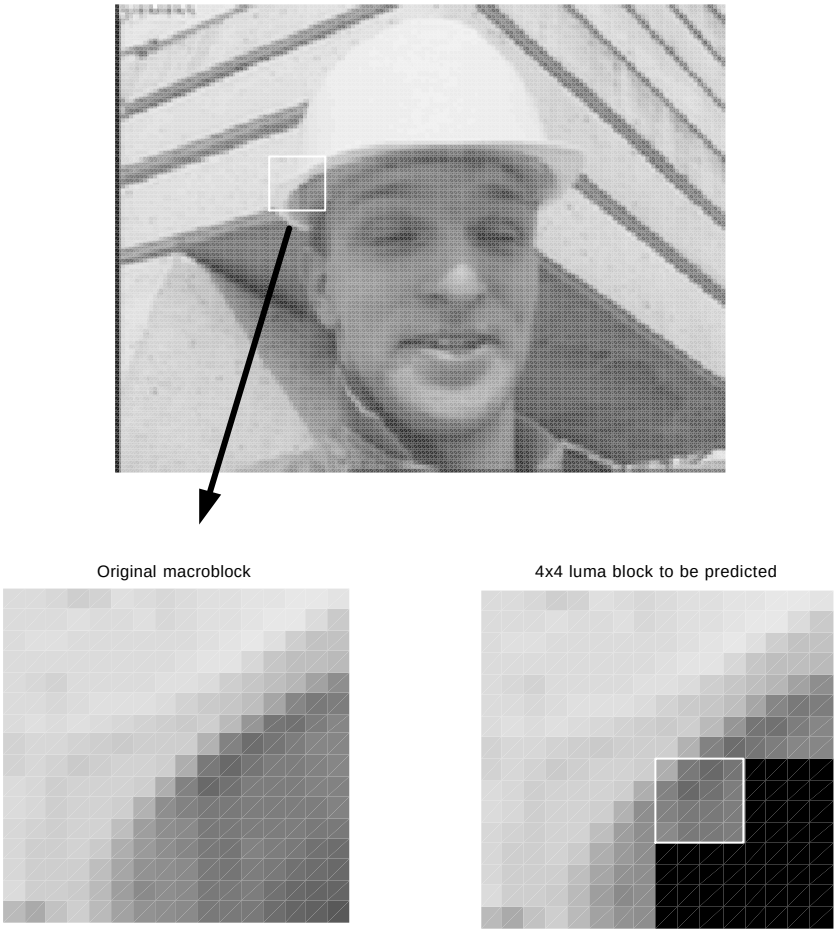


Figure 1 Original macroblock and 4x4 luma block to be predicted

M	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
I	a	b	c	d				
J	e	f	g	h				
K	i	j	k	l				
L	m	n	o	p				

Figure 2 Labelling of prediction samples (4x4)

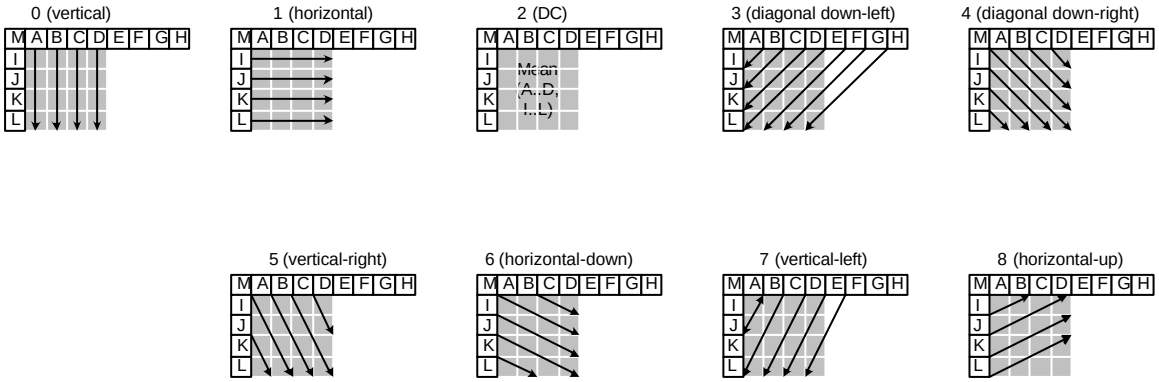


Figure 3 4x4 luma prediction modes

Example: The 9 prediction modes (0-8) are calculated for the 4x4 block shown in Figure 1. Figure 4 shows the prediction block P created by each of the predictions. The Sum of Absolute Errors (SAE) for each prediction indicates the magnitude of the prediction error. In this case, the best match to the actual current block is given by mode 7 (vertical-right) because this mode gives the smallest SAE; a visual comparison shows that the P block appears quite similar to the original 4x4 block.

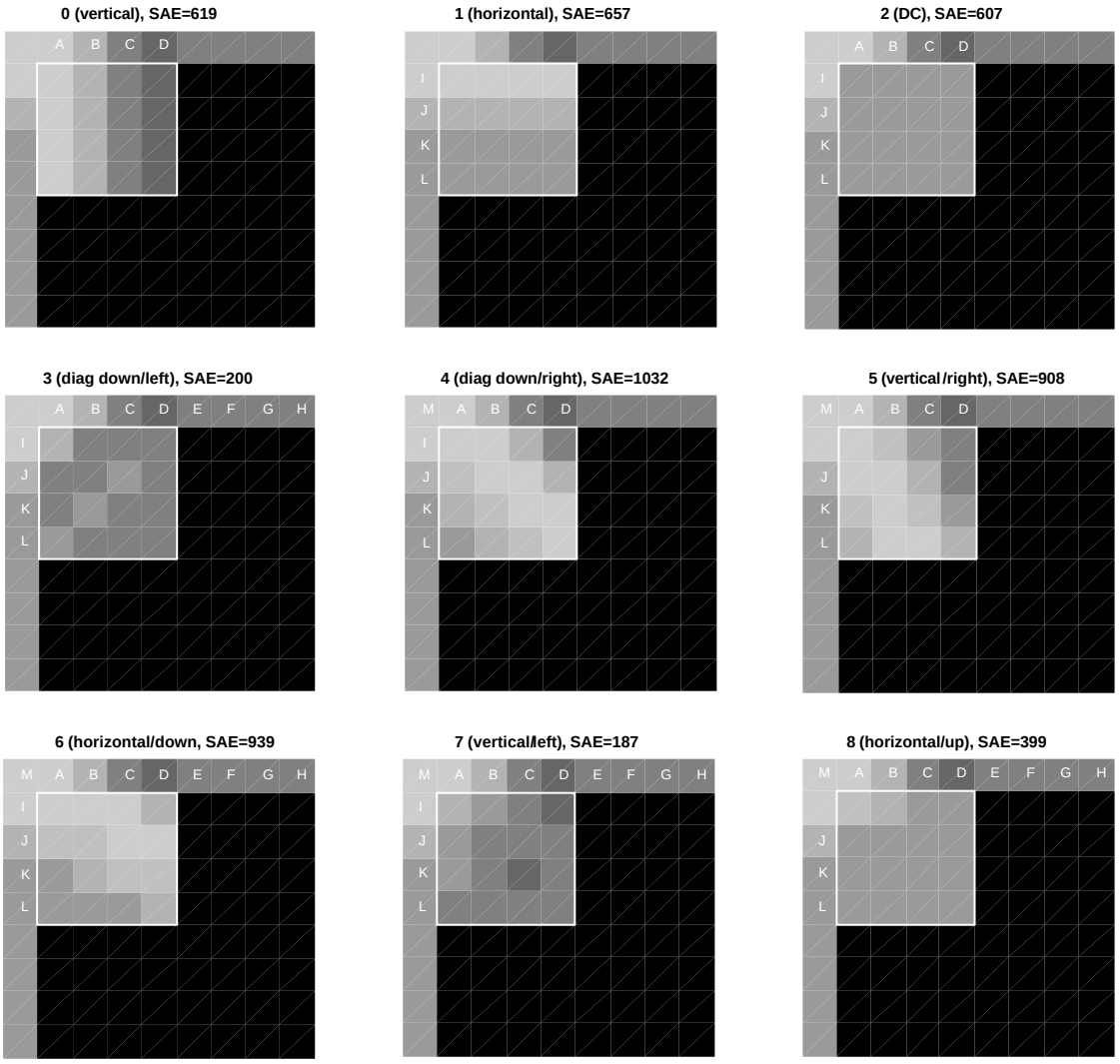
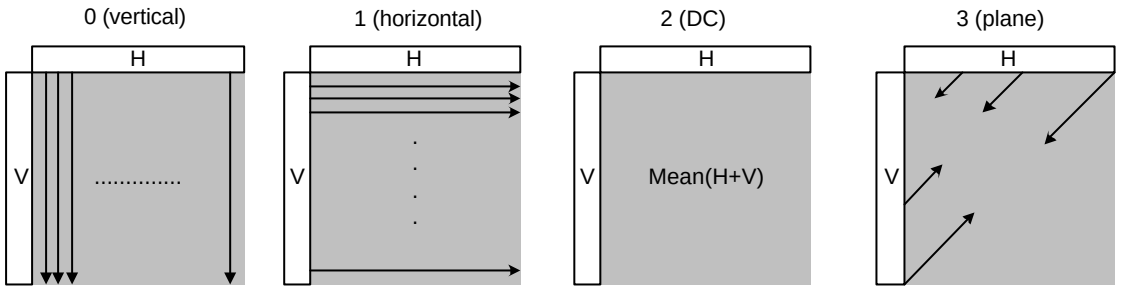


Figure 4 Prediction blocks P (4x4)

3. 16x16 luma prediction modes

As an alternative to the 4x4 luma modes described above, the entire 16x16 luma component of a macroblock may be predicted. Four modes are available, shown in diagram form in Figure 5:

Mode 0 (vertical): extrapolation from upper samples (H).
Mode 1 (horizontal): extrapolation from left samples (V).
Mode 2 (DC): mean of upper and left-hand samples (H+V).
Mode 4 (Plane): a linear "plane" function is fitted to the upper and left-hand samples H and V. This works well in areas of smoothly-varying luminance.



H.26L Intra 16x16 prediction modes (all predicted from pixels H and V)

Figure 5 Intra 16x16 prediction modes

Example:

Figure 6 shows a luminance macroblock with the previously-encoded samples at the upper and left-hand edges. The results of prediction (**Figure 7**) indicate that the best match is given by mode 3. Intra 16x16 mode works best in homogeneous areas of an image.

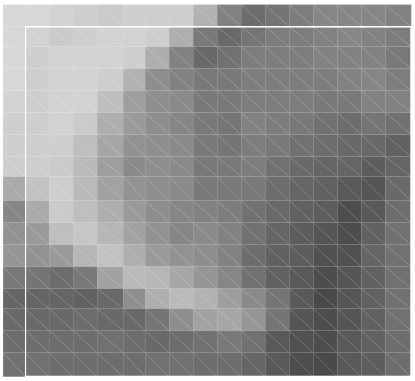


Figure 6 16x16 macroblock

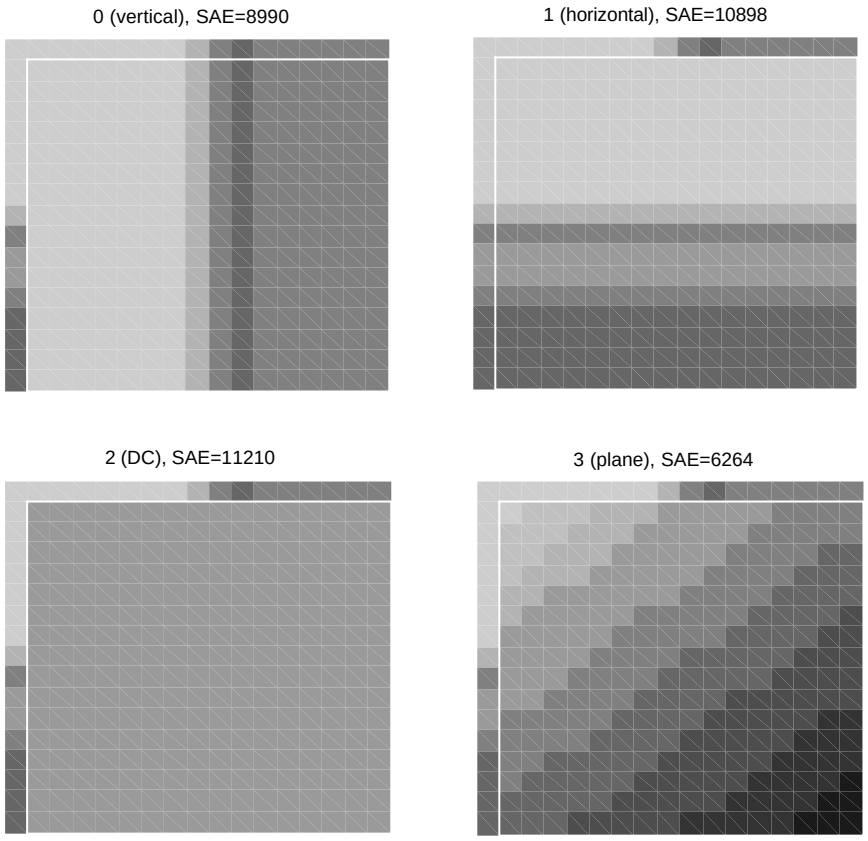


Figure 7 Intra 16x16 predictions

4. 8x8 chroma prediction mode

Each 8x8 chroma component of a macroblock is predicted from chroma samples above and/or to the left that have previously been encoded and reconstructed. The 4 prediction modes are very similar to the 16x16 luma prediction modes described in section 3 and illustrated in Figure 5, except that the order of mode numbers is different: DC (mode 0), horizontal (mode 1), vertical (mode 2) and plane (mode 3). The same prediction mode is always applied to both chroma blocks.

Note: if any of the 8x8 blocks in the luma component are coded in Intra mode, both chroma blocks are Intra coded.

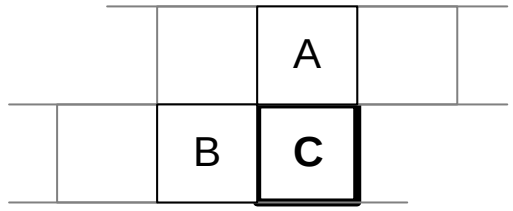
5. Encoding intra prediction modes

The choice of intra prediction mode for each 4x4 block must be signalled to the decoder and this could potentially require a large number of bits. However, intra modes for neighbouring 4x4 blocks are highly correlated. For example, if previously-encoded 4x4 blocks A and B in Figure 8 were predicted using mode 2, it is likely that the best mode for block C (current block) is also mode 2.

For each current block C, the encoder and decoder calculate the most_probable_mode. If A and B are both coded in 4x4 intra mode and are both within the current slice, most_probable_mode is the minimum of the prediction modes of A and B; otherwise most_probable_mode is set to 2 (DC prediction).

The encoder sends a flag for each 4x4 block, use_most_probable_mode. If the flag is “1”, the parameter most_probable_mode is used. If the flag is “0”, another parameter remaining_mode_selector is sent to indicate a change of mode. If remaining_mode_selector is smaller than the current most_probable_mode then the prediction mode is set to remaining_mode_selector;

otherwise the prediction mode is set to remaining_mode_selector+1. In this way, only 8 values of remaining_mode_selector are required (0 to 7) to signal the current intra mode (0 to 8).



.

Figure 8 Adjacent 4x4 intra coded blocks

6. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document **JVT-F100**, December 2002

2 Iain E G Richardson, "H.264 and MPEG-4 Video Compression", John Wiley & Sons, to be published late 2003

H.264 / MPEG-4 Part 10 White Paper

Prediction of Inter Macroblocks in P-slices

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". This document describes the methods of predicting inter-coded macroblocks in P-slices in an H.264 CODEC.

Inter prediction creates a prediction model from one or more previously encoded video frames. The model is formed by shifting samples in the reference frame(s) (motion compensated prediction). The AVC CODEC uses block-based motion compensation, the same principle adopted by every major coding standard since H.261. Important differences from earlier standards include the support for a range of block sizes (down to 4x4) and fine sub-pixel motion vectors (1/4 pixel in the luma component).

2. Tree structured motion compensation

AVC supports motion compensation block sizes ranging from 16x16 to 4x4 luminance samples with many options between the two. The luminance component of each macroblock (16x16 samples) may be split up in 4 ways as shown in Figure 2-1: 16x16, 16x8, 8x16 or 8x8. Each of the sub-divided regions is a macroblock partition. If the 8x8 mode is chosen, each of the four 8x8 macroblock partitions within the macroblock may be split in a further 4 ways as shown in Figure 2-2: 8x8, 8x4, 4x8 or 4x4 (known as macroblock sub-partitions). These partitions and sub-partitions give rise to a large number of possible combinations within each macroblock. This method of partitioning macroblocks into motion compensated sub-blocks of varying size is known as **tree structured motion compensation**.

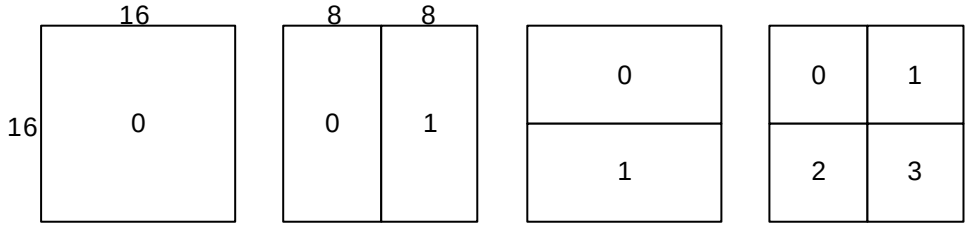


Figure 2-1 Macroblock partitions: 16x16, 8x16, 16x8, 8x8

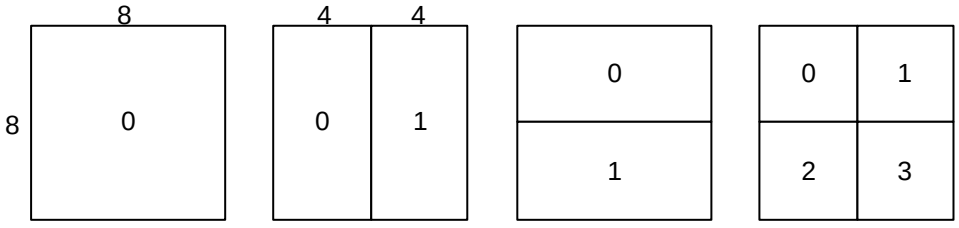


Figure 2-2 Macroblock sub-partitions: 8x8, 4x8, 8x4, 4x4

A separate motion vector is required for each partition or sub-partition. Each motion vector must be coded and transmitted; in addition, the choice of partition(s) must be encoded in the compressed bitstream. Choosing a large partition size (e.g. 16x16, 16x8, 8x16) means that a small number of bits are required to signal the choice of motion vector(s) and the type of partition; however, the motion compensated residual may contain a significant amount of energy in frame areas with high detail. Choosing a small partition size (e.g. 8x4, 4x4, etc.) may give a lower-energy residual after motion compensation but requires a larger number of bits to signal the motion vectors and choice of partition(s). The choice of partition size therefore has a significant impact on compression

performance. In general, a large partition size is appropriate for homogeneous areas of the frame and a small partition size may be beneficial for detailed areas.

The resolution of each chroma component in a macroblock (Cr and Cb) is half that of the luminance (luma) component. Each chroma block is partitioned in the same way as the luma component, except that the partition sizes have exactly half the horizontal and vertical resolution (an 8x16 partition in luma corresponds to a 4x8 partition in chroma; an 8x4 partition in luma corresponds to 4x2 in chroma; and so on). The horizontal and vertical components of each motion vector (one per partition) are halved when applied to the chroma blocks.

Example: Figure 2-3 shows a residual frame (without motion compensation). The AVC reference encoder selects the "best" partition size for each part of the frame, i.e. the partition size that minimizes the coded residual and motion vectors. The macroblock partitions chosen for each area are shown superimposed on the residual frame. In areas where there is little change between the frames (residual appears grey), a 16x16 partition is chosen; in areas of detailed motion (residual appears black or white), smaller partitions are more efficient.

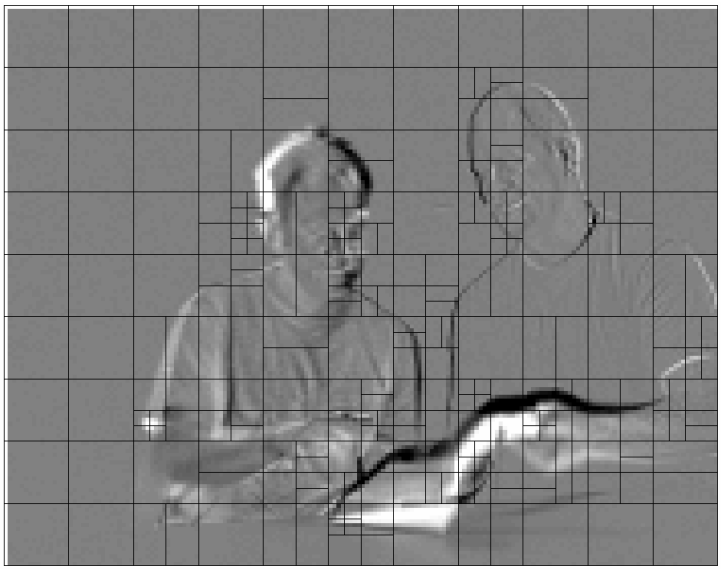


Figure 2-3 Residual (without MC) showing optimum choice of partitions

3. Sub-pixel motion vectors

Each partition in an inter-coded macroblock is predicted from an area of the same size in a reference picture. The offset between the two areas (the motion vector) has ¼-pixel resolution (for the luma component). The luma and chroma samples at sub-pixel positions do not exist in the reference picture and so it is necessary to create them using interpolation from nearby image samples. Figure 3-1 gives an example. A 4x4 sub-partition in the current frame (a) is to be predicted from a neighbouring region of the reference picture. If the horizontal and vertical components of the motion vector are integers (b), the relevant samples in the reference block actually exist (grey dots). If one or both vector components are fractional values (c), the prediction samples (grey dots) are generated by interpolation between adjacent samples in the reference frame (white dots).

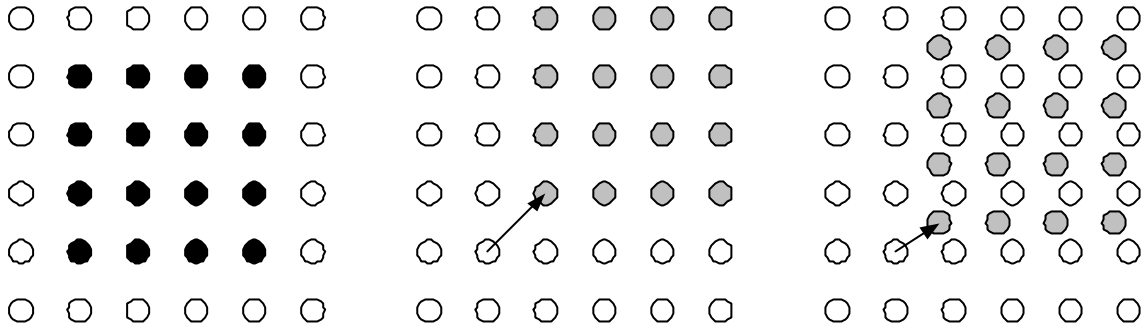


Figure 3-1 Example of integer and sub-pixel prediction

Sub-pixel motion compensation can provide significantly better compression performance than integer-pixel compensation, at the expense of increased complexity. Quarter-pixel accuracy outperforms half-pixel accuracy.

In the luma component, the sub-pixel samples at half-pixel positions are generated first and are interpolated from neighbouring integer-pixel samples using a 6-tap Finite Impulse Response filter. This means that each half-pixel sample is a weighted sum of 6 neighbouring integer samples. Once all the half-pixel samples are available, each quarter-pixel sample is produced using bilinear interpolation between neighbouring half- or integer-pixel samples.

If the video source sampling is 4:2:0, 1/8 pixel samples are required in the chroma components (corresponding to ¼-pixel samples in the luma). These samples are interpolated (linear interpolation) between integer-pixel chroma samples.

4. Motion vector prediction

Encoding a motion vector for each partition can take a significant number of bits, especially if small partition sizes are chosen. Motion vectors for neighbouring partitions are often highly correlated and so each motion vector is predicted from vectors of nearby, previously coded partitions. A predicted vector, MVp, is formed based on previously calculated motion vectors. MVD, the difference between the current vector and the predicted vector, is encoded and transmitted. The method of forming the prediction MVp depends on the motion compensation partition size and on the availability of nearby vectors. The "basic" predictor is the median of the motion vectors of the macroblock partitions or sub-partitions immediately above, diagonally above and to the right, and immediately left of the current partition or sub-partition. The predictor is modified if (a) 16x8 or 8x16 partitions are chosen and/or (b) if some of the neighbouring partitions are not available as predictors. If the current macroblock is skipped (not transmitted), a predicted vector is generated as if the MB was coded in 16x16 partition mode.

At the decoder, the predicted vector MVp is formed in the same way and added to the decoded vector difference MVD. In the case of a skipped macroblock, there is no decoded vector and so a motion-compensated macroblock is produced according to the magnitude of MVp.

5. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document **JVT-G050**, March 2003

H.264 / MPEG-4 Part 10 White Paper

Transform and quantization

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". This document describes the transform and quantization processes defined, or implied, by the standard.

Each residual macroblock is transformed, quantized and coded. Previous standards such as MPEG-1, MPEG-2, MPEG-4 and H.263 made use of the 8x8 Discrete Cosine Transform (DCT) as the basic transform. The "baseline" profile of H.264 uses three transforms depending on the type of residual data that is to be coded: a transform for the 4x4 array of luma DC coefficients in intra macroblocks (predicted in 16x16 mode), a transform for the 2x2 array of chroma DC coefficients (in any macroblock) and a transform for all other 4x4 blocks in the residual data. If the optional "adaptive block size transform" mode is used, further transforms are chosen depending on the motion compensation block size (4x8, 8x4, 8x8, 16x8, etc).

Data within a macroblock are transmitted in the order shown in Figure 1-1. If the macroblock is coded in 16x16 Intra mode, then the block labelled "-1" is transmitted first, containing the DC coefficient of each 4x4 luma block. Next, the luma residual blocks 0-15 are transmitted in the order shown (with the DC coefficient set to zero in a 16x16 Intra macroblock). Blocks 16 and 17 contain a 2x2 array of DC coefficients from the Cb and Cr chroma components respectively. Finally, chroma residual blocks 18-25 (with zero DC coefficients) are sent.

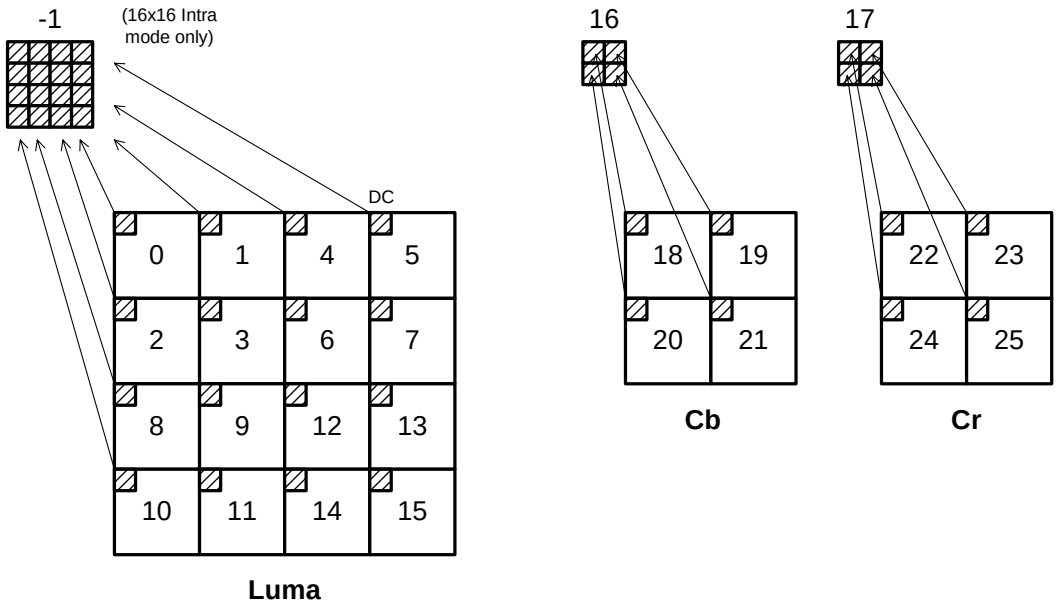


Figure 1-1 Scanning order of residual blocks within a macroblock

2. 4x4 residual transform and quantization (blocks 0-15, 18-25)

This transform operates on 4x4 blocks of residual data (labelled 0-15 and 18-25 in Figure 1-1) after motion-compensated prediction or Intra prediction. The transform is based on the DCT but with some fundamental differences:

1. It is an integer transform (all operations can be carried out with integer arithmetic, without loss of accuracy).

2. The inverse transform is fully specified in the H.264 standard and if this specification is followed correctly, mismatch between encoders and decoders should not occur.
3. The core part of the transform is multiply-free, i.e. it only requires additions and shifts.
4. A scaling multiplication (part of the complete transform) is integrated into the quantizer (reducing the total number of multiplications).

The entire process of transform and quantization can be carried out using 16-bit integer arithmetic and only a single multiply per coefficient, without any loss of accuracy.

2.1 Development from the 4x4 DCT

The 4x4 DCT of an input array **X** is given by:

$$\mathbf{Y} = \mathbf{AXA}^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \begin{bmatrix} & & & \\ & \mathbf{X} & & \\ & & & \end{bmatrix} \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix}$$

Equation 2-1

where:

$$a = \frac{1}{2}$$

$$b = \sqrt{\frac{1}{2}} \cos\left(\frac{\pi}{8}\right)$$

$$c = \sqrt{\frac{1}{2}} \cos\left(\frac{3\pi}{8}\right)$$

This matrix multiplication can be factorised [2] to the following equivalent form (Equation 2-2):

$$\mathbf{Y} = \left(\mathbf{CXC}^T\right) \otimes \mathbf{E} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} \begin{bmatrix} & & & \\ & \mathbf{X} & & \\ & & & \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix}$$

Equation 2-2

$\mathbf{CXC}^\mathbf{T}$ is a "core" 2-D transform. **E** is a matrix of scaling factors and the symbol ⊗ indicates that each element of ($\mathbf{CXC}^\mathbf{T}$) is multiplied by the scaling factor in the same position in matrix **E** (scalar multiplication rather than matrix multiplication). The constants *a* and *b* are as before; *d* is *c/b* (approximately 0.414).

To simplify the implementation of the transform, *d* is approximated by 0.5. To ensure that the transform remains orthogonal, *b* also needs to be modified so that:

$$a = \frac{1}{2}$$

$$b = \sqrt{\frac{2}{5}}$$

$$d = \frac{1}{2}$$

The 2nd and 4th rows of matrix **C** and the 2nd and 4th columns of matrix $\mathbf{C}^T$ are scaled by a factor of 2 and the post-scaling matrix **E** is scaled down to compensate. (This avoids multiplications by ½ in the "core" transform $\mathbf{CXC}^T$ which would result in loss of accuracy using integer arithmetic). The final forward transform becomes:

$$\mathbf{Y} = \mathbf{C}_f \mathbf{X} \mathbf{C}_f^T \otimes \mathbf{E}_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \mathbf{X} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \\ a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \end{bmatrix}$$

Equation 2-3

This transform is an approximation to the 4x4 DCT. Because of the change to factors *d* and *b*, the output of the new transform will not be identical to the 4x4 DCT.

The inverse transform (defined in [1]) is given by:

$$\mathbf{X}' = \mathbf{C}_i^T \left(\mathbf{Y} \otimes \mathbf{E}_i \right) \mathbf{C}_i = \begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix} \left(\begin{bmatrix} & \mathbf{Y} & \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix}$$

Equation 2-4

This time, **Y** is **pre-scaled** by multiplying each coefficient by the appropriate weighting factor from matrix $\mathbf{E_i}$. Note the factors +/-½ in the matrices **C** and $\mathbf{C}^T$; these can be implemented by a right-shift without a significant loss of accuracy because the coefficients **Y** are pre-scaled.

The forward and inverse transforms are orthogonal, i.e. $\mathbf{T^{-1}(T(X))} = \mathbf{X}$.

2.2 Quantization

H.264 uses a scalar quantizer . The definition and implementation are complicated by the requirements to (a) avoid division and/or floating point arithmetic and (b) incorporate the post- and pre-scaling matrices $\mathbf{E_f}$ and $\mathbf{E_i}$ described above.

The basic forward quantizer operation is as follows:

$Z_{ij} = \text{round}(Y_{ij}/Qstep)$

where Y_{ij} is a coefficient of the transform described above, Qstep is a quantizer step size and Z_{ij} is a quantized coefficient.

A total of 52 values of Qstep are supported by the standard and these are indexed by a Quantization Parameter, QP. The values of Qstep corresponding to each QP are shown in Table 2-1. Note that Qstep doubles in size for every increment of 6 in QP; Qstep increases by 12.5% for each increment of 1 in QP. The wide range of quantizer step sizes makes it possible for an encoder to accurately and flexibly control the trade-off between bit rate and quality. The values of QP may be different for luma and chroma; both parameters are in the range 0-51 but QP_{Chroma} is derived from QP_Y so that it QP_C is less that QP_Y for values of QP_Y above 30. A user-defined offset between QP_Y and QP_C may be signalled in a Picture Parameter Set.

Table 2-1 Quantization step sizes in H.264 CODEC

QP	0	1	2	3	4	5	6	7	8	9	10	11	12	….
QStep	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625	1.75	2	2.25	2.5	….
QP	…	18	…	24	…	30	…	36	…	42	…	48	…	51
QStep		5		10		20		40		80		160		224

The post-scaling factor a^2, ab/2 or $b^2/4$ (Equation 2-3) is incorporated into the forward quantizer. First, the input block **X** is transformed to give a block of unscaled coefficients $\mathbf{W = CXC^T}$. Then, each coefficient W_{ij} is quantized and scaled in a single operation:

$$Z_{ij} = \text{round}\left(W_{ij} . \frac{PF}{Qstep} \right)$$

Equation 2-5

PF is a^2, ab/2 or $b^2/4$ depending on the position (i,j) (see Equation 2-3):

Position	PF
(0,0), (2,0), (0,2) or (2,2)	a^2
(1,1), (1,3), (3,1) or (3,3)	$b^2/4$
Other	ab/2

The factor (PF/Qstep) is implemented in the H.264 reference model software [3] as a multiplication by MF (a multiplication factor) and a right-shift, thus avoiding any division operations:

$$Z_{ij} = \text{round}\left(W_{ij} . \frac{MF}{2^{qbits}} \right)$$

Equation 2-6

where $\frac{MF}{2^{qbits}} = \frac{PF}{Qstep}$ and qbits = 15+floor(QP/6)

In integer arithmetic, Equation 2-6 can be implemented as follows:

$|Z_{ij}| = (|W_{ij}|.MF + f) >> qbits$
$sign(Z_{ij}) = sign(W_{ij})$

Equation 2-7

where >> indicates a binary shift right. In the reference model software, f is $2^{qbits}/3$ for Intra blocks or $2^{qbits}/6$ for Inter blocks.

Example:

QP = 4, hence Qstep = 1.0
(i,j) = (0,0), hence PF = a^2 = 0.25
qbits = 15, hence 2^{qbits} = 32768

$\frac{MF}{2^{qbits}} = \frac{PF}{Qstep}$, hence MF = (32768x0.25)/1 = **8192**

The first 6 values of MF can be calculated as follows, depending on QP and the coefficient position (i,j):

Table 2-2 Multiplication factor MF

QP	Positions (0,0),(2,0),(2,2),(0,2)	Positions (1,1),(1,3),(3,1),(3,3)	Other positions
0	13107	5243	8066
1	11916	4660	7490
2	10082	4194	6554
3	9362	3647	5825
4	8192	3355	5243
5	7282	2893	4559

The 2nd and 3rd columns of this table (positions with factors $b^2/4$ and ab/2) have been modified slightly[1] from the results of Equation 2-6.

For QP>5, the factors MF remain unchanged but the divisor 2^{qbits} increases by a factor of 2 for each increment of 6 in QP. For example, qbits=16 for 6≤QP≤11; qbits=17 for 12≤QP≤17; and so on.

2.3 Rescaling

The basic rescale (or "inverse quantizer") operation is:

$Y'_{ij} = Z_{ij}.Qstep$

Equation 2-8

The pre-scaling factor for the inverse transform (matrix $\mathbf{E_i}$, containing values a^2, ab and b^2 depending on the coefficient position) is incorporated in this operation, together with a further constant scaling factor of 64 to avoid rounding errors:

$W'_{ij} = Z_{ij}.Qstep.PF.64$

Equation 2-9

W'_{ij} is a scaled coefficient which is then transformed by the "core" inverse transform ($\mathbf{C_i^T W C_i}$: see Equation 2-4). The values at the output of the inverse transform are divided by 64 to remove the scaling factor (this can be implemented using only an addition and a right-shift).

[1] It is acceptable to modify a forward quantizer in order to improve perceptual quality at the decoder, since only the rescaling (inverse quantizer) process is standardised.

The H.264 standard does not specify Qstep or PF directly. Instead, the parameter V=(Qstep.PF.64) are defined for 0≤QP≤5 and each coefficient position and the rescaling operation is:

$W'_{ij} = Z_{ij}.V_{ij}.2^{floor(QP/6)}$

Equation 2-10

Example:

QP=3, hence Qstep = 0.875 and $2^{floor(QP/6)} = 1$
(i,j)=(1,2), hence PF = ab = 0.3162
V=(Qstep.PF.64)= 0.875x0.3162x65 ≅ 18
$W'_{ij} = Z_{ij}$x18x1

The values of V for 0≤QP≤5 are defined in the standard as follows:

Table 2-3 Rescaling factor V

QP	Positions (0,0),(2,0),(2,2),(0,2)	Positions (1,1),(1,3),(3,1),(3,3)	Other positions
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	29	23

The factor $2^{floor(QP/6)}$ in Equation 2-10 makes the rescaled output increase by a factor of 2 for every increment of 6 in QP.

3. 4x4 luma DC coefficient transform and quantization (16x16 Intra-mode only)

If the macroblock is encoded in 16x16 Intra prediction mode (where the entire 16x16 luminance component is predicted from neighbouring pixels), each 4x4 residual block is first transformed using the "core" transform described above ($\mathbf{C_f X C_f^T}$). The DC coefficient of each 4x4 block is then transformed again using a 4x4 Hadamard transform:

$$\mathbf{Y}_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \mathbf{W}_D \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) / 2$$

Equation 3-1

$\mathbf{W_D}$ is the block of 4x4 DC coefficients and $\mathbf{Y_D}$ is the block after transformation. The output coefficients $Y_{D(i,j)}$ are divided by 2 (with rounding).

The output coefficients $Y_{D(i,j)}$ are then quantized to produce a block of quantized DC coefficients:

$|Z_{D(i,j)}| = (|Y_{D(i,j)}|.MF_{(0,0)} + 2f) >> (qbits+1)$
$sign(Z_{D(i,j)}) = sign(Y_{D(i,j)})$

Equation 3-2

where MF, f and qbits are defined as before and MF depends on the position (i,j) within the 4x4 DC coefficient block as before.

At the decoder, an inverse Hadamard transform is applied **followed by** rescaling (note that the order is not reversed as might be expected):

$$\mathbf{W}_{QD} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} & \mathbf{Z}_D & \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right)$$

If QP is greater than or equal to 12, rescaling is performed by:

$W'_{D(i,j)} = W_{QD(i,j)}.V_{(0,0)}.2^{floor(QP/6)-2}$

If QP is less than 12, rescaling is performed by:

$W'_{D(i,j)} = [W_{QD(i,j)}.V_{(0,0)} + 2^{1-floor(QP/6)}]>>(2-floor(QP/6)$

V is defined as before. The rescaled DC coefficients $\mathbf{W'_D}$ are then inserted into their respective 4x4 blocks and each 4x4 block of coefficients is inverse transformed using the core DCT-based inverse transform ($\mathbf{C_i^T W'C_i}$).

In an intra-coded macroblock, much of the energy is concentrated in the DC coefficients and this extra transform helps to de-correlate the 4x4 luma DC coefficients (i.e. to take advantage of the correlation between the coefficients).

4. 2x2 chroma DC coefficient transform and quantization

Each chroma component in a macroblock is made up of four 4x4 blocks of samples. Each 4x4 block is transformed as described in section 2. The DC coefficients of each 4x4 block of coefficients are grouped in a 2x2 block ($\mathbf{W_D}$) and are further transformed prior to quantization:

$$\mathbf{Y}_D = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \mathbf{W}_D & \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Equation 4-1

Quantization of the 2x2 output block $\mathbf{Y_D}$ is performed by:

$|Z_{D(i,j)}| = (|Y_{D(i,j)}|.MF_{(0,0)} + 2f) >> (qbits+1)$
$sign(Z_{D(i,j)}) = sign(Y_{D(i,j)})$

Equation 4-2

where MF, f and qbits are defined as before.

During decoding, the inverse transform is applied before rescaling:

$$\mathbf{W}_{QD} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \mathbf{Z}_D & \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Equation 4-3

If QP is greater than or equal to 6, rescaling is performed by:

$W'_{D(i,j)} = W_{QD(i,j)}.V_{(0.0)}.2^{floor(QP/6)-1}$

If QP is less than 6, rescaling is performed by:

$W'_{D(i,j)} = [W_{QD(i,j)}.V_{(0,0)}]>>1$

The rescaled coefficients are replaced in their respective 4x4 blocks of chroma coefficients which are then transformed as above ($\mathbf{C_i^T W' C_i}$). As with the Intra luma DC coefficients, the extra transform helps to de-correlate the 2x2 chroma DC coefficients and hence improves compression performance.

5. The complete transform, quantization, rescaling and inverse transform process

The complete process from input residual block **X** to output residual block **X'** is described below and illustrated in Figure 5-1.

Encoding:

1. Input: 4x4 residual samples: **X**

2. Forward "core" transform: $\mathbf{W = C_f X C_f^T}$
(followed by forward transform for Chroma DC or Intra-16 Luma DC coefficients)

3. Post-scaling and quantization: $\mathbf{Z = W}.\frac{PF}{Qstep.2^{qbits}}$

(modified for Chroma DC or Intra-16 Luma DC)

Decoding:

(inverse transform for Chroma DC or Intra-16 Luma DC coefficients)
4. Re-scaling (incorporating inverse transform pre-scaling):
$\mathbf{W'} = \mathbf{Z}.Qstep.PF.64$
(modified for Chroma DC or Intra-16 Luma DC)

5. Inverse "core" transform: $\mathbf{X' = C_i^T W' C_i}$

6. Post-scaling: $\mathbf{X''} = round(X'/64)$

7. Output: 4x4 residual samples: **X''**

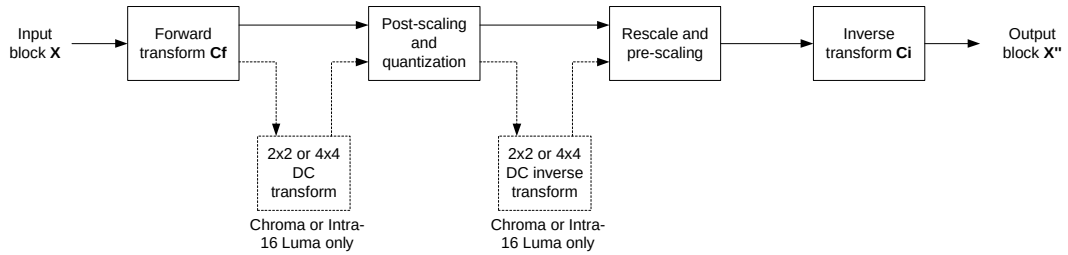


Figure 5-1 Transform, quantization, rescale and inverse transform flow diagram

Example (luma 4x4 residual block, Inter mode):

QP = 10

Input block **X:**

	j=0	1	2	3
i=0	5	11	8	10
1	9	8	4	12
2	1	10	11	4
3	19	6	15	7

Output of "core" transform **W:**

	j=0	1	2	3
i=0	140	-1	-6	7
1	-19	-39	7	-92
2	22	17	8	31
3	-27	-32	-59	-21

MF = 8192, 3355 or 5243 (depending on the coefficient position) and qbits=16. Output of forward quantizer **Z**:

	j=0	1	2	3
i=0	17	0	-1	0
1	-1	-2	0	-5
2	3	1	1	2
3	-2	-1	-5	-1

V = 16, 25 or 20 (depending on position) and $2^{\text{floor}(QP/6)} = 2^1 = 2$. Output of rescale **W':**

	j=0	1	2	3
i=0	544	0	-32	0
1	-40	-100	0	-250
2	96	40	32	80
3	-80	-50	-200	-50

Output of "core" inverse transform **X''** (after division by 64 and rounding)**:**

	j=0	1	2	3
i=0	4	13	8	10
1	8	8	4	12
2	1	10	10	3
3	18	5	14	7

6. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document JVT-F100, December 2002

2 A. Hallapuro and M. Karczewicz, "Low complexity transform and quantization – Part 1: Basic Implementation", JVT document JVT-B038, February 2001

3 JVT Reference Software version 4.0, ftp://ftp.imtc-files.org/jvt-experts/reference_software/

H.264 / MPEG-4 Part 10 White Paper

Reconstruction Filter

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". This document describes the methods of filtering reconstructed blocks in an H.264 CODEC. Note that the H.264 draft standard is not yet finalised and so readers are encouraged to refer to the latest version of the standard.

2. Description of reconstruction filter

A filter is applied to every decoded macroblock in order to reduce blocking distortion. The deblocking filter is applied after the inverse transform in the encoder (before reconstructing and storing the macroblock for future predictions) and in the decoder (before reconstructing and displaying the macroblock). The filter has two benefits: (1) block edges are smoothed, improving the appearance of decoded images (particularly at higher compression ratios) and (2) the filtered macroblock is used for motion-compensated prediction of further frames in the encoder, resulting in a smaller residual after prediction. (Note: intra-coded macroblocks are filtered, but intra prediction is carried out using **unfiltered** reconstructed macroblocks to form the prediction). Picture edges are not filtered.

Filtering is applied to vertical or horizontal edges of 4x4 blocks in a macroblock, in the following order:

1. Filter 4 vertical boundaries of the luma component (in order a,b,c,d in Figure 1)
2. Filter 4 horizontal boundaries of the luma component (in order e,f,g,h, Figure 1)
3. Filter 2 vertical boundaries of each chroma component (i,j)
4. Filter 2 horizontal boundaries of each chroma component (k,l)

Each filtering operation affects up to **three** pixels on either side of the boundary. Figure 2 shows 4 pixels on either side of a vertical or horizontal boundary in adjacent blocks p and q (p0,p1,p2,p3 and q0,q1,q2,q3). Depending on the current quantizer, the coding modes of neighbouring blocks and the gradient of image samples across the boundary, several outcomes are possible, ranging from (a) no pixels are filtered to (b) p0, p1, p2, q0, q1, q2 are filtered to produce output pixels P0, P1, P2, Q0, Q1 and Q2.

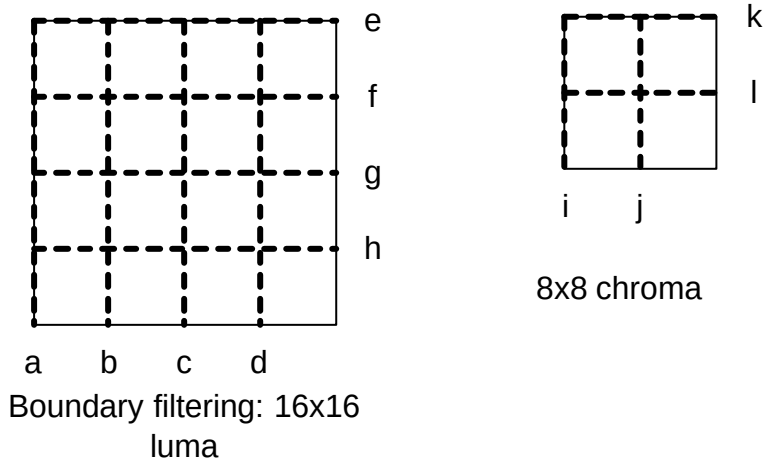


Figure 1 Edge filtering order in a macroblock

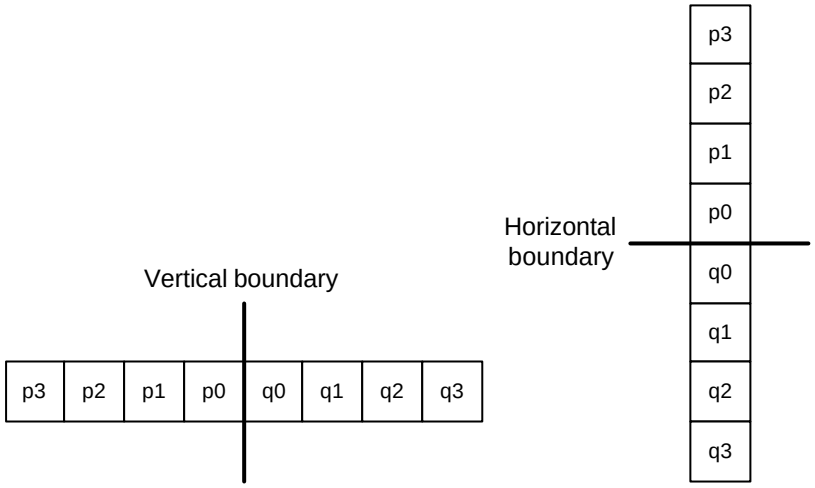


Figure 2 Pixels adjacent to vertical and horizontal boundaries

3. Boundary strength

The choice of filtering outcome depends on the **boundary strength** and on the **gradient** of image samples across the boundary. The boundary strength parameter **Bs** is chosen according to the following rules:

p or q is intra coded **and** boundary is a macroblock boundary	Bs=4 (strongest filtering)
p or q is intra coded and boundary is **not** a macroblock boundary	Bs=3
neither p or q is intra coded; p or q contain coded coefficients	Bs=2
neither p or q is intra coded; neither p or q contain coded coefficients; p and q have different reference frames **or** a different number of reference frames **or** different motion vector values	Bs=1
neither p or q is intra coded; neither p or q contain coded coefficients; p and q have same reference frame and identical motion vectors	Bs=0 (no filtering)

The filter is "stronger" at places where there is likely to be significant blocking distortion, such as the boundary of an intra coded macroblock or a boundary between blocks that contain coded coefficients.

4. Filter decision

A group of samples from the set (p2,p1,p0,q0,q1,q2) is filtered only if:
(a) Bs > 0 and
(b) |p0-q0|, |p1-p0| and |q1-q0| are each **less** than a threshold ? or ? (? and ? are defined in the standard [1]).

The thresholds ? and ? increase with the average quantizer parameter QP of the two blocks p and q. The purpose of the filter decision is to "switch off" the filter when there is a significant change (gradient) across the block boundary in the original image. The definition of a significant change depends on QP. When QP is small, anything other than a very small gradient across the boundary is likely to be due to image features (rather than blocking effects) that should be preserved and so the thresholds ? and ? are low. When QP is larger, blocking distortion is likely to be more significant and ? , ? are higher so that more filtering takes place.

5. Filter implementation

(a) Bs ? {1,2,3}:
A 4-tap linear filter is applied with inputs p1, p0, q0 and q1, producing filtered outputs P0 and Q0 (0<Bs<4).

In addition, if |p2-p0| is less than threshold ?, a 4-tap linear filter is applied with inputs p2, p1, p0 and q0, producing filtered output P1. If |q2-q0| is less than threshold ?, a 4-tap linear filter is applied with inputs q2, q1, q0 and p0, producing filtered output Q1. (p1 and q1 are never filtered for chroma, only for luma data).

(b) Bs = 4:
If |p2-p0|<? and |p0-q0|<round(?/4):
P0 is produced by 5-tap filtering of p2, p1, p0, q0 and q1
P1 is produced by 4-tap filtering of p2, p1, p0 and q0
(Luma only) P2 is produced by 5-tap filtering of p3, p2, p1, p0 and q0.
else:
P0 is produced by 3-tap filtering of p1, p0 and q1.

If |q2-q0|<? and |p0-q0|<round(?/4):
Q0 is produced by 5-tap filtering of q2, q1, q0, p0 and p1
Q1 is produced by 4-tap filtering of q2, q1, q0 and p0
(Luma only) Q2 is produced by 5-tap filtering of q3, q2, q1, q0 and p0.
else:
Q0 is produced by 3-tap filtering of q1, q0 and p1.

6. Filtering example

A QCIF video clip is encoded using the AVC reference software with a fixed Quantization Parameter of 32. Figure 3 shows an original frame from the clip; Figure 4 shows the same frame after inter coding and reconstruction, with the loop filter disabled. With the loop filter enabled (Figure 5) the appearance is considerably better; there is still some distortion but most of the block edges have disappeared or faded. Note that sharp contrast boundaries tend to be preserved by the filter whilst block edges in smoother regions of the picture are smoothed.

Figure 6 shows a decoded frame with a higher QP of 36 and with the loop filter disabled (note the increased blocking artefacts) and Figure 7 shows the same frame with the loop filter enabled.

Figure 3 Original frame

Figure 4 Reconstructed, QP=32 (no filter)

Figure 5 Reconstructed, QP=32 (with filter)

Figure 6 Reconstructed, QP=36 (no filter)

Figure 7 Reconstructed, QP=36 (with filter)

7. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document JVT-G050, March 2003

H.264 / MPEG-4 Part 10 White Paper

Variable-Length Coding

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". The standard specifies two types of entropy coding: Context-based Adaptive Binary Arithmetic Coding (CABAC) and Variable-Length Coding (VLC). The Variable-Length Coding scheme, part of the Baseline Profile of H.264, is described in this document.

Please note that the H.264 draft standard is not yet finalised and so readers are encouraged to refer to the latest version of the standard.

2. Coded elements

Parameters that require to be encoded and transmitted include the following (Table 2-1).

Table 2-1 Parameters to be encoded

Parameters	Description
Sequence-, picture- and slice-layer syntax elements	
Macroblock type mb_type	Prediction method for each coded macroblock
Coded block pattern	Indicates which blocks within a macroblock contain coded coefficients
Quantizer parameter	Transmitted as a delta value from the previous value of QP
Reference frame index	Identify reference frame(s) for inter prediction
Motion vector	Transmitted as a difference (mvd) from predicted motion vector
Residual data	Coefficient data for each 4x4 or 2x2 block

Above the slice layer, syntax elements are encoded as fixed- or variable-length binary codes. At the slice layer and below, elements are coded using either variable-length codes (VLCs) [2] or context-adaptive arithmetic coding (CABAC) [3] depending on the entropy encoding mode.

3. Variable length coding (VLC)

When entropy_coding_mode is set to 0, residual block data is coded using a context-adaptive variable length coding (CAVLC) scheme and other variable-length coded units are coded using Exp-Golomb codes.

3.1 Exp-Golomb entropy coding

Exp-Golomb codes (Exponential Golomb codes) are variable length codes with a regular construction. Table 3-1 lists the first 9 codewords; it is clear from this table that the codewords progress in a logical order. Each codeword is constructed as follows:

[M zeros][1][INFO]

where INFO is an M-bit field carrying information. The first codeword has no leading zero or trailing INFO; codewords 1 and 2 have a single-bit INFO field; codewords 3-6 have a 2-bit INFO field; and so on. The length of each codeword is (2M+1) bits.

Each Exp-Golomb codeword can be constructed by the encoder based on its index **code_num**:

$M = ?\log_2(code\_num+1)?$
$INFO = code\_num + 1 - 2^M$

A codeword can be decoded as follows:
1. Read in M leading zeros followed by 1.
2. Read M-bit INFO field.
3. code_num = 2^M + INFO – 1

(For codeword 0, INFO and M are zero).

Table 3-1 Exp-Golomb codewords

code_num	Codeword
0	1
1	010
2	011
3	00100
4	00101
5	00110
6	00111
7	0001000
8	0001001
…	…

A parameter **v** to be encoded is mapped to code_num in one of 3 ways:

ue(v) : Unsigned direct mapping, code_num = v. Used for macroblock type, reference frame index and others.

se(v) : Signed mapping, used for motion vector difference, delta QP and others. v is mapped to code_num as follows (Table 3-2).
code_num = 2|v| (v < 0)
code_num = 2|v| - 1 (v ? 0)

Table 3-2 Signed mapping se(v)

v	code_num
0	0
1	1
-1	2
2	3
-2	4
3	5
…	…

me(v) : Mapped symbols; parameter v is mapped to code_num according to a table specified in the standard. This mapping is used for the coded_block_pattern parameter. Table 3-3 lists a small part of the table for Inter predicted macroblocks: coded_block_pattern indicates which 8x8 blocks in a macroblock contain non-zero coefficients.

Table 3-3 Part of coded_block_pattern table

coded_block_pattern (Inter prediction)	code_num
0 (no non-zero blocks)	0
16 (chroma DC block non-zero)	1
1 (top-left 8x8 luma block non-zero)	2
2 (top-right 8x8 luma block non-zero)	3

4 (lower-left 8x8 luma block non-zero)	4
8 (lower-right 8x8 luma block non-zero)	5
32 (chroma DC and AC blocks non-zero)	6
3 (top-left and top-right 8x8 luma blocks non-zero)	7
…	…

Each of these mappings (ue, se and me) is designed to produce short codewords for frequently-occurring values and longer codewords for less common parameter values. For example, macroblock type Pred_L0_16x16 (i.e. 16x16 prediction from a previous picture) is assigned code_num 0 because it occurs frequently whereas macroblock type Pred_8x8 (8x8 prediction from a previous picture) is assigned code_num 3 because it occurs less frequently. The commonly-occurring motion vector difference (MVD) value of 0 maps to code_num 0 whereas the less-common MVD = -3 maps to code_num 6.

3.2 Context-based adaptive variable length coding (CAVLC)

This is the method used to encode residual, zig-zag ordered 4x4 (and 2x2) blocks of transform coefficients. CAVLC is designed to take advantage of several characteristics of quantized 4x4 blocks:

1. After prediction, transformation and quantization, blocks are typically sparse (containing mostly zeros). CAVLC uses run-level coding to compactly represent strings of zeros.
2. The highest non-zero coefficients after the zig-zag scan are often sequences of +/-1. CAVLC signals the number of high-frequency +/-1 coefficients ("Trailing 1s" or "T1s") in a compact way.
3. The number of non-zero coefficients in neighbouring blocks is correlated. The number of coefficients is encoded using a look-up table; the choice of look-up table depends on the number of non-zero coefficients in neighbouring blocks.
4. The level (magnitude) of non-zero coefficients tends to be higher at the start of the reordered array (near the DC coefficient) and lower towards the higher frequencies. CAVLC takes advantage of this by adapting the choice of VLC look-up table for the "level" parameter depending on recently-coded level magnitudes.

CAVLC encoding of a block of transform coefficients proceeds as follows.

1. Encode the number of coefficients and trailing ones (coeff_token).
The first VLC, coeff_token, encodes both the total number of non-zero coefficients (TotalCoeffs) and the number of trailing +/-1 values (T1). TotalCoeffs can be anything from 0 (no coefficients in the 4x4 block) [1] to 16 (16 non-zero coefficients). T1 can be anything from 0 to 3; if there are more than 3 trailing +/-1s, only the last 3 are treated as "special cases" and any others are coded as normal coefficients.

There are 4 choices of look-up table to use for encoding coeff_token, described as Num-VLC0, Num-VLC1, Num-VLC2 and Num-FLC (3 variable-length code tables and a fixed-length code). The choice of table depends on the number of non-zero coefficients in upper and left-hand previously coded blocks N_u and N_L. A parameter N is calculated as follows:

If blocks U and L are available (i.e. in the same coded slice), $N = (N_u + N_L)/2$
If only block U is available, $N=N_U$; if only block L is available, $N=N_L$; if neither is available, N=0.

Figure 3-1 Neighbouring blocks N_U and N_L

[1] Note: coded_block_pattern (described earlier) indicates which **8x8** blocks in the macroblock contain non-zero coefficients; however, within a coded 8x8 block, there may be **4x4** sub-blocks that do not contain any coefficients, hence TotalCoeff may be 0 in any 4x4 sub-block. In fact, this value of TotalCoeff occurs most often and is assigned the shortest VLC.

N selects the look-up table (Table 3-4) and in this way the choice of VLC adapts depending on the number of coded coefficients in neighbouring blocks (**context adaptive**). Num-VLC0 is "biased" towards small numbers of coefficients; low values of TotalCoeffs (0 and 1) are assigned particularly short codes and high values of TotalCoeff particularly long codes. Num-VLC1 is biased towards medium numbers of coefficients (TotalCoeff values around 2-4 are assigned relatively short codes), Num-VLC2 is biased towards higher numbers of coefficients and FLC assigns a fixed 6-bit code to every value of TotalCoeff.

Table 3-4 Choice of look-up table for coeff_token

N	Table for coeff_token
0, 1	Num-VLC0
2, 3	Num-VLC1
4, 5, 6, 7	Num-VLC2
8 or above	FLC

2. Encode the sign of each T1.
For each T1 (trailing +/-1) signalled by coeff_token, a single bit encodes the sign (0=+, 1=-). These are encoded **in reverse order**, starting with the highest-frequency T1.

3. Encode the levels of the remaining non-zero coefficients.
The **level** (sign and magnitude) of each remaining non-zero coefficient in the block is encoded **in reverse order**, starting with the highest frequency and working back towards the DC coefficient. The choice of VLC table to encode each level adapts depending on the magnitude of each successive coded level (**context adaptive**). There are 7 VLC tables to choose from, Level_VLC0 to Level_VLC6. Level_VLC0 is biased towards lower magnitudes; Level_VLC1 is biased towards slightly higher magnitudes and so on. The choice of table is adapted in the following way:

(a) Initialise the table to Level_VLC0 (unless there are more than 10 non-zero coefficients and less than 3 trailing ones, in which case start with Level_VLC1).
(b) Encode the highest-frequency non zero coefficient.
(c) If the magnitude of this coefficient is larger than a pre-defined threshold, move up to the next VLC table.

In this way, the choice of level is matched to the magnitude of the recently-encoded coefficients. The thresholds are listed in Table 3-5; the first threshold is zero which means that the table is always incremented after the first coefficient level has been encoded.

Table 3-5 Thresholds for determining whether to increment Level table number

Current VLC table	Threshold to increment table
VLC0	0
VLC1	3
VLC2	6
VLC3	12
VLC4	24
VLC5	48
VLC6	N/A (highest table)

4. Encode the total number of zeros before the last coefficient.
TotalZeros is the sum of all zeros preceding the highest non-zero coefficient in the reordered array. This is coded with a VLC. The reason for sending a separate VLC to indicate TotalZeros is that many blocks contain a number of non-zero coefficients at the start of the array and (as will be seen later) this approach means that zero-runs at the start of the array need not be encoded.

5. Encode each run of zeros.
The number of zeros preceding each non-zero coefficient (run_before) is encoded **in reverse order**. A run_before parameter is encoded for each non-zero coefficient, starting with the highest frequency, with two exceptions:
(a) If there are no more zeros left to encode (i.e. ? [run_before] = TotalZeros), it is not necessary to encode any more run_before values.
(b) It is not necessary to encode run_before for the final (lowest frequency) non-zero coefficient.

The VLC for each run of zeros is chosen depending on (a) the number of zeros that have not yet been encoded (ZerosLeft) and (b) run_before. For example, if there are only 2 zeros left to encode, run_before can only take 3 values (0,1 or 2) and so the VLC need not be more than 2 bits long; if there are 6 zeros still to encode then run_before can take 7 values (0 to 6) and the VLC table needs to be correspondingly larger.

CAVLC Examples

In all the following examples, we assume that table Num-VLC0 is used to encode coeff_token.

Example 1

4x4 block:

0	3	-1	0
0	-1	1	0
1	0	0	0
0	0	0	0

Reordered block:
0,3,0,1,-1,-1,0,1,0…

TotalCoeffs = 5 (indexed from highest frequency [4] to lowest frequency [0])
TotalZeros = 3
T1s = 3 (in fact there are 4 trailing ones but only 3 can be encoded as a “special case”)

Encoding:

Element	Value	Code
coeff_token	TotalCoeffs=5, T1s=3	0000100
T1 sign (4)	+	0
T1 sign (3)	-	1
T1 sign (2)	-	1
Level (1)	+1 (use Level_VLC0)	1
Level (0)	+3 (use Level_VLC1)	0010
TotalZeros	3	111
run_before(4)	ZerosLeft=3; run_before=1	10
run_before(3)	ZerosLeft=2; run_before=0	1
run_before(2)	ZerosLeft=2; run_before=0	1
run_before(1)	ZerosLeft=2; run_before=1	01
run_before(0)	ZerosLeft=1; run_before=1	No code required; last coefficient.

The transmitted bitstream for this block is 000010001110010111101101 .

Decoding:

The output array is "built up" from the decoded values as shown below. Values added to the output array at each stage are underlined.

Code	Element	Value	Output array
0000100	coeff_token	TotalCoeffs=5, T1s=3	Empty
0	T1 sign	+	1
1	T1 sign	-	-1, 1
1	T1 sign	-	-1, -1, 1
1	Level	+1	1, -1, -1, 1
0010	Level	+3	3, 1, -1, -1, 1
111	TotalZeros	3	3, 1, -1, -1, 1
10	run_before	1	3, 1, -1, -1, 0, 1
1	run_before	0	3, 1, -1, -1, 0, 1
1	run_before	0	3, 1, -1, -1, 0, 1
01	run_before	1	3, 0, 1, -1, -1, 0, 1

The decoder has inserted two zeros; however, TotalZeros is equal to 3 and so another 1 zero is inserted before the lowest coefficient, making the final output array:
0, 3, 0, 1, -1, -1, 0, 1

Example 2

4x4 block:

-2	4	0	-1
3	0	0	0
-3	0	0	0
0	0	0	0

Reordered block:
-2, 4, 3, -3, 0, 0, -1, …

TotalCoeffs = 5 (indexed from highest frequency [4] to lowest frequency [0])
TotalZeros = 2
T1s = 1

Encoding:

Element	Value	Code
coeff_token	TotalCoeffs=5, T1s=1	0000000110
T1 sign (4)	-	1
Level (3)	Sent as –2 (**see note 1**) (use Level_VLC0)	0001
Level (2)	3 (use Level_VLC1)	0010
Level (1)	4 (use Level_VLC1)	00010
Level (0)	-2 (use Level_VLC2)	111
TotalZeros	2	0011
run_before(4)	ZerosLeft=2; run_before=2	00
run_before(3..0)	0	No code required

The transmitted bitstream for this block is 000000011010001001000010111001100.

Note 1: Level (3), with a value of -3, is encoded as a special case. If there are less than 3 T1s, then the first **non**-T1 level will **not** have a value of +/-1 (otherwise it would have been encoded as a T1). To

save bits, this level is incremented if negative (decremented if positive) so that +/-2 maps to +/-1, +/-3 maps to +/-2, and so on. In this way, shorter VLCs are used.

Note 2: After encoding level (3), the level_VLC table is incremented because the magnitude of this level is greater than the first threshold (which is 0). After encoding level (1), with a magnitude of 4, the table number is incremented again because level (1) is greater than the second threshold (which is 3). Note that the final level (-2) uses a different code from the first encoded level (also –2).

Decoding:

Code	Element	Value	Output array
0000000110	coeff_token	TotalCoeffs=5, T1s=1	Empty
1	T1 sign	-	-1
0001	Level	-2 decoded as -3	-3, -1
0010	Level	+3	+3, -3, -1
00010	Level	+4	+4, 3, -3, -1
111	Level	-2	-2, 4, 3, -3, -1
0011	TotalZeros	2	-2, 4, 3, -3, -1
00	run_before	2	-2, 4, 3, -3, 0, 0, -1

All zeros have now been decoded and so the output array is:
-2, 4, 3, -3, 0, 0, -1

(This example illustrates how bits are saved by encoding TotalZeros: only a single run needs to be coded even though there are 5 non-zero coefficients).

4. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, “Advanced Video Coding”, Final Committee Draft, Document JVT-E022, September 2002
2 JVT Document JVT-C028, G.Bjontegaard and K. Lillevold, “Context-Adaptive VLC Coding of Coefficients”, Fairfax, VA, May 2002
3 JVT Document JVT-L13, D. Marpe, G. Blattermann and T. Wiegand, “Adaptive Codes for H.26L”, Eibsee, January 2001

H.264 / MPEG-4 Part 10 White Paper

Context-Based Adaptive Arithmetic Coding (CABAC)

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". The standard specifies two types of entropy coding: Context-based Adaptive Binary Arithmetic Coding (CABAC) and Variable-Length Coding (VLC). This document provides a short introduction to CABAC. Familiarity with the concept of Arithmetic Coding is assumed (see chapter 8 of [2] for an introduction to Arithmetic Coding).

Please note that the H.264 draft standard is not yet finalised and so readers are encouraged to refer to the latest version of the standard.

2. Context-based adaptive binary arithmetic coding (CABAC)

When entropy_coding_mode is set to 1, an arithmetic coding system is used to encode and decode H.264 syntax elements. The arithmetic coding scheme selected for H.264, Context-based Adaptive Binary Arithmetic Coding or CABAC [3], achieves good compression performance through (a) selecting probability models for each syntax element according to the element's context, (b) adapting probability estimates based on local statistics and (c) using arithmetic coding.

Coding a data symbol involves the following stages.

1. Binarization: CABAC uses Binary Arithmetic Coding which means that only binary decisions (1 or 0) are encoded. A non-binary-valued symbol (e.g. a transform coefficient or motion vector) is "binarized" or converted into a binary code prior to arithmetic coding. This process is similar to the process of converting a data symbol into a variable length code but the binary code is further encoded (by the arithmetic coder) prior to transmission.

Stages 2, 3 and 4 are repeated for each bit (or "bin") of the binarized symbol.

2. Context model selection: A "context model" is a probability model for one or more bins of the binarized symbol. This model may be chosen from a selection of available models depending on the statistics of recently-coded data symbols. The context model stores the probability of each bin being "1" or "0".

3. Arithmetic encoding: An arithmetic coder encodes each bin according to the selected probability model. Note that there are just two sub-ranges for each bin (corresponding to "0" and "1").

4. Probability update: The selected context model is updated based on the actual coded value (e.g. if the bin value was "1", the frequency count of "1"s is increased).

3. The coding process

We will illustrate the coding process for one example, MVD_x (motion vector difference in the x-direction).

1. Binarize the value MVD_x . Binarization is carried out according to the following table for $|MVD_x|<9$ (larger values of MVD_x are binarized using an Exp-Golomb codeword).

$\|MVD_x\|$	Binarization
0	0

1	10
2	110
3	1110
4	11110
5	111110
6	1111110
7	11111110
8	111111110

(Note that each of these binarized codewords are uniquely decodeable).

The first bit of the binarized codeword is bin 1; the second bit is bin 2; and so on.

2. Choose a context model for each bin. One of 3 models is selected for bin 1, based on previous coded MVD values. The L1 norm of two previously-coded values, e_k, is calculated:

$e_k = |MVD_A| + |MVD_B|$ where A and B are the blocks immediately to the left and above the current block (respectively).

e_k	Context model for bin 1
0 ? e_k < 3	Model 0
3 ? e_k < 33	Model 1
33 ? e_k	Model 2

If e_k is small, then there is a high probability that the current MVD will have a small magnitude; conversely, if e_k is large then it is more likely that the current MVD will have a large magnitude. We select a probability table (context model) accordingly.

The remaining bins are coded using one of 4 further context models:

Bin	Context model
1	0, 1 or 2 depending on e_k
2	3
3	4
4	5
5	6
6 and higher	6

3. Encode each bin. The selected context model supplies two probability estimates: the probability that the bin contains “1” and the probability that the bin contains “0”. These estimates determine the two sub-ranges that the arithmetic coder uses to encode the bin.

4. Update the context models. For example, if context model 2 was selected for bin 1 and the value of bin 1 was “0”, the frequency count of “0”s is incremented. This means that the next time this model is selected, the probability of an “0” will be slightly higher. When the total number of occurrences of a model exceeds a threshold value, the frequency counts for “0” and “1” will be scaled down, which in effect gives higher priority to recent observations.

4. The context models

Context models and binarization schemes for each syntax element are defined in the standard. There are a total of 267 separate context models, 0 to 266 (as of September 2002) for the various syntax elements. Some models have different uses depending on the slice type: for example, skipped

macroblocks are not permitted in an I-slice and so context models 0-2 are used to code bins of mb_skip or mb_type depending on whether the current slice is Intra coded.

At the beginning of each coded slice, the context models are initialised depending on the initial value of the Quantization Parameter QP (since this has a significant effect on the probability of occurrence of the various data symbols).

5. The arithmetic coding engine

The arithmetic decoder is described in some detail in the Standard. It has three distinct properties:
1. Probability estimation is performed by a transition process between 64 separate probability states for "Least Probable Symbol" (LPS, the least probable of the two binary decisions "0" or "1").
2. The range R representing the current state of the arithmetic coder is quantized to a small range of pre-set values before calculating the new range at each step, making it possible to calculate the new range using a look-up table (i.e. multiplication-free).
3. A simplified encoding and decoding process is defined for data symbols with a near-uniform probability distribution.

The definition of the decoding process is designed to facilitate low-complexity implementations of arithmetic encoding and decoding. Overall, CABAC provides improved coding efficiency compared with VLC at the expense of greater computational complexity.

6. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document JVT-E022, September 2002
2 I. Richardson, "Video CODEC Design", John Wiley & Sons, 2002.
3 D. Marpe, G Blättermann and T Wiegand, "Adaptive Codes for H.26L", ITU-T SG16/6 document VCEG-L13, Eibsee, Germany, January 2001

H.264 / AVC

Frame and picture management Draft 1.0 © Iain G Richardson Jan 04

iainrichardson@ieee.org

1 Introduction

This document introduces the parameters and processes involved in managing coded frames within the H.264/AVC standard. This document is informative only and readers should refer to the standard for accurate definitions of the parameters and processes described here.

A frame or field of video is decoded from an access unit (a series of NAL units including one or more coded slices making up a coded picture). The decoding order of access units is indicated by the parameter frame_num (section 2) and the display order is indicated by the parameter Picture Order Count (section 3). Decoded pictures may be marked as "used for reference" (section 4.1) in which case they are available for inter prediction of further decoded pictures. Reference pictures are organised into one or two lists for inter prediction of P, B or SP slices. The default order of these lists (section 4.2) may be explicitly modified by a reference picture list reordering process (section 4.3).

2 frame_num

The parameter frame_num is decoded from each slice header. frame_num increases in **decoding** order of access units and does not necessarily indicate **display** order.

IDR : frame_num set to zero.

Otherwise: increments by 1 from previous reference frame (in decoding order) (unless gaps_in_frame_num_value_allowed, in which case decoder has to create "dummy" decoded frames to fill gap; or unless the current picture and the preceding reference picture are fields with opposite parity).

3 Picture Order Count (POC)

3.1 Overview

POC determines the **display** (output) order of decoded frames, starting from first field of an IDR picture (POC=0).

POC is derived from the slice header in one of 3 ways (see below). POC derived as TopFieldOrderCount and BottomFieldOrderCount, for the top and bottom fields of each coded frame.

Note 1: an application may assign POC proportional to the sampling time of a picture relative to the last IDR. This could lead to variable gaps in POC.

Note 2: the JM reference encoder increments POC by 2 for every complete frame.

3.2 POC Updating

There are 3 supported methods of updating POC:

3.2.1 Type 0: send POC explicitly in each slice header

(Allows maximum flexibility)

TopFieldOrderCount = POCMsb + POCLsb
POCLsb is sent in each slice header
POCMsb is incremented when POCLsb reaches its maximum value

Picture_order_present: delta_POC_bottom is present in slice_header, can change the delta POC between top (first) and bottom (2^{nd}) fields (default is zero delta)

Example (generated from JM reference software)

Frame pictures
Display order: IBPBPBPB… POC type 0 B not used for reference

In order of access units:

Access unit	Type	Used for reference	frame_num	POC_lsb	TopFOC	Display order
1^{st}	I	Yes	0	0	0	0
2^{nd}	P	Yes	1	4	4	2
3^{rd}	B	No	2	2	2	1
4^{th}	P	Yes	2	8	8	4
5^{th}	B	No	3	6	6	3
6^{th}	P	Yes	3	12	12	6
7^{th}	B	No	4	10	10	5
8^{th}	P	Yes	4	16	16	8
…	…					

(Frame number increments relative to previous **reference** picture)

3.2.2 Type 1: set up expected increments in sequence parameter set; only send a delta if there is any change to expected order

(Suitable for situation where there is a repeating "cycle" of pictures, c.f. MPEG-2 GOP)

Sequence parameter set defines number of ref frames in POC "cycle" (repeating group of ref + non-ref frames); offset to each ref frame in the "cycle"; offset to non-reference frame

For each picture, calculate an expected POC as follows:

- ? calculate number of POC cycles (since last IDR picture)
- ? calculate position of current frame in POC cycle
- ? calculate expected POC for current reference frame
- ? add offset_for_non_ref_pic if this is not a reference frame

TopFieldOrderCount = expected POC + delta_pic_order_cnt[0]
BottomFieldOrderCount = expected POC + delta[1] (if field_pic)
= expected POC + offset to bottom field + delta[0] (otherwise)

Example (a)

Display order: IBPBPBPB… POC type 1 B not used for reference
1 ref frame in POC cycle; offset to next ref frame = 4; offset for non-ref pic = -2

In order of access units:

Access unit	Type	Used for reference	frame_num	delta_pic_order_cnt[0]	TopFOC	Display order
1st	I	Yes	0	0	0	0
2nd	P	Yes	1	0	4	2
3rd	B	No	2	0	2	1
4th	P	Yes	2	0	8	4
5th	B	No	3	0	6	3
6th	P	Yes	3	0	12	6
7th	B	No	4	0	10	5
8th	P	Yes	4	0	16	8
…	…					

Example (b)

Display order: IBBPBBPBBPB… POC type 1 B not used for reference
1 ref frame in POC cycle; offset to next ref frame = 6; offset for non-ref pic = -4

In order of access units:

Access unit	Type	Used for reference	frame_num	delta_pic_order_cnt[0]	TopFOC	Display order
1st	I	Yes	0	0	0	0
2nd	P	Yes	1	0	6	3
3rd	B	No	2	0	2	1
4th	B	No	2	2	4	2
5th	P	Yes	2	0	12	6
6th	B	No	3	0	8	1
7th	B	No	3	2	10	2
8th	P	Yes	3	0	18	9
…	…					

[Note that 4th and 7th access units have a delta POC of +2; the POC (TopFOC) of each is (expected ref frame count)-4+2]

3.2.3 Type 2: display order same as decoding order

(POC is derived from frame_num; minimal overhead)

For each picture:

```
if (used for reference)
        set TopFOC and/or BottomFOC to (2*frame_num)
else
        set TopFOC and/or BottomFOC to (2*frame_num)-1
```

This effectively means that (1) only one non-reference picture can occur between reference pictures, (2) the display order is the same as the access unit order (decoding order), (3) if all pictures are used for reference, POC increments by 2 each time.

4 Reference lists

4.1 Reference picture marking

Picture that is encoded or decoded and available for reference is stored in the Decoded Picture Buffer (DPB) and marked as (a) a short term reference picture, indexed according to frame_num or PicOrderCount or (b) a long term reference picture, indexed according to LongTermPicNum, a reference index assigned when a picture is marked as a long term reference picture. Short term reference pictures may be assigned a LongTermPicNum ("changed" to a long term reference picture) at a later time.

Short term reference pictures are removed from the DPB (a) by an explicit command in the bitstream or (b) when the DPB is "full" (oldest short term picture is removed). Long term pictures are removed by an explicit command in the bitstream.

4.2 Reference picture ordering

Reference pictures are ordered in one or two lists prior to encoding or decoding a slice.

P slices use a single list of reference pictures, list0; B slices use 2 lists, list0 and list1. In each list, short term reference pictures are listed first by default (see below) followed by long term reference pictures (in increasing order of LongTermPicNum). The default short term reference picture order depends on **decoding order** when the current slice is a P slice and depends on **display order** when the current slice is a B slice.

Default order of short term reference pictures in reference lists:

List0 (P slice) : decreasing order of PicNum. (PicNum is a "wrapped around" (mod MaxFrameNum) version of frame_num).

List0 (B slice) : (1) decreasing order of PicOrderCount (for pictures with POC earlier than current picture) then (2) increasing order of PicOrderCount (for pictures with POC later than current picture).

List1 (B slice): (1) increasing order of PicOrderCount (later than current picture) then (2) decreasing order of PicOrderCount (earlier than current picture)

Example:
P slice, list0. Reference picture list is initially empty. Current frame_num is 150. Maximum size of the DPB is 5 frames. Italics indicate a LongTermPicNum.

Operation	list0(0)	list0(1)	list0(2)	list0(3)	list0(4)
Initial state	-	-	-	-	-
Encode frame 150	150	-	-	-	-
Encode 151	151	150	-	-	-
Encode 152	152	151	150	-	-
Encode 153	153	152	151	150	-
Encode 154	154	153	152	151	150
Encode 155	155	154			
Assign 154 to	155	153	152	151	*3*

LongTermPicNum *3*					
Encode 156 and mark it as LongTermPicNum *1*	155	153	152	*1*	*3*
Encode 157	157	155	153	*1*	*3*
…..					

4.3 Reference picture list reordering

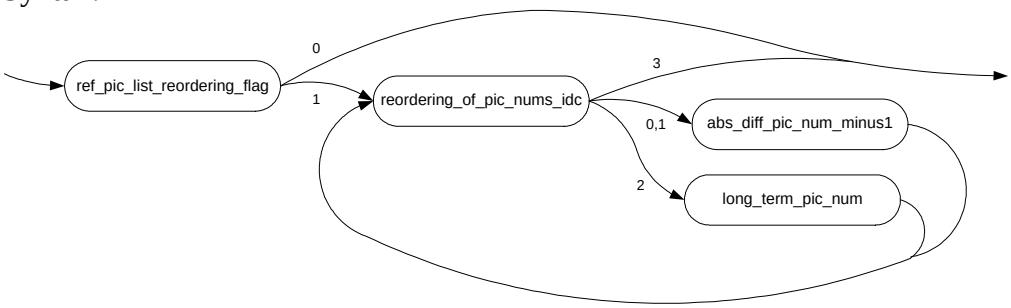
4.3.1 Overview

Purpose: enables encoder to change default order of reference pictures in list0 (and list1 for B-slices) **temporarily** for the next decoded slice.

Example application: The reference picture index ref_idx_l0 (or l1) occurs once in each MB or MB partition. This is signalled as a truncated Exp-Golomb code (te). Larger values of ref_idx cost more bits. There may be a reference picture (short term or long term) that is particularly useful for prediction of the current slice but is not in position 0 in the default list. This process enables the encoder to place this reference picture at a low index in the list so that it costs fewer bits to signal prediction from this picture.

4.3.2 Process overview

Syntax:



If ref_pic_list_reordering_flag is 1, the reordering process is repeatedly carried out until reordering_of_pic_nums_idc is 3.

Reordering process (list0; simplified) (similar for list1):

Initialise a pointer (refIdxL0) to point to the first reference picture index (0)
While reordering_of_pic_nums_idc ? 3
    Select a reference picture (short term, indicated by abs_diff_pic_num_minus1, or long term, indicated by long_term_pic_num)
    Move this picture to the position in the list indicated by refIdxL0
    Move all pictures from this position onwards one position later in list
    Increment pointer refIdxL0

4.3.3 Selecting a reference picture to move (remap) to current position

Short-term:
abs_diff_pic_num_minus1 signals an offset (positive or negative) from a **predicted** reference picture. For the first reordering instruction (remapping), the predicted picture is the current picture number. For subsequent reordering instructions, the predicted picture is the picture number of the most recently remapped picture.

If reordering_of_pic_nums_idc is 0, the picture to be remapped is calculated as follows:
remapped_picture = predicted_picture – abs_diff_pic_num_minus_1

If reordering_of_pic_nums_idc is 1, the remapped picture is calculated as follows:
remapped_picture = predicted_picture + abs_diff_pic_num_minus_1

(in each case, the calculation is modified to prevent errors due to the picture number wrapping round).

Long-term:
long_term_pic_num indicates a long term picture to be remapped to the current position in the list.

Example:

P slice, list0, DPB contains 5 reference pictures. Current frame_num is 158 [check: is this correct if latest frame in DPB is 157 ?].
Default reference list is as follows:

157, 155, 153, *1*, *3*

The following series of reference picture reordering commands are received:

Initial predicted_picture = 158; initial refIdxL0 = 0

1. reordering_of_pic_nums_idc = 0, abs_diff_pic_num_minus_1 = 5
remapped_picture = 158 – 5 = 153
New list: 153, 157, 155, *1*, *3*
New predicted picture = 153; new refIdxL0 = 1.

2. reordering_of_pic_nums_idc = 1, abs_diff_pic_num_minus_1 = 2
remapped_picture = 153 + 2 = 155
New list: 153, 155, 157, *1*, *3*
New predicted picture = 155; new refIdxL0 = 2.

3. reordering_of_pic_nums_idc = 2, long_term_pic_num = 3
remapped_picture = *3*
New list: 153, 155, *3*, 157, *1*

4. reordering_of_pic_nums_idc = 3
End of reordering process.

H.264 / MPEG-4 Part 10 White Paper

Switching P and I slices

1. Introduction

The Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG are finalising a new standard for the coding (compression) of natural video images. The new standard [1] will be known as H.264 and also MPEG-4 Part 10, "Advanced Video Coding". This document introduces the concepts of Switching P and I slices, part of the Extended Profile of H.264.

2. SP and SI slices

SP and SI slices are specially-coded slices that enable (among other things) efficient switching between video streams and efficient random access for video decoders. A common requirement is for a video decoder to switch between one of several encoded streams. For example, the same video material is coded at multiple bitrates for transmission across the Internet; a decoder attempts to decode the highest-bitrate stream it can receive but may require to switch automatically to a lower-bitrate stream if the data throughput drops.

Example: A decoder is decoding Stream A and wants to switch to decoding Stream B (Figure 2-1). For simplicity, assume that each frame is encoded as a single slice and predicted from one reference (the previous decoded frame). After decoding P-slices A_0 and A_1, the decoder wants to switch to Stream B and decode B_2, B_3 and so on. If all the slices in Stream B are coded as P-slices, then the decoder will not have the correct decoded reference frame(s) required to reconstruct B_2 (since B_2 is predicted from the decoded frame B_1 which does not exist in stream A). One solution is to code frame B_2 as an I-slice. Because it is coded without prediction from any other frame, it can be decoded independently of preceding frames in stream B. The decoder can therefore switch between stream A and stream B as shown in Figure 2-1. Switching can be accomplished by inserting an I-slice at regular intervals in the coded sequence to create "switching points". However, an I-slice is likely to contain much more coded data than a P-slice and the result is a peak in the coded bitrate at each switching point.

SP-slices are designed to support switching without the increased bitrate penalty of I-slices. Figure 2-2 shows how they may be used. At the switching point (frame 2 in each sequence), there are now 3 coded SP-slices. These are coded using motion compensated prediction (making them more efficient than I-slices). SP-slice A_2 can be decoded using reference frame A_1; SP-slice B_2 can be decoded using reference frame B_1. The key to the switching process is SP-slice AB_2 (known as a **switching SP-slice**): this is created in such a way that it can be decoded using motion-compensated reference frame A_1, to produce decoded frame B_2 (i.e. the decoder output frame B_2 is identical whether decoding B_1 followed by B_2 or A_1 followed by AB_2). An extra SP-slice is required at each switching point (and in fact another SP-slice, BA_2, would be required to switch in the other direction) but this is still more efficient than encoding frames A_2 and B_2 as I-slices. Table 2-1 lists the steps involved when a decoder switches from stream A to stream B.

Table 2-1 Switching from stream A to stream B using SP-slices

Input to decoder	MC reference	Output of decoder
P-slice A_0	[earlier frame]	Decoded frame A_0
P-slice A_1	Decoded frame A_0	Decoded frame A_1
SP-slice AB_2	Decoded frame A_1	Decoded frame B_2
P-slice B_3	Decoded frame B_2	Decoded frame B_3
….	….	….

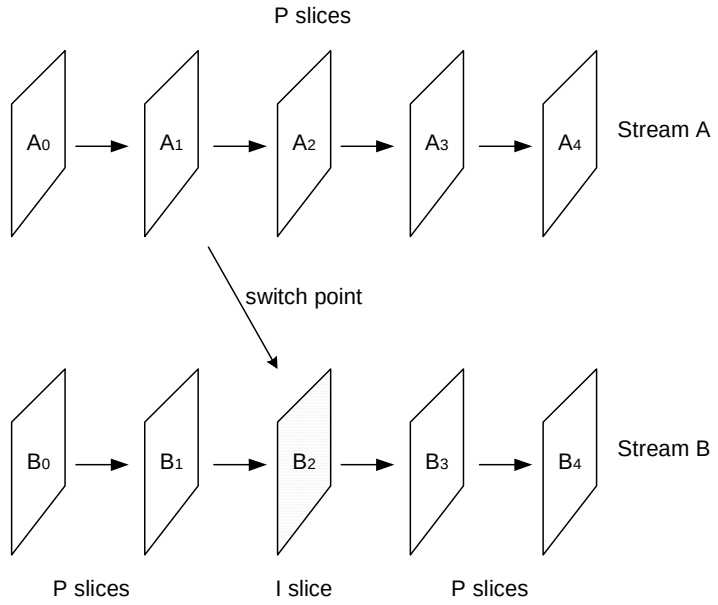


Figure 2-1 Switching streams using I-slices

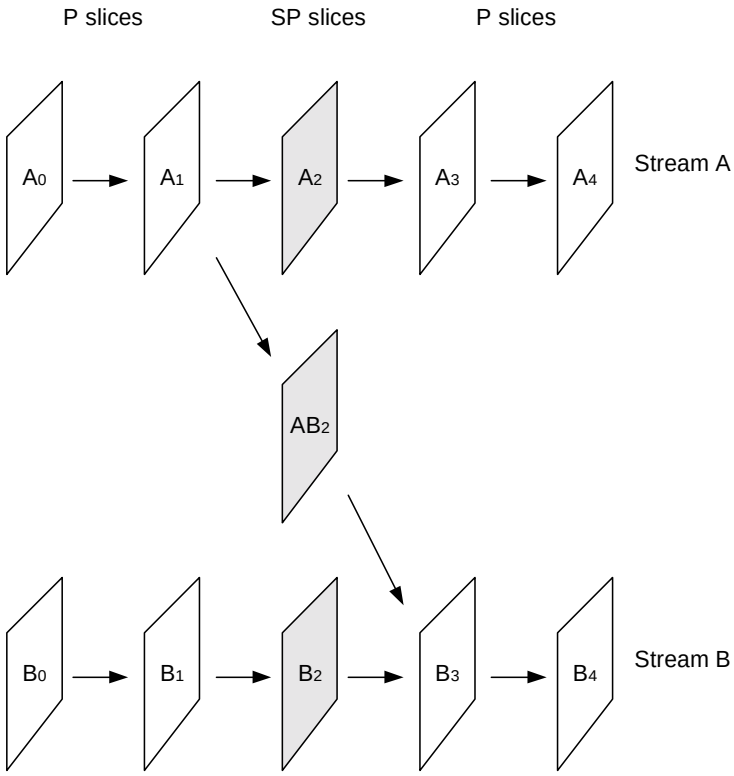


Figure 2-2 Switching streams using SP-slices

A simplified diagram of the encoding process for SP-slice A_2 is shown in Figure 2-3. A_2 is produced by subtracting a motion-compensated version of A_1' (decoded frame A_1) from frame A_2 and then coding the residual. Unlike a "normal" P-slice, the subtraction occurs in the transform domain (i.e. after the block transform). SP-slice B_2 is encoded in the same way (Figure 2-4). A decoder that has previously decoded frame A_1 can decode SP-slice A_2 as shown in Figure 2-5 (again, this is a

simplified diagram for clarity; in practice further quantization and rescaling steps are required to avoid mismatch).

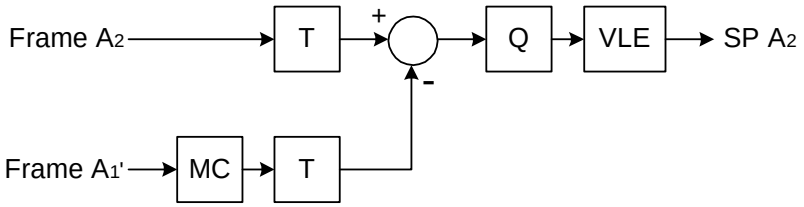


Figure 2-3 Encoding SP-slice A_2 (simplified)

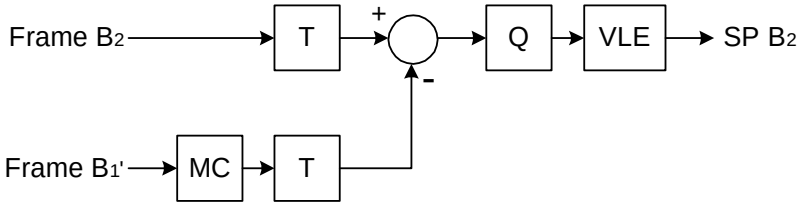


Figure 2-4 Encoding SP-slice B_2 (simplified)

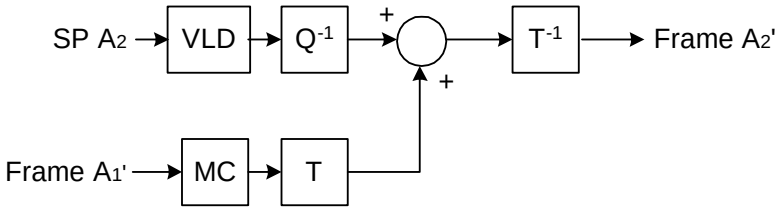


Figure 2-5 Decoding SP-slice A_2 (simplified)

SP-slice AB_2 is encoded as shown in Figure 2-6. Frame B_2 (the frame we are switching to) is transformed and quantized. A motion-compensated prediction is formed from A_1' (the frame we are switching from). It should be noted that the "MC" block in this diagram attempts to find the best match for each MB of frame B_2 **using decoded frame A_1 as a reference**. The motion-compensated prediction is transformed, quantized and subtracted from the transformed and quantized B_2 and the residual is encoded. A decoder that has previously decoded A_1' can decode SP-slice AB_2 to produce frame B_2' (Figure 2-7). Frame A_1' is motion compensated (using the motion vector data encoded as part of AB_2), transformed, quantized and added to the decoded residual, then the result is rescaled and inverse transformed to produce B_2'.

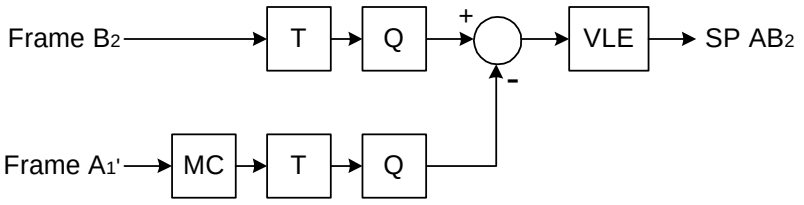


Figure 2-6 Encoding SP-slice AB_2 (simplified)

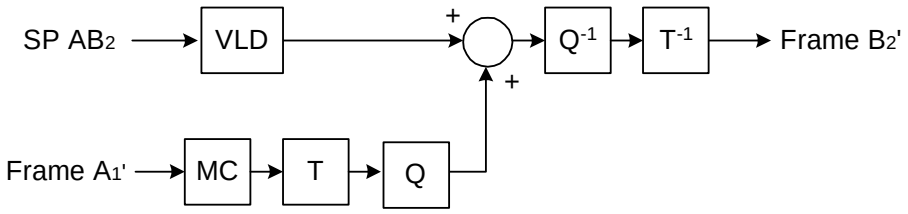


Figure 2-7 Decoding SP-slice AB_2

If streams A and B are versions of the same original sequence coded at different bitrates, then the motion-compensated prediction of B_2 from A_1' (SP-slice AB_2) should be quite efficient. Results show that using SP-slices to switch between different versions of the same sequence is significantly more efficient than inserting I-slices at switching points [2]. Another application of SP-slices is to provide random access and "VCR-like" functionalities. For example, an SP-slice and a switching SP-slice are placed at the position of frame 10 (Figure 2-8). A decoder can fast-forward from frame A_0 directly to frame A_{10} by first decoding A_0, then decoding switching SP-slice $A_{0\text{-}10}$ which can be decoded to produce frame A_{10} by prediction from A_0.

A second type of switching slice, the SI-slice, is also supported in the Extended Profile. This is used in a similar way to a switching SP-slice, except that the prediction is formed using 4x4 Intra Prediction from previously-decoded samples of the reconstructed frame. This slice mode may be used (for example) to switch from one sequence to a completely different sequence (in which case it will not be efficient to use motion compensated prediction because there is no correlation between the two sequences).

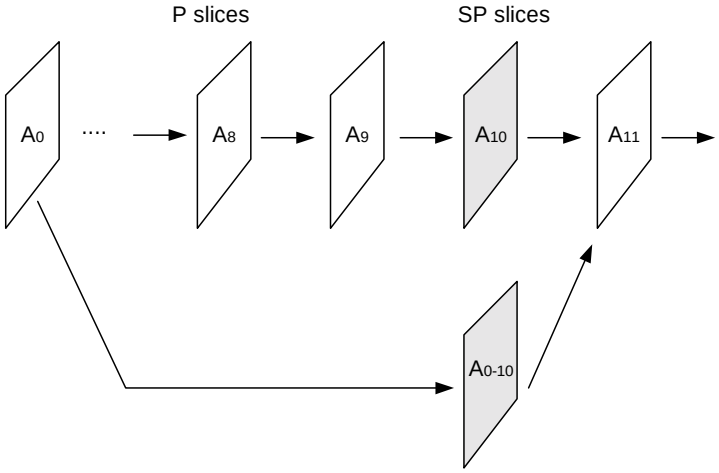


Figure 2-8 Fast-forward using SP-slices

3. References

1 ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document **JVT-F100**, December 2002

2 M. Karczewicz and R. Kurceren, "A Proposal for SP-Frames', ITU-T SG16/6 document VCEG-L27, Eibsee, Germany, January 2001