Specification for WebP Lossless Bitstream

Jyrki Alakuijala, Ph.D.
Google, Inc.
2012-06-04

Abstract — WebP lossless is an image format for lossless compression of ARGB images. The lossless format stores and restores the pixel values exactly, including the color values for zero alpha pixels. The format uses subresolution images, recursively embedded into the format itself, for storing statistical data about the images, such as the used entropy codes, spatial predictors, color space conversion, and color table. LZ77, Huffman coding, and a color cache are used for compression of the bulk data. Decoding speeds faster than PNG have been demonstrated, as well as 25 % denser compression than what can be achieved using today's PNG format.

Table of contents

Nomenclature

ARGB — a pixel value consisting of alpha, red, green, and blue values.
ARGB image — a two-dimensional array containing ARGB pixels.
color cache — a small hash-addressed array to store recently used colors and to be able to recall them with shorter codes.

color indexing image — a one-dimensional image of colors that can be indexed using a small integer (up to 256 within WebP lossless).

color transform image — a two-dimensional subresolution image containing data about correlations of color components.

distance mapping — changes LZ77 distances to have the smallest values for pixels in 2d proximity.

entropy image — a two-dimensional subresolution image indicating which entropy coding should be used in a respective square in the image, i.e., each pixel is a meta Huffman code.

Huffman code — a classic way to do entropy coding where a smaller number of bits are used for more frequent codes.

LZ77 — dictionary-based sliding window compression algorithm that either emits symbols or describes them as sequences of past symbols.

meta Huffman code — a small integer (up to 16 bits) that indexes an element in the meta Huffman table.

predictor image — a two-dimensional subresolution image indicating which spatial predictor is used for a particular square in the image.

prefix coding — a way to entropy code larger integers that codes a few bits of the integer using an entropy code and codifies the remaining bits raw. This allows for the descriptions of the entropy codes to remain relatively small even when the range of symbols is large.

scan-line order — a processing order of pixels, left-to-right, top-to-bottom, starting from the left-hand-top pixel, proceeding towards right. Once a row is completed, continue from the left-hand column of the next row.

1 Introduction

This document describes the compressed data representation of a WebP lossless image. It is intended as a detailed reference for WebP lossless encoder and decoder implementation.

In this document, we use extensively the syntax of the C programming language to describe the bitstream, and assume the existence of a function for reading bits, `ReadBits(n)`. The bytes are read in the natural order of the stream containing them, and bits of each byte are read in the least-significant-bit-first order. When multiple bits are read at the same time the integer is constructed from the original data in the original order, the most significant bits of the returned integer are also the most significant bits of the original data. Thus the statement

```
b = ReadBits(2);
```

is equivalent with the two statements below:

```
b = ReadBits(1);
b |= ReadBits(1) << 1;
```

We assume that each color component (e.g. alpha, red, blue and green) is represented using an 8-bit byte. We define the corresponding type as uint8. A whole ARGB pixel is represented by a type called uint32, an unsigned integer consisting of 32 bits. In the code showing the behavior of the transformations, alpha value is codified in bits 31..24, red in bits 23..16, green in bits 15..8

and blue in bits 7..0, but implementations of the format are free to use another representation internally.

Broadly a WebP lossless image contains header data, transform information and actual image data. Headers contain width and height of the image. A WebP lossless image can go through five different types of transformation before being entropy encoded. The transform information in the bitstream contains the required data to apply the respective inverse transforms.

2 RIFF Header

The beginning of the header has the RIFF container. This consist of the following 21 bytes:

1. string “RIFF”
2. A little-endian 32 bit value of the block length, the whole size of the block controlled by the RIFF header. Normally this equals the payload size (file size subtracted by 8 bytes, i.e., 4 bytes for ‘RIFF’ identifier and 4 bytes for storing this value itself).
3. string “WEBP” (RIFF container name).
4. string “VP8L” (chunk tag for lossless encoded image data).
5. A little-endian 32-bit value of the number of bytes in the lossless stream.
6. One byte signature 0x2f.

First 28 bits of the bitstream specify the width and height of the image. Width and height are decoded as 14-bit integers as follows:

```
int image_width = ReadBits(14) + 1;
int image_height = ReadBits(14) + 1;
```

The 14-bit dynamics for image size limit the maximum size of a WebP lossless image to 16384×16384 pixels.

3 Transformations

Transformations are reversible manipulations of the image data that can reduce the remaining symbolic entropy by modeling spatial and color correlations. Transformations can make the final compression more dense.

An image can go through four types of transformations. A 1 bit indicates the presence of a transform. Every transform is allowed to be used only once. The transformations are used only for the main level ARGB image — the subresolution images have no transforms, not even the 0 bit indicating the end-of-transforms.

Typically an encoder would use these transforms to reduce the Shannon entropy in the residual image. Also, the transform data can be decided based on entropy minimization.

```
while (ReadBits(1)) {  // Transform present.
   // Decode transform type.
   enum TransformType transform_type = ReadBits(2);
   // Decode transform data.
   ...
}

// Decode actual image data (section 4).
```

If a transform is present then the next two bits specify the transform type. There are four types of transforms.

```
enum TransformType {
  PREDICTOR_TRANSFORM             = 0,
  COLOR_TRANSFORM                 = 1,
  SUBTRACT_GREEN                  = 2,
  COLOR_INDEXING_TRANSFORM        = 3,
};
```

The transform type is followed by the transform data. Transform data contains the required information to apply the inverse transform and depends on the transform type. Next we describe the transform data for different types.

Predictor transform

The predictor transform can be used to reduce entropy by exploiting the fact that neighboring pixels are often correlated. In the predictor transform, the current pixel value is predicted from the pixels already decoded (in scan-line order) and only the residual value (actual - predicted) is encoded. The *prediction mode* determines the type of prediction to use. We divide the image into squares and all the pixels in a square use same prediction mode.

The first 4 bits of prediction data define the block width and height in number of bits. The number of block columns, *block_xsize*, is used in indexing two-dimensionally.

```
int size_bits = ReadBits(3) + 2;
int block_width = (1 << size_bits);
int block_height = (1 << size_bits);
#define DIV_ROUND_UP(num, den) ((num) + (den) - 1) / (den))
int block_xsize = DIV_ROUND_UP(image_width, 1 << size_bits);
```

The transform data contains the prediction mode for each block of the image. All the `block_width * block_height` pixels of a block use same prediction mode. The prediction modes are treated as pixels of an image and encoded using the same techniques described in chapter 4.

For a pixel *x, y*, one can compute the respective filter block address by:

```
int block_index = (y >> size_bits) * block_xsize +
                  (x >> size_bits);
```

There are 14 different prediction modes. In each prediction mode, the current pixel value is predicted from one or more neighboring pixels whose values are already known.

We choose the neighboring pixels (TL, T, TR, and L) of the current pixel (P) as follows:

```
O    O    O    O    O    O    O    O    O    O    O
O    O    O    O    O    O    O    O    O    O    O
O    O    O    O    TL   T    TR   O    O    O    O
O    O    O    O    L    P    X    X    X    X    X
X    X    X    X    X    X    X    X    X    X    X
X    X    X    X    X    X    X    X    X    X    X
```

where TL means top-left, T top, TR top-right, L left pixel.
At the time of predicting a value for P, all pixels O, TL, T, TR and L have been already processed, and pixel P and all pixels X are unknown.

Given the above neighboring pixels, the different prediction modes are defined as follows.

Mode	**Predicted value of each channel of the current pixel**
0	0xff000000 (represents solid black color in ARGB)
1	L
2	T
3	TR
4	TL
5	Average2(Average2(L, TR), T)
6	Average2(L, TL)
7	Average2(L, T)
8	Average2(TL, T)
9	Average2(T, TR)
10	Average2(Average2(L, TL), Average2(T, TR))
11	Select(L, T, TL)
12	ClampedAddSubtractFull(L, T, TL)

13	ClampedAddSubtractHalf(Average2(L, T), TL)

`Average2` is defined as follows for each ARGB component:

```
uint8 Average2(uint8 a, uint8 b) {
  return (a + b) / 2;
}
```

The Select predictor is defined as follows:

```
uint32 Select(uint32 L, uint32 T, uint32 TL) {
  // L = left pixel, T = top pixel, TL = top left pixel.

  // ARGB component estimates for prediction.
  int pAlpha = ALPHA(L) + ALPHA(T) - ALPHA(TL);
  int pRed = RED(L) + RED(T) - RED(TL);
  int pGreen = GREEN(L) + GREEN(T) - GREEN(TL);
  int pBlue = BLUE(L) + BLUE(T) - BLUE(TL);

  // Manhattan distances to estimates for left and top pixels.
  int pL = abs(pAlpha - ALPHA(L)) + abs(pRed - RED(L)) +
           abs(pGreen - GREEN(L)) + abs(pBlue - BLUE(L));
  int pT = abs(pAlpha - ALPHA(T)) + abs(pRed - RED(T)) +
           abs(pGreen - GREEN(T)) + abs(pBlue - BLUE(T));

  // Return either left or top, the one closer to the prediction.
  if (pL <= pT) {
    return L;
  } else {
    return T;
  }
}
```

The function `ClampedAddSubstractFull` and `ClampedAddSubstractHalf` are performed for each ARGB component as follows:

```
// Clamp the input value between 0 and 255.
int Clamp(int a) {
  return (a < 0) ? 0 : (a > 255) ?  255 : a;
}
```

```
int ClampAddSubtractFull(int a, int b, int c) {
  return Clamp(a + b - c);
}
```

```
int ClampAddSubtractHalf(int a, int b) {
  return Clamp(a + (a - b) / 2);
}
```

There are special handling rules for some border pixels. If there is a prediction transform, regardless of the mode [0..13] for these pixels, the predicted value for the left-topmost pixel of the image is 0xff000000, L-pixel for all pixels on the top row, and T-pixel for all pixels on the leftmost column.

Addressing the TR-pixel for pixels on the rightmost column is exceptional. The pixels on the rightmost column are predicted by using the modes [0..13] just like pixels not on border, but by using the leftmost pixel on the same row as the current TR-pixel. The TR-pixel offset in memory is the same for border and non-border pixels.

Color Transform

The goal of the color transform is to decorrelate the R, G and B values of each pixel. Color transform keeps the green (G) value as it is, transforms red (R) based on green and transforms blue (B) based on green and then based on red.

As is the case for the predictor transform, first the image is divided into blocks and the same transform mode is used for all the pixels in a block. For each block there are three types of color transform elements.

```
typedef struct {
 uint8 green_to_red;
 uint8 green_to_blue;
 uint8 red_to_blue;
} ColorTransformElement;
```

The actual color transformation is done by defining a color transform delta. The color transform delta depends on the `ColorTransformElement` which is same for all the pixels in a particular block. The delta is added during color transform. The inverse color transform then is just subtracting those deltas.

The color transform function is defined as follows:

```
void ColorTransform(uint8 red, uint8 blue, uint8 green,
                    ColorTransformElement *trans,
                    uint8 *new_red, uint8 *new_blue) {
  // Transformed values of red and blue components
  uint32 tmp_red = red;
  uint32 tmp_blue = blue;

  // Applying transform is just adding the transform deltas
  tmp_red  += ColorTransformDelta(trans->green_to_red, green);
  tmp_blue += ColorTransformDelta(trans->green_to_blue, green);
  tmp_blue += ColorTransformDelta(trans->red_to_blue, red);

  *new_red = tmp_red & 0xff;
  *new_blue = tmp_blue & 0xff;
}
```

`ColorTransformDelta` is computed using a signed 8-bit integer representing a 3.5-fixed-point number, and a signed 8-bit RGB color channel (c) [-128..127] and is defined as follows:

```
int8 ColorTransformDelta(int8 t, int8 c) {
  return (t * c) >> 5;
}
```

The multiplication is to be done using more precision (with at least 16 bit dynamics). The sign extension property of the shift operation does not matter here: only the lowest 8 bits are used from the result, and there the sign extension shifting and unsigned shifting are consistent with each other.

Now we describe the contents of color transform data so that decoding can apply the inverse color transform and recover the original red and blue values. The first 4 bits of the color transform data contain the width and height of the image block in number of bits, just like the predictor transform:

```
int size_bits = ReadStream(3) + 2;
int block_width = 1 << size_bits;
int block_height = 1 << size_bits;
```

The remaining part of the color transform data contains ColorTransformElement instances corresponding to each block of the image. ColorTransformElement instances are treated as pixels of an image and encoded using the methods described in section 4.

During decoding ColorTransformElement instances of the blocks are decoded and the inverse color transform is applied on the ARGB values of the pixels. As mentioned earlier that inverse color transform is just subtracting ColorTransformElement values from the red and blue channels.

```
void InverseTransform(uint8 red, uint8 green, uint8 blue,
                      ColorTransfromElement *p,
                      uint8 *new_red, uint8 *new_blue) {
  // Applying inverse transform is just subtracting the
  // color transform deltas
  red  -= ColorTransformDelta(p->green_to_red_,  green);
  blue -= ColorTransformDelta(p->green_to_blue_, green);
  blue -= ColorTransformDelta(p->red_to_blue_, red & 0xff);

  *new_red = red & 0xff;
  *new_blue = blue & 0xff;
}
```

Subtract Green Transform

The subtract green transform subtracts green values from red and blue values of each pixel. When this transform is present, the decoder needs to add the green value to both red and blue. There is no data associated with this transform. The decoder applies the inverse transform as follows:

```
void AddGreenToBlueAndRed(uint8 green, uint8 *red, uint8 *blue) {
  *red  = (*red  + green) & 0xff;
  *blue = (*blue + green) & 0xff;
}
```

This transform is redundant as it can be modeled using the color transform. This transform is still often useful, and since it can extend the dynamics of the color transform, and there is no additional data here, this transform can be coded using less bits than a full blown color transform.

Color Indexing Transform

If there are not many unique values of the pixels then it may be more efficient to create a color index array and replace the pixel values by the indices to this color index array. Color indexing transform is used to achieve that. In the context of the WebP lossless, we specifically do not call this transform a palette transform, since another slightly similar, but more dynamic concept exists within WebP lossless encoding, called color cache.

The color indexing transform checks for the number of unique ARGB values in the image. If that number is below a threshold (256), it creates an array of those ARGB values is created which replaces the pixel values with the corresponding index. The green channel of the pixels are replaced with the index, all alpha values are set to 255, all red and blue values to 0.

The transform data contains color table size and the entries in the color table. The decoder reads the color indexing transform data as follow:

```
// 8 bit value for color table size
int color_table_size = ReadStream(8) + 1;
```

The color table is stored using the image storage format itself. The color table can be obtained by reading an image, without the RIFF header, image size, and transforms, assuming an height of one pixel, and a width of color_table_size. The color table is always subtraction coded for reducing the entropy of this image. The deltas of palette colors contain typically much less entropy than the colors themselves leading to significant savings for smaller images. In decoding, every final color in the color table can be obtained by adding the previous color component values, by each ARGB-component separately and storing the least significant 8 bits of the result.

The inverse transform for the image is simply replacing the pixel values (which are indices to the color table) with the actual color table values. The indexing is done based on the green component of the ARGB color.

```
// Inverse transform
argb = color_table[GREEN(argb)];
```

When the color table is of a small size (equal to or less than 16 colors), several pixels are bundled into a single pixel. The pixel bundling packs several (2, 4, or 8) pixels into a single pixel reducing the image width respectively. Pixel bundling allows for a more efficient joint distribution entropy coding of neighboring pixels, and gives some arithmetic coding like benefits to the entropy code, but it can only be used when there is a small amount of unique values.

color_table_size specifies how many pixels are combined together:

```
int width_bits;
if (color_table_size <= 2) {
  width_bits = 3;
} else if (color_table_size <= 4) {
  width_bits = 2;
} else if (color_table_size <= 16) {
  width_bits = 1;
} else {
  width_bits = 0;
}
```

The *width_bits* has a value of 0, 1, 2 or 3. A value of 0 indicates no pixel bundling to be done for the image. A value of 1 indicates that two pixels are combined together, and each pixel has a range of [0..15]. A value of 2 indicates that four pixels are combined together, and each pixel has a range of [0..3]. A value of 3 indicates that eight pixels are combined together and each pixels has a range of [0..1], i.e., a binary value.

The values are packed into the green component as follows:

- *width_bits = 1:* for every x value where x ≡ 0 (mod 2), a green value at x is positioned into the 4 least-significant bits of the green value at x / 2, a green value at x + 1 is positioned into the 4 most-significant bits of the green value at x / 2.
- *width_bits = 2:* for every x value where x ≡ 0 (mod 4), a green value at x is positioned into the 2 least-significant bits of the green value at x / 4, green values at x + 1 to x + 3 in order to the more significant bits of the green value at x / 4.

- *width_bits = 3:* for every x value where x ≡ 0 (mod 8), a green value at x is positioned into the least-significant bit of the green value at x / 8, green values at x + 1 to x + 7 in order to the more significant bits of the green value at x / 8.

4 Image Data

Image data is an array of pixel values in scan-line order. We use image data in five different roles: The main role, an auxiliary role related to entropy coding, and three further roles related to transforms.

1. ARGB image.
2. Entropy image. The red and green components define the meta Huffman code used in a particular area of the image.
3. Predictor image. The green component defines which of the 14 values is used within a particular square of the image.
4. Color indexing image. An array of up to 256 ARGB colors are used for transforming a green-only image, using the green value as an index to this one-dimensional array.
5. Color transformation image. Defines signed 3.5 fixed-point multipliers that are used to predict the red, green, blue components to reduce entropy.

To divide the image into multiple regions, the image is first divided into a set of fixed-size blocks (typically 16x16 blocks). Each of these blocks can be modeled using an entropy code, in a way where several blocks can share the same entropy code. There is a cost in transmitting an entropy code, and in order to minimize this cost, statistically similar blocks can share an entropy code. The blocks sharing an entropy code can be found by clustering their statistical properties, or by repeatedly joining two randomly selected clusters when it reduces the overall amount of bits needed to encode the image. [See section “*Decoding of meta Huffman codes*” in Chapter 5 for an explanation of how this *entropy image* is stored.]

Each pixel is encoded using one of three possible methods:

1. Huffman coded literals, where each channel (green, alpha, red, blue) is entropy-coded independently,
2. LZ77, a sequence of pixels in scan-line order copied from elsewhere in the image, or,
3. color cache, using a short multiplicative hash code (color cache index) of a recently seen color.

In the following sections we introduce the main concepts in LZ77 prefix coding, LZ77 entropy coding, LZ77 distance mapping, and color cache codes. The actual details of the entropy code are described in more detail in chapter 5.

LZ77 prefix coding

Prefix coding divides large integer values into two parts, the prefix code and the extra bits. The benefit of this approach is that entropy coding is later used only for the prefix code, reducing the resources needed by the entropy code. The extra bits are stored as they are, without an entropy code.

This prefix code is used for coding backward reference lengths and distances. The extra bits form an integer that is added to the lower value of the range. Hence the LZ77 lengths and distances are divided into prefix codes and extra bits performing the Huffman coding only on the prefixes reduces the size of the Huffman codes to tens of values instead of otherwise a million (distance) or several thousands (length).

Prefix code	Value range	Extra bits
0	1	0
1	2	0
2	3	0
3	4	0
4	5..6	1
5	7..8	1
6	9..12	2
7	13..16	2
...	...	...
38	262145..524288	17
39	524289..1048576	17

The code to obtain a value from the prefix code is as follows:

```
if (prefix_code < 4) {
  return prefix_code;
}
int extra_bits = (prefix_code - 2) >> 1;
int offset = (2 + (prefix_code & 1)) << extra_bits;
return offset + ReadBits(extra_bits) + 1;
```

LZ77 backward reference entropy coding

Backward references are tuples of length and distance. Length indicates how many pixels in scan-line order are to be copied. The length is codified in two steps: prefix and extra bits. Only the first 24 prefix codes with their respective extra bits are used for length codes, limiting the maximum length to 4096. For distances, all 40 prefix codes are used.

LZ77 distance mapping

120 smallest distance codes [1..120] are reserved for a close neighborhood within the current pixel. The rest are pure distance codes in scan-line order, just offset by 120. The smallest codes are coded into x and y offsets by the following table. Each tuple shows the x and the y coordinates in 2d offsets — for example the first tuple (0, 1) means 0 for no difference in x, and 1 pixel difference in y (indicating previous row).

```
(0, 1), (1, 0), (1, 1), (-1, 1), (0, 2), (2, 0), (1, 2), (-1, 2),
(2, 1), (-2, 1), (2, 2), (-2, 2), (0, 3), (3, 0), (1, 3), (-1, 3),
(3, 1), (-3, 1), (2, 3), (-2, 3), (3, 2), (-3, 2), (0, 4), (4, 0),
(1, 4), (-1, 4), (4, 1), (-4, 1), (3, 3), (-3, 3), (2, 4), (-2, 4),
(4, 2), (-4, 2), (0, 5), (3, 4), (-3, 4), (4, 3), (-4, 3), (5, 0),
(1, 5), (-1, 5), (5, 1), (-5, 1), (2, 5), (-2, 5), (5, 2), (-5, 2),
(4, 4), (-4, 4), (3, 5), (-3, 5), (5, 3), (-5, 3), (0, 6), (6, 0), (1,
6), (-1, 6), (6, 1), (-6, 1), (2, 6), (-2, 6), (6, 2), (-6, 2), (4,
5), (-4, 5), (5, 4), (-5, 4), (3, 6), (-3, 6), (6, 3), (-6, 3), (0,
7), (7, 0), (1, 7), (-1, 7), (5, 5), (-5, 5), (7, 1), (-7, 1), (4, 6),
(-4, 6), (6, 4), (-6, 4), (2, 7), (-2, 7), (7, 2), (-7, 2), (3, 7), (-
3, 7), (7, 3), (-7, 3), (5, 6), (-5, 6), (6, 5), (-6, 5), (8, 0), (4,
7), (-4, 7), (7, 4), (-7, 4), (8, 1), (8, 2), (6, 6),
(-6, 6), (8, 3), (5, 7), (-5, 7), (7, 5), (-7, 5), (8, 4), (6, 7),
(-6, 7), (7, 6), (-7, 6), (8, 5), (7, 7), (-7, 7), (8, 6), (8, 7)
```

The distances codes that map into these tuples are changes into scan-line order distances using the following formula: *dist* = *x* + *y* * *xsize*, where *xsize* is the width of the image in pixels.

Color Cache Code

Color cache stores a set of colors that have been recently used in the image. Using the color cache code, the color cache colors can be referred more efficiently than emitting the respective ARGB values independently or by sending them as backward references with a length of one pixel.

Color cache codes are coded as follows. First, there is a bit that indicates if the color cache is used or not. If this bit is 0, no color cache codes exist, and they are not transmitted in the Huffman code that decodes the green symbols and the length prefix codes. However, if this bit is 1, the color cache size is read:

```
int color_cache_code_bits = ReadBits(br, 4);
int color_cache_size = 1 << color_cache_code_bits;
```

color_cache_code_bits defines the size of the color_cache by (*1* << *color_cache_code_bits*). The range of allowed values for *color_cache_code_bits* is [1..11]. Compliant decoders must indicate a corrupted bit stream for other values.

A color cache is an array of the size *color_cache_size*. Each entry stores one ARGB color. Colors are looked up by indexing them by (0x1e35a7bd * *color*) >> (32 - *color_cache_code_bits*). Only one lookup is done in a color cache, there is no conflict resolution.

In the beginning of decoding or encoding of an image, all entries in all color cache values are set to zero. The color cache code is converted to this color at decoding time. The state of the color cache is maintained by inserting every pixel, be it produced by backward referencing or as literals, into the cache in the order they appear in the stream.

5 Entropy Code

Huffman coding

Most of the data is coded using a canonical Huffman code. This includes the following:

- A combined code that defines either the value of the green component, a color cache code, or a prefix of the length codes,
- the data for alpha, red and blue components, and
- prefixes of the distance codes.

The Huffman codes are transmitted by sending the code lengths, the actual symbols are implicit and done in order for each length. The Huffman code lengths are run-length-encoded using three different prefixes, and the result of this coding is further Huffman coded.

Spatially-variant Huffman coding

For every pixel (x, y) in the image, there is a definition of which entropy code to use. First, there is an integer called 'meta Huffman code' that can be obtained from a subresolution 2d image. This meta Huffman code identifies a set of five Huffman codes, one for green (along with length codes and color cache codes), one for each of red, blue and alpha, and one for distance. The Huffman codes are identified by their position in a table by an integer.

Decoding flow of image data

Read next symbol S

1. S < 256
    a. Use S as green component
    b. read alpha
    c. read red
    d. read blue
2. S < 256 + 24
    a. Use S - 256 as a length prefix code
    b. read length extra bits
    c. read distance prefix code
    d. read distance extra bits
3. S >= 256 + 24
    a. Use ARGB color from the color cache, at index S - 256 + 24

Decoding the code lengths

There are two different ways to encode the code lengths of a Huffman code, indicated by the first bit of the code: *simple code length code* (1), and *normal code length code* (0).

Simple code length code

This variant can codify 1 or 2 non-zero length codes in the range of [0, 255]. All other code lengths are implicitly zeros.

The first bit indicates the number of codes:

```
int num_symbols = ReadBits(1) + 1;
```

The first symbol is stored either using a 1-bit code for values of 0 and 1, or using a 8-bit code for values in range [0, 255]. The second symbol, when present, is coded as an 8-bit code.

```
int first_symbol_len_code = VP8LReadBits(br, 1);
symbols[0] = ReadBits(1 + 7 * first_symbol_len_code);
if (num_symbols == 2) {
  symbols[1] = ReadBits(8);
}
```

Empty trees can be coded as trees that contain one 0 symbol, and can be codified using four bits. For example, a distance tree can be empty if there are no backward references. Similarly, alpha, red, and blue trees can be empty if all pixels within the same meta Huffman code are produced using the color cache.

Normal code length code

The code lengths of a Huffman code are read as follows. *num_codes* specifies the number of code lengths, the rest of the codes lengths (according to the order in *kCodeLengthCodeOrder*) are zeros.

```
int kCodeLengthCodes = 19;
int kCodeLengthCodeOrder[kCodeLengthCodes] = {
  17, 18, 0, 1, 2, 3, 4, 5, 16, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
int num_codes = 4 + ReadStream(4);
for (i = 0; i < num_codes; ++i) {
  code_lengths[kCodeLengthCodeOrder[i]] = ReadBits(3);
}
```

- Code length code [0..15] indicate literal code lengths.
    - Value 0 means no symbols have been coded,
    - values [1..15] indicate the bit length of the respective code.
- Code 16 repeats the previous non-zero value [3..6] times, i.e., 3 + ReadStream(2) times. If code 16 is used before a non-zero value has been emitted, a value of 8 is repeated.
- Code 17 emits a streak of zeros [3..10], i.e., 3 + ReadStream(3) times.
- Code 18 emits a streak of zeros of length [11..138], i.e., 11 + ReadStream(7) times.

The entropy codes for alpha, red and blue have a total of 256 symbols. The entropy code for distance prefix codes has 40 symbols. The entropy code for green has 256 + 24 + *color_cache_size*, 256 symbols for different green symbols, 24 length code prefix symbols, and symbols for the color cache.

The meta Huffman code, specified in the next section, defines how many Huffman codes there are. There are always 5 times the number of Huffman codes to the number of meta Huffman codes.

Decoding of meta Huffman codes

There are two ways to code the meta Huffman codes, indicated by one bit.

If this bit is zero, there is only one meta Huffman code, using Huffman codes 0, 1, 2, 3 and 4 for green, alpha, red, blue and distance, respectively. This meta Huffman code is used everywhere in the image.

If this bit is one, the meta Huffman codes are controlled by the entropy image, where the index of the meta Huffman code is codified in the red and green components. The index can be obtained from the uint32 value by *((pixel >> 8) & 0xffff),* thus there can be up to 65536 unique meta Huffman codes. When decoding a Huffman encoded symbol at a pixel x, y, one chooses the meta Huffman code respective to these coordinates. However, not all bits of the coordinates are used for choosing the meta Huffman code, i.e., the entropy image is of subresolution to the real image.

```
int huffman_bits = ReadBits(3) + 2;
int huffman_xsize = DIV_ROUND_UP(xsize, 1 << huffman_bits);
int huffman_ysize = DIV_ROUND_UP(ysize, 1 << huffman_bits);
```

huffman_bits gives the amount of subsampling in the entropy image.

After reading the *huffman_bits*, an entropy image stream of size huffman_xsize, huffman_ysize is read.

The meta Huffman code, identifying the five Huffman codes per meta Huffman code, is coded only by the number of codes:

```
int num_meta_codes = max(entropy_image) + 1;
```

Now, we can obtain the five Huffman codes for green, alpha, red, blue and distance for a given (x, y) by the following expression:

```
meta_codes[(entropy_image[(y >> huffman_bits) * huffman_xsize +
                          (x >> huffman_bits)] >> 8) & 0xffff]
```

The *huffman_code[5 * meta_code + k],* codes with *k* == 0 are for the green & length code, *k* == 4 for the distance code, and the codes at *k* == 1, 2, and 3, are for codes of length 256 for red, blue and alpha, respectively.

The value of *k* for the reference position in *meta_code* determines the length of the Huffman code:

- k = 0; length = 256 + 24 + cache_size
- k = 1, 2, or 3; length = 256
- k = 4, length = 40.

6 Overall Structure of the Format

Below there is a eagles-eye-view into the format in Backus-Naur form. It does not cover all details. End-of-image EOI is only implicitly coded into the number of pixels (xsize * ysize).

Basic structure

```
<format> ::= <RIFF header><image size><image stream>
<image stream> ::= (<optional-transform><image stream>) |
                   <entropy-coded image>
```

Structure of transforms

```
<optional-transform> ::= 1-bit <transform> <optional-transform> | 0-bit
<transform> ::= <predictor-tx> | <color-tx> | <subtract-green-tx> |
                <color-indexing-tx>
<predictor-tx> ::= 2-bit value 0; 4-bit sub-pixel code | <entropy-coded
image>
<color-tx> ::= 2-bit value 1; 4-bit sub-pixel code | <entropy-coded image>
<subtract-green-tx> ::= 2-bit value 2
<color-indexing-tx> ::= 2-bit value 3; 8-bit color count | <entropy-coded
image>
```

Structure of the image data

```
<entropy-coded image> ::= <color cache info><optional meta huffman><huffman
codes>
                          <lz77-coded image>
<optional meta huffman> ::= 1-bit value 0 |
                            (1-bit value 1;
                             <huffman image><meta Huffman size>)
<huffman image> ::= 4-bit subsample value; <image stream>
<meta huffman size> ::= 4-bit length; meta Huffman size (subtracted by 2).
<color cache info> ::= 1-bit value 0 |
                       (1-bit value 1; 4-bit value for color cache size)
<huffman codes> ::= <huffman code> | <huffman code><huffman codes>
<huffman code> ::= <simple huffman code> | <normal huffman code>
<simple huffman code> ::= see “Simple code length code” for details
<normal huffman code> ::= <code length code>; encoded code lengths
<code length code> ::= see section “Normal code length code”
<lz77-coded image> ::= (<argb-pixel> | <color-cache-code> | <lz77-copy>) |
                       (<lz77-coded image> | “”)
```

A possible example sequence

```
<RIFF header><image size>1-bit value 1<subtract-green-tx>
1-bit value 1<predictor-tx>1-bit value 0<huffman image>
<color cache info><meta huffman code><huffman codes>
<lz77-coded image>
```