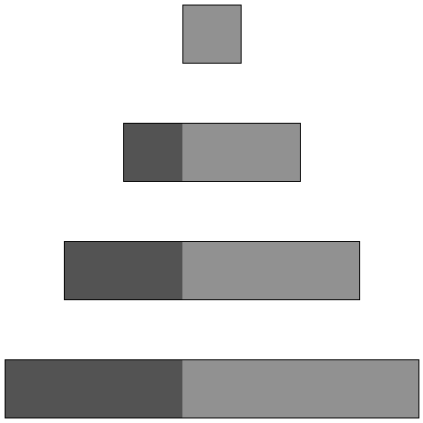
DIGITAL SUBBAND VIDEO

VERSION 1.0

REVISION 2

April, 2024



2023-2024 EMMIR
ENVEL GRAPHICS

Contact Information:

GitHub : https://github.com/LMP88959
YouTube: https://www.youtube.com/@EMMIR_KC/videos
Discord: https://discord.com/invite/hdYctSmyQJ

LICENSING

Specification License

Software License

Please note this document is not intended to be formal or comprehensive.

CONVENTIONS

Pseudo code is often written in the C or a pseudo-C language.
All data types are assumed to be 32-bit integers unless otherwise specified.

A prefix of 0x for a number means that the number is in hexadecimal format. For example, 0x55 is 85 in decimal.

DEFINITIONS

Block: a rectangular subregion of a plane of a frame.

Chroma/UV: the two planes of a frame that represent the color of the image at every point within it.

Frame: three two-dimensional grids of pixels representing the visual contents of a certain point in time.

Half-Pixel Filter: a filter used to generate a new image as though the original image had been shifted over by half a pixel.

Interleaved Exponential-Golomb Code: A form of variable length coding which can be more efficiently decoded due to its structure.

Inter Frame: a frame of video that relies on another frame to be correctly reconstructed.

Intra Frame: a frame of video that does not rely on another frame to be correctly reconstructed.

Luma/Y: the plane of a frame that represents the brightness of the image at every point within it.

Motion Compensation: the process of applying temporal prediction in order to reconstruct an inter frame.

Plane: a single component of the video, also known as a channel. Luma is its own plane, chroma consists of two planes: U, and V.

Variable Length Code: A way to represent an arbitrarily large integer in a variable number of bits.

TABLE OF CONTENTS

A. INTRODUCTION

A.1 Description

Digital Subband Video (DSV) is a digital video compression specification designed for efficient software decoding and playback on consumer hardware while offering compression performance comparable to Moving Picture Experts Group's MPEG-1/MPEG-2 video standards.

DSV is highly portable and does **not** require use of:

- data types with widths greater than 32 bits
- floating point data types or literals
- a specific operating system, processor
- any 3rd party libraries

A.2 Brief Technology Overview

DSV is a lossy hybrid video codec[1] which utilizes two different subband filters and block motion compensation to achieve spatial and temporal compression.

Overview:

- operates on YCC video (luma with two chroma components)
  - supports chroma subsampling of 4:4:4, 4:2:2, 4:2:0, and 4:1:1
  - only 8 bits of depth per component supported
  - no explicit support for interlaced video
- frame sizes must be divisible by two
- subband transform
  - two-dimensional Haar + smoothing filter
  - biorthogonal 4-tap filter
  - full decomposition (recurse on LL band until 1 pixel remaining)
- block motion compensation
  - half-pixel precision
  - block size can vary but all blocks in one frame must be the same size
  - intra blocks can be split into 4 sub-blocks
- adaptive quantization
- hierarchical zero coefficient coding (HZCC) using run-length encoding
- intra frames and inter frames with variable length closed GOP (group of pictures)
  - no bidirectional prediction, only forward prediction using previous picture as reference
- entropy coding using interleaved exponential-golomb codes

---

[1] Hybrid refers to the use of image transform coding and motion estimation/compensation. This is in contrast to coding a video as if it were a 3D signal.

B. BITSTREAM DECODING

The fundamental unit of digital information is a binary digit, or bit, which can be either a 0 or 1. A byte is defined as 8 bits.

DSV operates on bits exclusively and only uses these types of binary encoding:

Encoding type	Description
Raw	The bits stored directly represent the binary digits of a number.
Interleaved Exponential-Golomb Code (UEG)	A method of encoding arbitrarily large positive (including zero) integers in a variable number of bits.
Signed Interleaved Exponential-Golomb Code (SEG)	A method of encoding arbitrarily large integers in a variable number of bits.
Non-zero Interleaved Exponential-Golomb Code (NEG)	A method of encoding arbitrarily large non-zero integers in a variable number of bits.

Some parts of the DSV bitstream contain a form of data encoding called Zero Bit Run-Length-Encoding (ZBRLE). ZBRLE operates on bits and consists of a concatenated sequence of UEG codes. The value of each decoded UEG code is the number of zero bits that it represents. If no zero bits remain, the decoded bit is a one. Decoding a ZBRLE bit is described in the following pseudo code:

```
if (number_of_zeros_remaining[2] == 0) {
    number_of_zeros_remaining = read_UEG();
    return (number_of_zeros_remaining == 0 ? 1 : 0);
}
number_of_zeros_remaining--;
return (number_of_zeros_remaining == 0 ? 1 : 0);
```

A DSV stream consists of a number of packets. Packets contain a header and a payload. The data in the following tables in each subsection is sequential in the bitstream.

[2] *number_of_zeros_remaining* is a state variable associated with the ZBRLE stream being decoded.

B.1 Packet Header

Data	Contains	Description
8 bits	ASCII character ‘D’ Hex 0x44	First character of the Four Character Code (FourCC)
8 bits	ASCII character ’S’ Hex 0x53	A method of encoding arbitrarily large positive (including zero) integers in a variable number of bits.
8 bits	ASCII character ‘V’ Hex 0x56	A method of encoding arbitrarily large integers in a variable number of bits.
8 bits	ASCII character ‘1’ Hex 0x31	A method of encoding arbitrarily large non-zero integers in a variable number of bits.
8 bits	Minor version number	Minor version number of the DSV specification the encoder adhered to.
8 bits	Packet type	A bit pattern which describes the contents packet.
32 bits	Previous link offset	Length (in bytes) of the previous packet.
32 bits	Next link offset	Length (in bytes) of the current packet.

B.1.1 Packet Type

Packet Type	Description
Equal to 0x00	Packet is metadata.
Equal to 0x10	Packet is an end of stream marker.
Packet Type bit 0x04 is set	Packet is a picture. If Packet Type bit 0x01 is set, it has a reference picture. If Packet Type bit 0x02 is set, it is a reference picture.

B.2 Packet Payload

B.2.1 Metadata Packet

Data	Contains	Description
UEG encoded	Width of video frame	Horizontal resolution in pixels.
UEG encoded	Height of video frame	Vertical resolution in pixels.
UEG encoded	Chroma subsampling	4:4:4 = 0x00 4:2:2 = 0x04 4:2:0 = 0x05 4:1:1 = 0x08
UEG encoded	Frames per second (FPS) numerator	Used in combination with the FPS denominator to define the (potentially fractional) frame rate of the video. FPS = numerator / denominator
UEG encoded	Frames per second (FPS) denominator	Used in combination with the FPS numerator to define the (potentially fractional) frame rate of the video. FPS = numerator / denominator
UEG encoded	Aspect ratio numerator	Used in combination with the aspect ratio denominator to define the aspect ratio of the video. Aspect ratio = numerator / denominator
UEG encoded	Aspect ratio denominator	Used in combination with the aspect ratio numerator to define the aspect ratio of the video. Aspect ratio = numerator / denominator

B.2.2 End of Stream Packet

Data	Contains	Description
None	Nothing	End of stream packet has no payload.

B.2.3 Picture Packet

Data	Contains	Valid Range	Description
32 bits	Frame number	All positive integers, including zero.	ID of the frame
UEG encoded	Quarter of the motion block width	[4, 16]	Motion block width can be calculated by multiplying the decoded number by four.
UEG encoded	Quarter of the motion block height	[4, 16]	Motion block height can be calculated by multiplying the decoded number by four.

The number of motion blocks in the horizontal and vertical direction can be derived like so:

```
horiz_blocks = (frame_width + block_width - 1) / block_width
vert_blocks = (frame_height + block_height - 1) / block_height
```

Where *frame_width* and *frame_height* come from the metadata packet and *block_width* and *block_height* are the decoded “quarter of the motion block width/height” values, respectively, multiplied by four.

B.2.3.1 Stability Blocks

Every picture has a stability block subsection.

Data	Contains	Valid Range	Description
UEG encoded Align to next byte afterwards.	Length of the stability blocks subsection in bytes.	All positive integers, including zero.	Number of bytes to skip to reach the next subsection of the picture packet.
ZBRLE encoded	One bit for each motion block in the frame, used for adaptive quantization.	N/A	Compute (horiz_blocks * vert_blocks) to determine how many ZBRLE bits must be read.

B.2.3.2 Motion Data

Only exists for pictures that have a reference (inter frames). Align to next byte before starting the decoding process. The subsections occur in order, data for each motion block is not interleaved. First, a UEG encoded integer is read before each subsection. This integer is the length in bytes of the subsection that follows it. The stream is aligned to the next byte after each UEG decoded.

Subsection	Data	Contains	Description
Mode	ZBRLE encoded	Block inter/ intra modes.	0x00 = block is INTER 0x01 = block is INTRA
Motion vector X components	SEG encoded	X component of each block's motion vector.	Exists only if block is INTER. To reconstruct the actual component, its prediction needs to be added to the decoded value.
Motion vector Y components	SEG encoded	Y component of each block's motion vector.	Exists only if block is INTER. To reconstruct the actual component, its prediction needs to be added to the decoded value.
Intra sub-block masks	Raw - Described below	Bit masks of the sub-blocks.	Exists only if block is INTRA.

Motion Vector Prediction

Motion vector components for a given block are predicted from the vector components of its left, upper, and upper left neighbors in the block grid. If the current block is on the left edge, top edge, or top left, the unavailable neighbors' component used for prediction will be zero.

The prediction using the three neighbors is obtained by:

```
motion_vector_prediction(LEFT, TOP, TOP_LEFT)
{
    dif = LEFT + TOP - TOP_LEFT;
    lft = absolute_value(dif - LEFT);
    top = absolute_value(dif - TOP);
    if (lft < top) {
        return LEFT;
    }
    return TOP;
}
```

Intra Sub-Block Mask Decoding

Intra blocks are split into four sub-blocks. Each sub-block can then either be coded as intra or inter with a zero motion vector.

1	2
4	8

Mask value of each sub-block

Decoding the mask is done by first reading a bit. If the bit was a 1 then all the sub-blocks are marked as intra, otherwise 4 bits are read and assigned to the mask. This is because of the observation that intra blocks where all sub-blocks are intra as well are much more common. Pseudo code for the described behavior:

```
if (read_bit() == 1) {
    submask = ALL_INTRA;
} else {
    submask = read_4_bits();
}
```

B.2.3.3 Image Data

Every picture has an image data subsection.

After aligning the stream to the next byte, 11 bits are read. This is the quantization parameter for this frame.

Plane Decoding

Data	Contains	Description
32 bits	Length in bytes of encoded plane.	Amount of bytes to skip to get to the next color plane.
Complex - Coefficient Decoding	Encoded coefficients of plane.	Subband coefficients of the color plane.

The above is performed for each of the three color frames, making sure to byte align the stream after each plane.

Coefficient Decoding

Data	Contains	Description
SEG encoded	LL coefficient of the entire frame.	All the low pass energy of the image. It is not quantized and often has a very large magnitude.
Complex - HZCC	Encoded coefficients of plane.	Subband coefficients of the color plane.
8 bits	End of Plane Symbol Hex 0x55	Check decoded value matches 0x55 after HZCC decoding to see if there was an error in the plane's bit stream.

Decoding Hierarchical Zero Coefficient Coding (HZCC) Data

Align the stream to the next byte before beginning.

Data	Contains	Description
32 bits	Number of runs.	Total number of zero runs in the coefficients.
UEG encoded	Length of first zero run.	How many of the first coefficients are zero.

Begin loop through the ‘LL’ subband. If there are no more zeros left in the current run, read the following data to get the next run and then continue the loop.

Data	Contains	Description
UEG encoded	Length of next zero run.	How many of the next set of coefficients are zero.
NEG encoded	Non-zero coefficient.	Quantized, needs to be dequantized.

After decoding the ‘LL’ band, begin decoding the 3 highest frequency subbands in order from lowest to highest. The decoding process is identical to that used in the ‘LL’ subband, described in the table above.

Once all subbands are decoded, align to the next byte and continue with the error check in the **Coefficient Decoding** table. Set the first coefficient of the plane to be the LL coefficient decoded in the **Coefficient Decoding** table.

Repeat for each plane until image data is fully decoded.

NOTE: **all** subband coefficients must be stored as signed 32-bit integers.

At this point, no more data needs to be read from the stream for this frame.

C. IMAGE RECONSTRUCTION

C.1 Subband Order and Traversal

Only the three highest frequency subbands are treated specially in DSV. The other lower frequency subbands get grouped together into the LL region in the context of HZCC and quantization. The subbands are traversed in raster scan order. Raster scan order is left to right, top to bottom. Pseudo code for such a traversal is given below:

```
for (y = top; y < bottom; y++) {
    for (x = left; x < right; x++) {
        /* data is at [x, y] */
    }
}
```

X increases horizontally, Y increases vertically

<table>
<tr><td>LL</td><td>LH
level 0</td><td rowspan="2">LH level 1</td><td rowspan="3" colspan="2">LH level 2
(highest frequency)</td></tr>
<tr><td>HL
level 0</td><td>HH
level 0</td></tr>
<tr><td colspan="2">HL level 1</td><td>HH level 1</td></tr>
<tr><td colspan="3">HL level 2
(highest frequency)</td><td colspan="2">HH level 2
(highest frequency)</td></tr>
</table>

DSV subband structure in 2D

NOTE: when computing subband dimensions through subdivision, make sure to round up to the next integer.

C.2 Dequantization

In DSV, each level of the structure is quantized in a different fashion.
The quantization parameter ***Q*** is passed into the HZCC decoder and is limited to **512** for chroma planes. The minimum quantization parameter ***MINQUANT*** is **16** for all levels except the highest frequency level which quantizes values differently altogether and is described later.

Some important data needed for correct dequantization:

Data	Denoted As	Contains
Stability Blocks	boolean = STABLE(x, y)	Bit array with enough elements for every block. A value of 0 at [x, y] means that block was determined to be unstable by the encoder. A value of 1 at [x, y] means that block was determined to be stable by the encoder.
Intra Blocks	boolean = INTRA(x, y)	Bit array with enough elements for every block. A value of 0 at [x, y] means that block is an INTER block. A value of 1 at [x, y] means that block is an INTRA block.
Is Inter Frame	boolean = IS_INTER	Boolean value denoting whether this frame was an intra frame or an inter frame. false = intra frame true = inter frame

C.2.1 Dequantization Functions

Dequantization is performed through dead-zone dequantization for all subbands besides the highest frequency subband. The pseudo code for the dequantization functions is given below:

```
dequantize_lower_frequency(VALUE, Q)
{
    if (VALUE < 0) {
        return -((-VALUE * (Q * 2) + Q) / 2);
    }
    return (VALUE * (Q * 2) + Q) / 2;
}

dequantize_highest_frequency(VALUE, Q)
{
    return (VALUE * (2 ^^ Q)); /* two to the power of Q */
}
```

C.2.2 Quantization Parameter Derivation

The modified quantization parameter ***MQ*** for a given level is derived as follows.

If level is not highest frequency level:

```
get_quant_lower_frequency(Q, level)
{
   MQ = Q;
   if (IS_INTER) {
      MQ = (MQ * 3) / 2;
   }
   if (level == 1) {
      MQ = (MQ * 2) / 3;
   } else if (level == 2) {
      MQ = (MQ * 3) / 2;
   }
   if (MQ < MINQUANT) {
      return MINQUANT;
   }
   return MQ;
}
```

If level is highest frequency level:

```
get_quant_highest_frequency(Q, level)
{
   MQ = integer_log_base_2(get_quant_lower_frequency(Q, level))
   if (IS_INTER) {
      MQ = CLAMP(MQ - 1, 1, 24);
   } else { /* is intra */
      MQ = CLAMP(MQ - 3, 1, 24);
   }
}
```

The ***CLAMP*** function is defined as:

```
CLAMP(x, min, max) /* clamp x between [min, max] */
     return ((x) < (min) ? (min) : ((x) > (max) ? (max) : (x)))
```

C.2.3 LL Subband Dequantization

Each NEG encoded coefficient that is decoded in the LL subband is dequantized as follows:

```
for (y = top; y < bottom; y++) {
    for (x = left; x < right; x++) {
        MQ = get_quant_lower_frequency(Q, LL);
        V = read_NEG();
        decoded[x,y] = dequantize_lower_frequency(V, MQ);
    }
}
```

C.2.4 Higher Level Subband Dequantization

The higher levels take the 2D position of the coefficient into account for deriving the true ***MQ***, denoted as ***TMQ*** from now on.

```
TMQ_for_position(X, Y, MQ)
{
    if (INTRA(X, Y)) {
        MQ /= 4;
    } else if (STABLE(X, Y)) {
        MQ /= 2;
    }
    return MQ < MINQUANT ? MINQUANT : MQ;
}
```

Each NEG encoded coefficient that is decoded in the higher level subbands is dequantized as follows:

NOTE: this does not include the highest level.

```
highest_level = 3;

for (level = 0; level < (highest_level - 1); level++) {
for (each subband in order LH, HL, HH) /* NOTE no bracket */
    for (y = top; y < bottom; y++) {
        for (x = left; x < right; x++) {
            MQ = get_quant_lower_frequency(Q, level);
            TMQ = TMQ_for_position(x, y, MQ);
            V = read_NEG();
            decoded[x,y] = dequantize_lower_frequency(V, TMQ);
        }
```

```
    }
}
```

C.2.5 Highest Level Subband Dequantization

The highest level also takes the 2D position of the coefficient into account for deriving the ***TMQ***.

```
MQ = integer_log_base_2(get_quant_lower_frequency(Q, level));
level = highest_level - 1;
for (each subband in order LH, HL, HH) {
     for (y = top; y < bottom; y++) {
        for (x = left; x < right; x++) {
            if (STABLE(x, y) OR INTRA(x, y)) {
               TMQ = get_quant_highest_frequency(Q, level);
            } else {
               TMQ = MQ;
            }
            V = read_NEG();
            decoded[x,y] = dequantize_highest_frequency(V, TMQ);
        }
    }
}
```

C.3 Subband Transforms

DSV performs a full decomposition. A full decomposition is a recursive subband transform on the LL subband of the structure until the LL subband is a single pixel.

DSV uses two different transforms, the *Haar Transform* (HT) and a *Biorthogonal 4-Tap Transform* (B4T).

The ***MIN*** function is defined as:

```
MIN(a, b) ((a) < (b) ? (a) : (b))
```

The ***MAX*** function is defined as:

```
MAX(a, b) ((a) > (b) ? (a) : (b))
```

The functions ***ROUND2, ROUND4,*** and ***ROUND8*** simply divide the argument by the number in the function's name while also rounding away from zero (properly accounting for negatives).
For example:

```
ROUND4(V)
    if (V < 0) {
        return -(((-V) + 2) / 4);
    }
    return (V + 2) / 4;
```

C.3.1 Haar Transform

C.3.1.1 LL Coefficient Scaling

The ***LVL_TEST*** condition is defined as:

```
if (IS_INTRA) {
    LVL_TEST = true
} else {
    if (level > 1) { /* if level is not the highest level */
        LVL_TEST = true
    } else {
        LVL_TEST = false
    }
}
```

It is used to determine whether the LL coefficient should be scaled or not.

C.3.1.2 Forward Transform

The HT is the simplest transform, it operates on 2x2 blocks of pixels.

Given a 2x2 block of pixels in the form:
{ x0, x1,
  x2, x3 }
the forward HT in DSV is defined as follows:

```
if (LVL_TEST) {
    LL = (x0 + x1 + x2 + x3) * 4 / 5;
} else {
    LL = (x0 + x1 + x2 + x3);
}
LH = (x0 - x1 + x2 - x3);
HL = (x0 + x1 - x2 - x3);
HH = (x0 - x1 - x2 + x3);
```

C.3.1.3 Simple Inverse Transform

The inverse HT in DSV is defined as follows:

```
if (LVL_TEST) {
    LL = (LL * 5 / 4);
}
x0 = (LL + LH + HL + HH) / 4;
x1 = (LL - LH + HL - HH) / 4;
x2 = (LL + LH - HL - HH) / 4;
x3 = (LL - LH - HL + HH) / 4;
```

C.3.1.4 Filtered Inverse Transform

The filtered inverse HT in DSV is defined as follows:

```
if (LVL_TEST) {
    LL = (LL * 5 / 4);
}
if (NOT THE FIRST OR LAST COEFFICIENT IN ROW) {
    if (LVL_TEST) {
        lp = p_LL * 5 / 4; /* previous LL on X axis (x - 1) */
        ln = n_LL * 5 / 4; /* next LL on X axis (x + 1) */
    } else {
        lp = p_LL; /* previous LL on X axis (x - 1) */
        ln = n_LL; /* next LL on X axis (x + 1) */
    }
    mx = LL - ln; /* find difference between LL values */
    mn = lp - LL;
    if (mn > mx) SWAP(mn, mx)
    mx = MIN(mx, 0); /* must be negative or zero */
    mn = MAX(mn, 0); /* must be positive or zero */
    /* if they are not zero, then there is a potential
     * consistent smooth gradient between them */
    if (mx != mn) {
        n = ROUND2(CLAMP(ROUND4(lp - ln), mx, mn) - (LH * 2));
        LH += CLAMP(n, -HQP, HQP); /* nudge LH to smooth it */
    }
}
/* do the same as above but in the Y direction, but nudging HL
 * instead of LH */
x0 = (LL + LH + HL + HH) / 4;
x1 = (LL - LH + HL - HH) / 4;
x2 = (LL + LH - HL - HH) / 4;
x3 = (LL - LH - HL + HH) / 4;
```

The ***HQP*** variable is approximately half the quantization value used for the subband. Due to the adaptive quantization, we cater to the regions that are stable and smooth those properly since they are more likely to be noticed when improperly filtered.

HQP for a subband level ***level***[3] can be derived as follows:

```
get_HQP(level, Q)
{
    if (level > 3) {
        HQP = get_quant_lower_frequency(Q, LL) / 2;
    } else {
        HQP = get_quant_lower_frequency(Q, 3 - level);
        if (i == 1) {
            HQP = integer_log_base_2(HQP);
            if (IS_INTRA) {
                HQP = CLAMP(HQP - 3, 1, 24);
            } else {
                HQP = CLAMP(HQP - 1, 1, 24);
            }
            HQP = (2 ^^ HQP) / 2; /* two to the power of HQP */
    }
    return HQP / 2;
}
```

NOTE: the Haar subband transforms must properly account for odd dimensions by treating any pixel or coefficient in the 2x2 block that lies outside of the image as though it has a value of zero.

[3] ***level*** in this context is a different numerical value than ***level*** seen in HZCC. In the case of the subband transforms, level 1 is the highest frequency subband and N is the lowest frequency.

C.3.2 Biorthogonal 4-Tap Transform

The B4T transform is ONLY done to level 1 (highest frequency) and ONLY done to INTRA frames. The four taps are 1, 3, 3, 1.

C.3.2.1 Forward Transform

The forward 2D B4T transform is defined as follows:

```
for (every row)
   forward_B4T(row, WIDTH, 1)
for (every column)
   forward_B4T(column, HEIGHT, WIDTH)

forward_B4T(in, n, s)
    x0 = in[1 * s]; /* top */
    x1 = in[0 * s];
    x2 = in[1 * s];
    x3 = in[2 * s];
    t1 = x1 * 3;
    t2 = x2 * 3;
    out[0 * s] = ROUND2(t1 + t2 - x0 - x3);
    out[(n / 2) * s] = ROUND2(x0 - t1 + t2 - x3);
    x0 = x2;
    x1 = x3;
    for (i = 1; i < n - 4; i += 2) {
        x2 = in[s + (i + 1) * s];
        x3 = in[s + (i + 2) * s];
        t1 = x1 * 3;
        t2 = x2 * 3;
        out[s + (i / 2) * s] = ROUND2(t1 + t2 - x0 - x3);
        out[s + ((i + n) / 2) * s] = ROUND2(x0 - t1 + t2 - x3);
        x0 = x2;
        x1 = x3;
    }
    x2 = in[s + (i + 1) * s]; /* bottom */
    x3 = x2;
    t1 = x1 * 3;
    t2 = x2 * 3;
    out[s + (i / 2) * s] = ROUND2(t1 + t2 - x0 - x3);
    out[s + (i + n) / 2) * s] = ROUND2(x0 - t1 + t2 - x3);
```

C.3.2.2 Inverse Transform

The inverse 2D B4T transform is defined as follows:

```
for (every column)
   inverse_B4T(column, HEIGHT, WIDTH)
for (every row)
   inverse_B4T(row, WIDTH, 1)

inverse_B4T(in, n, s)
    L0 = in[0 * s]; /* top */
    H0 = in[(n / 2) * s];
    L1 = L0;
    H1 = H0;
    L3 = L1 * 3;
    H3 = H1 * 3;
    out[0] = ROUND8(L0 + L3 + H0 - H3);
    L0 = L1;
    H0 = H1;
    L1 = in[((1 + 1) / 2) * s];
    H1 = in[((1 + 1 + n) / 2) * s];
    out[s] = ROUND8(L3 + L1 + H3 - H1);
    for (i = 1; i < n - 4; i += 2) {
        L3 = L1 * 3;
        H3 = H1 * 3;
        out[s + (i + 0) * s] = ROUND8(L0 + L3 + H0 - H3);
        L0 = L1;
        H0 = H1;
        L1 = in[s + ((i + 1) / 2) * s];
        H1 = in[s + ((i + 1 + n) / 2) * s];
        out[s + (i + 1) * s] = ROUND8(L3 + L1 + H3 - H1);
    }
    L3 = L1 * 3; /* bottom */
    H3 = H1 * 3;
    out[s + (i + 0) * s] = ROUND8(L0 + L3 + H0 - H3);
    out[s + (i + 1) * s] = ROUND8(L3 + L1 + H3 - H1);
```

C.3.3 Subband Recomposition

Subband recomposition is achieved by the following pseudo code:

```
num_levels(width, height)
    max = MAX(width, height)
    lb2 = integer_log_base_2(max)
    if (max > (2 ^^ lb2)) { /* two to the power of lb2 */
       lb2 += 1
    }
    return lb2;

/* coefs = subband coefficients, w = width, h = height,
 * component = which color plane is to be recomposed,
 * Q = quantization parameter
 */
recompose(coefs, w, h, component, Q)
    levels = num_levels(w, h);

    if (component = LUMA) {
        for (i = levels; i > 0; i--) {
            HQP = get_HQP(i, Q);
            if (IS_INTRA AND i == 1) {
                INVERSE_B4T_2D(coefs, w, h);
            } else {
                INVERSE_HAAR_FILTERED(coefs, w, h, i, HQP);
            }
        }
    } else {
        for (i = levels; i > 0; i--) {
            if (IS_INTRA AND i == 1) {
                INVERSE_B4T_2D(coefs, w, h);
            } else {
                INVERSE_HAAR_SIMPLE(coefs, w, h, i);
            }
        }
    }
    for (all recomposed pixels)
        add 128 to pixel and CLAMP it to [0, 255]
```

D. MOTION COMPENSATION

Motion compensation is done per plane. Do not expand the bit depth of the image at any point to do motion compensation.

D.1 Compensating Inter Blocks

Inter blocks are compensated using half-pixel precision. The luma plane has a special filter for computing the half pixels and the chroma planes use simple bilinear filtering.
Motion vectors for chroma are derived (in the case of chroma subsampled formats) by dividing the x and/or y component of the motion vector by the ratio of chroma subsampling in that direction. For example, in a 4:2:0 scheme, both the x and y components are divided by two for the chroma planes.

D.1.1 Luma Half-Pixel Filter

```
9 * (p[0] + p[1]) - (p[-1] + p[2])
```

D.1.2 Chroma Half-Pixel Filter

```
(p[0] + p[1] + 1) / 2
```

Compute the potentially filtered block and add it to the current block. Subtract 128 from each pixel and ***CLAMP*** each pixel to [0, 255].

D.2 Compensating Intra Blocks

For each of the four intra sub-blocks, calculate the average value of the sub-block in the reference picture at the same location and add it to the pixels in the current sub-block. If the block is not intra, add the reference sub-block directly.
Subtract 128 from each pixel and ***CLAMP*** each pixel to [0, 255].

D.3 Caveat For Encoder

Due to the restriction of 8 bits per channel, the compensation cannot account for intra blocks that have large changes. Therefore, the encoder must simulate the reduced range intra block motion compensation to see if the block can be successfully reconstructed before it decides to label the block as intra.

Such a test can be conducted as follows, the threshold can be adjusted as desired:

```
/* src = current block of current frame
 * ref = current block of reference frame
 * w = width of block
 * h = height of block */

C(x)
    CLAMP(x, 0, 255)

block_intra_test(src, ref, w, h)
    THRESHOLD = ((w * h) / 4);
    AVG = COMPUTE_AVERAGE_OF_REFERENCE_BLOCK();
    NUM_BAD = 0
    for (y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            d = C((AVG + C((src[x, y] - AVG) + 128)) - 128);
            if (absolute_value(d - src[x, y]) != 0) {
                NUM_BAD += 1;
            }
        }
        if (NUM_BAD > THRESHOLD) {
            return INTER; /* do inter, not intra */
        }
    }
    return INTRA; /* keep the block intra */
```

E. Rationale and Extra Information

Why use a Subband Transform?

To avoid using the all-too-common block based Discrete Cosine Transform (DCT). While DSV1 is not as feature-filled, general purpose, or well defined as MPEG-1 or MPEG-2 video, it does serve to demonstrate the strengths and weaknesses of subband transforms in video compression.

Why do a full subband decomposition?

Mainly simplicity and the desire to avoid the separate compression that most other subband or wavelet based formats do on the DC/LL subband. In DSV1, that DC/LL subband is a single integer and so it can be simply packed into the bitstream without extra hassle.

Why do a Haar transform?

Haar is the simplest subband transform and the filter done on the luma plane gets rid of the blocking artifacts that naturally arise from the Haar transform's inability to capture smooth gradients. Due to the 2D Haar using only 2x2 regions of data, the intra block coding method used in DSV1 can be effectively done without introducing ugly edge artifacts that would otherwise be introduced by transforms that operate on a larger group of pixels.

Why not do Haar for everything?

While the Haar transform with the smoothing filter is adequate for lower frequency subbands, it struggles greatly with capturing high frequency details under high compression. This is why the biorthogonal 4-tap filter transform (B4T) is in the DSV1 specification. This filter helps effectively capture and represent high frequency details. The B4T is only used in intra frames to create the highest frequency subband. This is because the B4T causes noticeable artifacting around intra blocks in inter frames due to its filter length.

Why scale the LL coefficients in the Haar Transform?

Doing a full decomposition increases the magnitude of the LL coefficient greatly every level. Scaling the coefficient down by an appropriate amount helps reduce the magnitude (making the coefficients use fewer bits) but also comes at a slight cost in visual quality. It is fundamentally a tradeoff to allow more high frequency information to exist for the same bit rate.

Why interleaved exponential-Golomb codes?

Interleaving the ‘data’ and ‘follow’ bits of the exponential-Golomb code allows for the decoding to be done in a single for-loop rather than two separate loops.

Why no statistical entropy coding like Huffman or arithmetic coding?

Adds too much complexity. Universal codes like the interleaved exponential-Golomb code get the job done (well enough) with a negligible amount of extra complexity.

Why are the luma and chroma half-pixel filters different?

The filter used for luma is both fast and provides a good sharpness to artifact ratio in slow panning scenes. A better half-pixel filter can provide extremely accurate half-pixel data but it also adds too many sharp edges in the non overlapped block motion compensation scheme in DSV1. The sharp edges are very ugly and generally worse for compression.

Chroma is not as important and the blurring nature of the bilinear filter helps reduce chroma edge artifacts from the motion compensation.

What are the stability blocks?

They are a cheap way of allowing the encoder to convey varying importance to different regions of the video, letting the encoder create better subjective quality for the same bit rate. They are named stability blocks since they were originally designed to keep stable/low motion blocks of the video high quality. The encoder, however, is free to assign stability in any way it desires.

Why no B-frames?

Adds too much complexity to both encoder and decoder.

Why use dead-zone quantization?

Due to the way the coefficients are being encoded (RLE of zero runs), it is desirable to maximize the number of zero coefficients. Dead-zone quantization does this without effecting visual quality much since it just widens the range of values that get quantized to zero by a small amount.

How was the quantization scheme designed?

In short, plenty of trial and error based on the fundamentals of the HVS, coding performance, and visual quality.

The highest frequency data is heavily quantized but some high frequency detail is, however, relevant and important and that is why the stability blocks exist.

The highest frequency subband is quantized using powers of two because the highest frequency subband is three-quarters of each frame. It made sense to avoid integer division/multiplication and use bit shifts instead.

Why are the three highest frequency subbands treated differently?

Since virtually all of the coefficients in the transformed image belong to the three highest frequency subbands, they are where compression can be leveraged. Conveniently, they also represent higher frequency data which is generally more flexible in terms of quantization amount and able to be quantized in an adaptive fashion. Messing with the lower frequency subbands results in awful, large artifacts in simple scenes with large flat regions. Errors in the high frequency subbands, on the other hand, result in smaller, less perceptible, and less annoying artifacts.