CHALMERS

Design and Implementation of an MPEG-1 Layer III Audio Decoder

KRISTER LAGERSTRÖM

Master's Thesis
Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Gothenburg, Sweden 2001

Abstract

Digital compression of audio data is important due to the bandwidth and storage limitations inherent in networks and computers. Algorithms based on perceptual coding are effective and have become feasible with faster computers. The ISO standard 11172-3 MPEG-1 layer III (a.k.a. MP3) is a perceptual codec that is presently very common for compression of CD quality music. An MP3 decoder has a complex structure and is computationally demanding.

The purpose of this master's thesis is to present a tutorial on the standard. We have analysed several algorithms suitable for implementing an MP3 decoder, their advantages and disadvantages with respect to speed, memory demands and implementation complexity. We have also designed and implemented a portable reference MP3 decoder in C.

Preface

This thesis is part of the requirements for the Master of Science degree at Chalmers University of Technology in Gothenburg, Sweden. The work was done at UniData HB by Krister Lagerström. I wish to thank Professor Per Stenström at Chalmers University for his guidance and support throughout the work.

Table of contents

1 Introduction

1.1 Background

Digital compression of audio has become increasingly more important with the advent of fast and inexpensive microprocessors. It is used in many applications such as transmission of speech in the GSM mobile phone system, storing music in the DCC digital cassette format, and for the DAB digital broadcast radio.

Normally no information loss is acceptable when compressing digital data such as programs, source code, and text documents. Entropy coding is the method most commonly used for lossless compression. It exploits the fact that all bit combinations are not as likely to appear in the data, which is used in coding algorithms such as Huffman. This approach works for the data types mentioned above, however audio signals such as music and speech cannot be efficiently compressed with entropy coding.

When compressing speech and music signals it is not crucial to retain the input signal exactly. It is sufficient that the output signal *appears* to sound identical to a human listener. This is the method used in perceptual audio coders. A perceptual audio coder uses a psychoacoustic effect called 'auditory masking', where the parts of a signal that are not audible due to the function of the human auditory system are reduced in accuracy or removed completely.

The international standard ISO 11172-3 ([2]) defines three different methods of increasing complexity and compression efficiency for perceptual coding of generic audio such as speech and music signals. This thesis deals exclusively with the third method, also known as MP3. It has become very popular for compressing CD quality music with almost no audible degradation down from 1.4 Mbit/s to 128 kbit/s. This means that an ISDN connection can be used for real-time transmission and that a full-length song can be stored in 3-4 Mbytes.

An MP3 decoder is relatively complex and CPU intensive. A commercial implementation must therefore be carefully designed in order to be cost-effective. This thesis is intended to serve both as a tutorial on the standard and as a reference model of an implementation.

The target audience of this document is mainly design engineers who need to write an MP3 decoder, e.g. for an embedded DSP.

1.2 Problem Statement

The MP3 standard is clearly very efficient for digital audio compression and it has subsequently become very successful in the marketplace. One reason for its popularity is that it is an international standard [2].

Unfortunately the standard is poorly written, as it is ambiguous in some places and lacks details in others. It also does not place any emphasis on computation efficiency. Writing a new MP3 decoder is therefore a greater task than might otherwise be expected.

The goals for this thesis have been the following:

One goal has been to compile an introduction to the subject of MP3 encoding and decoding as well as psychoacoustics. There exists a number of studies of various parts of the decoder, but complete treatments on a technical level are not as common. We have used material from papers, journals, and conference proceedings that best describe the various parts.

Another goal has been to search for algorithms that can be used to implement the most demanding components of an MP3 decoder:

- Huffman decoding of samples
- Requantization of samples
- Inverse Modified Cosine Transform (IMDCT)
- Polyphase filterbank

A third goal is to evaluate their performance with regard to speed, memory requirements, and complexity. These properties were chosen because they have the greatest impact on the implementation effort and the computation demands for MP3 decoding.

The Huffman decoding of samples deals with variable length decoding of samples from the bitstream. The other three parts all deal with various mathematical transforms of the samples that are specified by the standard.

A final goal has been to design and implement an MP3 decoder. This should be done in in C for Unix. The source code should be easy to understand so that it can serve as a reference on the standard for designers that need to implement a decoder.

1.3 Methods Used

The algorithms were evaluated using mathematical analysis and computer simulation. The sample requantization algorithm was constructed using Newton's method, and the accuracy was determined though a computer program. The IMDCT and filterbank algorithms were designed by using Lee's method of performing fast discrete cosine transforms (DCT). Both the design and evaluation of the Huffman decoding algorithm were obtained from [4].

1.4 Results

We have compiled an introduction to the subject of MP3 decoders from existing sources. The introduction relates the decoder to the encoder, as well as giving a background on perceptual audio coding. The decoder is also described in some depth.

The search for efficient algorithms has been successful and we give several examples of algorithms that are significantly better than those described by the standard.

The Huffman decoder was found to be best implemented using a combination of binary trees and table lookups.

For the requantization part we did not find any really fast algorithms for the calculations, instead a table lookup method was found to best in the general case. An algorithmic approach is also described.

Both the IMDCT and the polyphase filterbank has been shown to be best computed using fast DCT algorithms.

We also implemented the reference decoder in C as planned. We hope that it will be useful as a definite guide on the unclear parts of the standard.

1.5 Thesis Organization

In Chapter 2 we give an overview of the MP3 standard and the structure of both an encoder and a decoder, along with a review of perceptual audio coding.

Chapter 3 describes algorithms suitable for implementing different parts of an MP3 decoder.

Chapter 4 contains an evaluation of the algorithms in chapter 3.

Chapter 5 describes the MP3 decoder reference design.

Finally, Chapter 6 concludes the thesis by providing a summary of the results of this work and identifying directions for future work.

Appendix A explains abbreviations and terms used in this thesis.

Appendix B contains the source code of the reference MP3 decoder and the computer simulation models.

2 Overview of the MP3 Standard

2.1 Background

The MPEG-1 standard was developed by the Motion Pictures Expert Group (MPEG) within the International Organization of Standardization (ISO). It covers audio and video coding at bit rates around 1-2 Mbit/s. The standard defines three different systems, or layers, for coding of audio data. The third layer is the most efficient and has become a de facto standard for coding of near-CD quality audio. This thesis deals exclusively with layer III of the MPEG-1 standard, also known as MP3.

The related MPEG-2 standard is also intended for coding of audio-visual data, but at higher bitrates (5-10 Mbit/s) than MPEG-1. It introduced lower sampling rates, a backwards compatible multichannel mode and a non-backwards compatible audio coder (AAC) with a higher compression effiency than MPEG-1.

An important point for both the MPEG-1 and MPEG-2 standards is that they only describe the decoder and the meaning of the encoded bitstream. However, the encoder is not standardized, thus leaving room for evolutionary improvements. The decoder can also be more or less efficiently implemented, depending on the choice of algorithms.

The latest addition to the MPEG family of standards is MPEG-4 which offers higher compression rates through more advanced coding [7], [8]. It also supports a wider spectrum of applications ranging from low bit rate speech coding to high-fidelity audio systems. MPEG-3 was never made a standard.

This chapter includes material from [1], [3], [4], [5] and [6] which all contain good presentations of one or more aspects of perceptual audio coding, MPEG audio encoding and decoding.

2.2 A Brief Review of Perceptual Audio Coding

The MPEG audio coding standard is based on perceptual audio coding principles. This section gives an introduction to the subject. It is based mainly on [3] which contains a detailed study of perceptual audio coding as it is used for MPEG audio coding and other standards, and also on [1] which contains a comprehensive introduction.

2.2.1 Generic Perceptual Audio Coding Architecture

A *lossless* or *noiseless* coder is able to perfectly reconstruct the original signal. A *lossy* coder on the other hand is incapable of perfect reconstruction. Lossy coding has the advantage of lower bit rates compared to lossless coding.

The lossy compression scheme described here achieves coding gain by exploiting both perceptual irrelevancies and statistical redundancies. Most perceptual audio coders follow the general outline of figure 1 below.

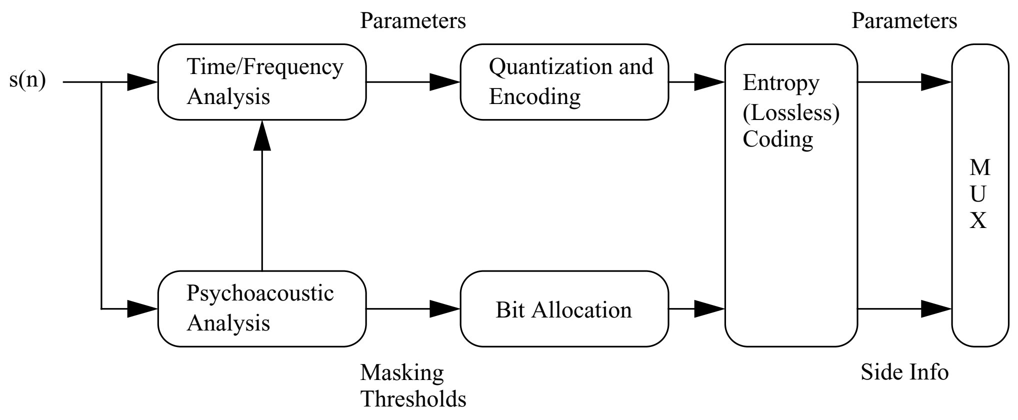

FIGURE 1. Generic perceptual audio coder (Source: [3]).

The coder segments the input *s(n)* into quasistationary frames ranging from 2 to 50 ms in duration. Then a time-frequency analysis block estimates the temporal and spectral components of each frame. These components are mapped to the analysis properties of the human auditory system and the time-frequency parameters suitable for quantization and encoding are extracted.

The psychoacoustic block allows the quantization and encoding block to exploit perceptual irrelevancies in the time-frequency parameter set. The remaining redundancies are typically removed through lossless entropy coding techniques.

2.2.2 Psychoacoustic Principles

The field of psychoacoustics deals with characterizing human auditory perception, in particular the time-frequency analysis capabilities of the inner ear. Most current audio coders achieve compression by exploiting the fact that irrelevant signal information is not detectable even by a sensitive listener.

The inner ear performs short-term critical band analyses where frequency-to-place transformations occur along the basilar membrane [1]. The power spectra are not represented on a linear frequency scale but on limited frequency bands called *critical bands*.

The auditory system can roughly be described as a bandpass filterbank, consisting of strongly overlapping bandpass filters with bandwidths in the order of 50 to 100 Hz for signals below 500 Hz and up to 5000 Hz for signals at high frequencies. Twenty-six critical bands covering frequencies of up to 24 kHz have to be taken into account.

Simultaneous masking is a frequency domain phenomenon where a low-level signal (the maskee) can be made inaudible (masked) by a simultaneously occurring stronger signal (the masker) as long as masker and maskee are close enough to each other in frequency. Such masking is largest in the critical band in which the masker is located, and it is effective to a lesser degree in neighboring bands.

In addition to simultaneous masking, the time-domain phenomenon of *temporal masking* plays an important role in human auditory perception. It may occur when two sounds appear within a small interval of time. Depending on the signal levels, the stronger sound may mask the weaker one, even if the maskee precedes the masker. The duration within which *premasking* applies is significantly less than that of the *postmasking* which is in the order of 50 to 200 ms.

A *masking threshold* can be measured and low-level signals below this threshold will not be audible, as illustrated by figure 2 below. This masked signal can consist of low-level signal contributions, of quantization noise, aliasing distortion, or of transmission errors. The threshold will vary with time and depend on the sound pressure level, the frequency of the masker, and on characteristics of masker and maskee (e.g. noise is a better masker than a tone).

Without a masker, a signal is inaudible if its sound pressure level is below the *threshold in quiet.* This depends on the frequency and covers a dynamic range of more than 60 dB, as shown by the lower curve of figure 2 below.

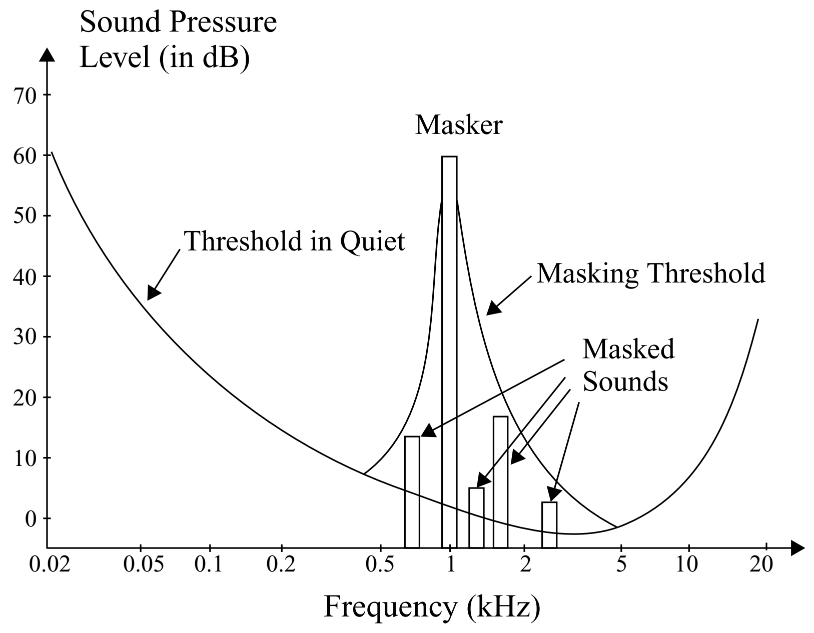

FIGURE 2. Threshold in quiet and masking threshold (acoustical events under the masking thresholds will not be audible). (Source: [1]).

2.3 MP3 Encoding

2.3.1 Encoder Structure

The encoder has the following structure:

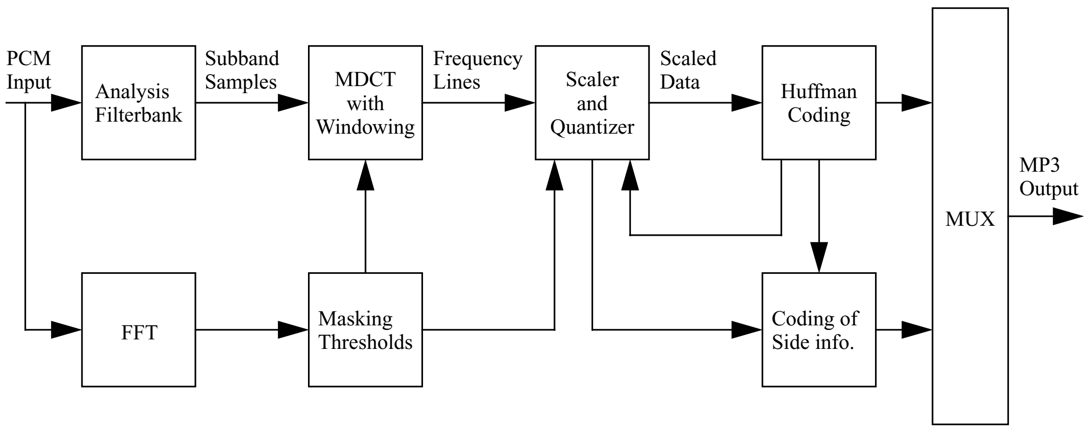

FIGURE 3. MP3 Encoder Structure (Source: [1]).

The input to the encoder is normally PCM coded data that is split into frames of 1152 samples. The frames are further divided into two granules of 576 samples each. The frames are sent to both the Fast Fourier Transform (FFT) block and the analysis filterbank.

2.3.2 FFT Analysis

The FFT block transforms granules of 576 samples to the frequency domain using a Fourier transform.

2.3.3 Masking Thresholds

The frequency information from the FFT block is used together with a psychoacoustic model to determine the masking thresholds for all frequencies. The masking thresholds are applied by the quantizer to determine how many bits are needed to encode each sample. They are also used to determine if window switching is needed in the MDCT block.

2.3.4 Analysis Filterbank

The analysis filterbank consists of 32 bandpass filters of equal width. The output of the filters are critically sampled. That means that for each granule of 576 samples there are 18 samples output from each of the 32 bandpass filters, which gives a total of 576 subband samples.

2.3.5 MDCT with Dynamic Windowing

The subband samples are transformed to the frequency domain through a modified discrete cosine transform (MDCT). The MDCT is performed on blocks that are windowed and overlapped 50%.

The MDCT is normally performed for 18 samples at a time (long blocks) to achieve good frequency resolution. It can also be performed on 6 samples at a time (short blocks) to achieve better time resolution, and to minimize pre-echoes. There are special window types for the transition between long and short blocks.

2.3.6 Scaling and Quantization

The masking thresholds are used to iteratively determine how many bits are needed in each critical band to code the samples so that the quantization noise is not audible. The encoder usually also has to meet a fixed bitrate requirement.

The Huffman coding is part of the iteration since it is not otherwise possible to determine the number of bits needed for the encoding.

2.3.7 Huffman Coding and Bitstream Generation

The quantized samples are Huffman coded and stored in the bitstream along with the scale factors and side information.

2.3.8 Side Information

The side information contains parameters that control the operation of the decoder, such as Huffman table selection, window switching and gain control.

2.4 MP3 Decoding

2.4.1 Decoder Structure

The decoder has the following structure:

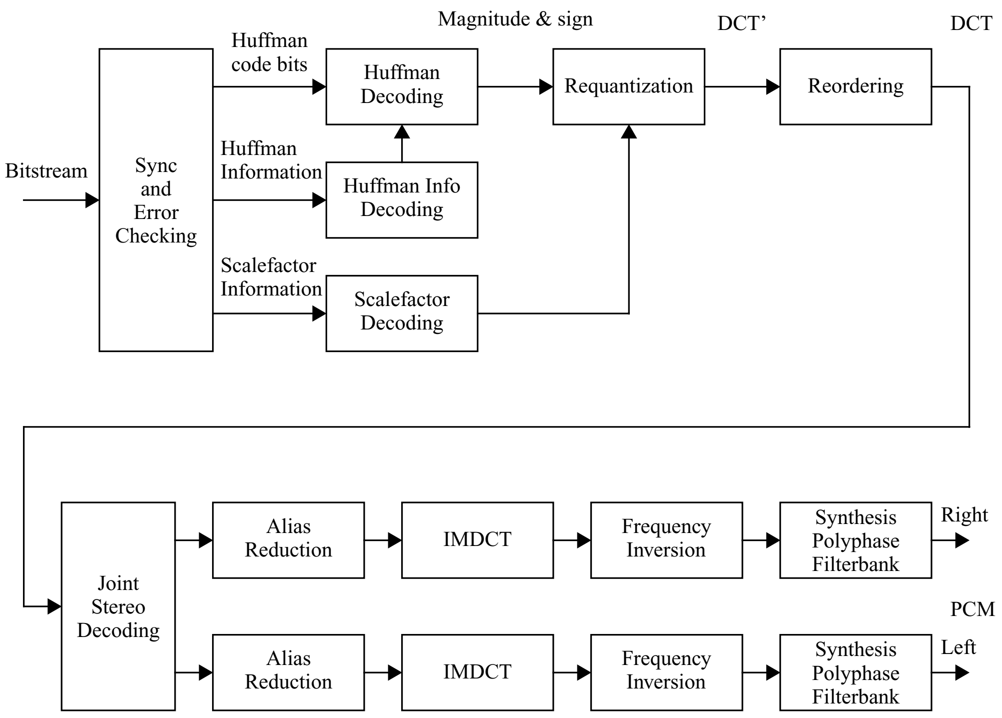

FIGURE 4. MP3 decoder structure (Source: [4]).

The different parts of the decoder are described in more detail below.

2.4.2 Frame Format

The frame is a central concept when decoding MP3 bitstreams. It consists of 1152 mono or stereo frequency-domain samples, divided into two granules of 576 samples each. Each granule is further divided into 32 subband blocks of 18 frequency lines apiece:

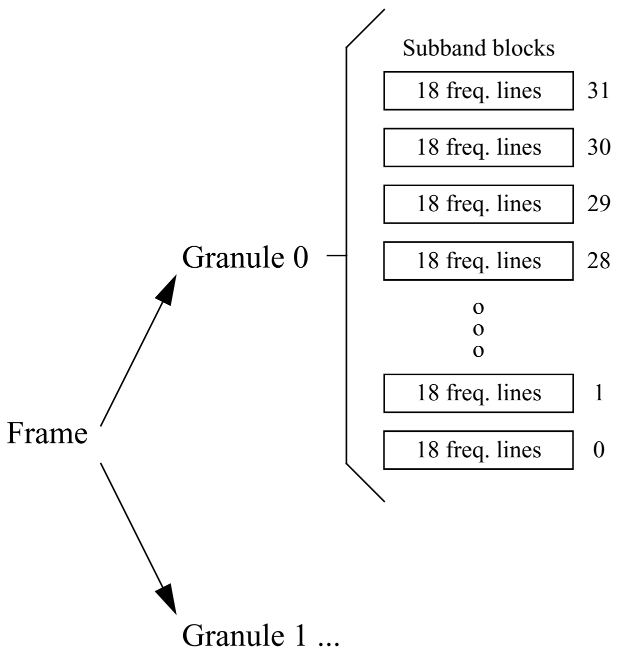

FIGURE 5. Format of MP3 frame, granules, subband blocks and frequency lines.

The frequency spectrum ranges from 0 to $F_S/2$ Hz. The subbands divide the spectrum into 32 equal parts. The subbands each contain 18 samples that have been transformed to the frequency domain by a modified discrete cosine transform (MDCT).

The 576 frequency lines in a granule are also divided into 21 scalefactor bands that have been designed to match the critical band frequencies as closely as possible. The scalefactor bands are used primarily for the requantization of the samples.

The frame consists of four parts: header, side information, main data, and ancillary data:

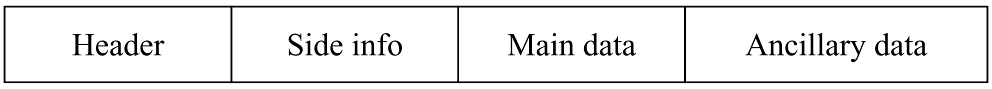

FIGURE 6. MP3 frame format.

The length of a frame is constant for a fixed bitrate, with the possible deviation of one byte to maintain an exact bitrate. There is also a variable bitrate format where the frame lengths can vary according to the momentaneous demands of the encoder.

The main data (scalefactors and Huffman coded data) are not necessarily located adjacent to the side information, as shown in figure 7 below.

2.4.2.1 Header

The header is always 4 bytes long and contains information about the layer, bitrate, sampling frequency and stereo mode. It also contains a 12-bit syncword that is used to find the start of a frame in a bitstream, e.g. for broadcasting applications.

2.4.2.2 Side Information

The side information section contains the necessary information to decode the main data, such as Huffman table selection, scale factors, requantization parameters and window selection.

This section is 17 bytes long in single channel mode and 32 bytes in dual channel mode.

2.4.2.3 Main Data

The main data section contains the coded scale factor values and the Huffman coded frequency lines ("main data"). The length depends on the bitrate and the length of the ancillary data.

The length of the scale factor part depends on whether scale factors are reused, and also on the window length (short or long). The scalefactors are used in the requantization of the samples, see section 2.4.4 for details.

The demand for Huffman code bits varies with time during the coding process. The variable bitrate format can be used to handle this, but a fixed bitrate is often a requirement for an application (e.g. for broadcasting). Therefore there is also a bit reservoir technique defined that allows unused main data storage in one frame to be used by up to two consecutive frames:

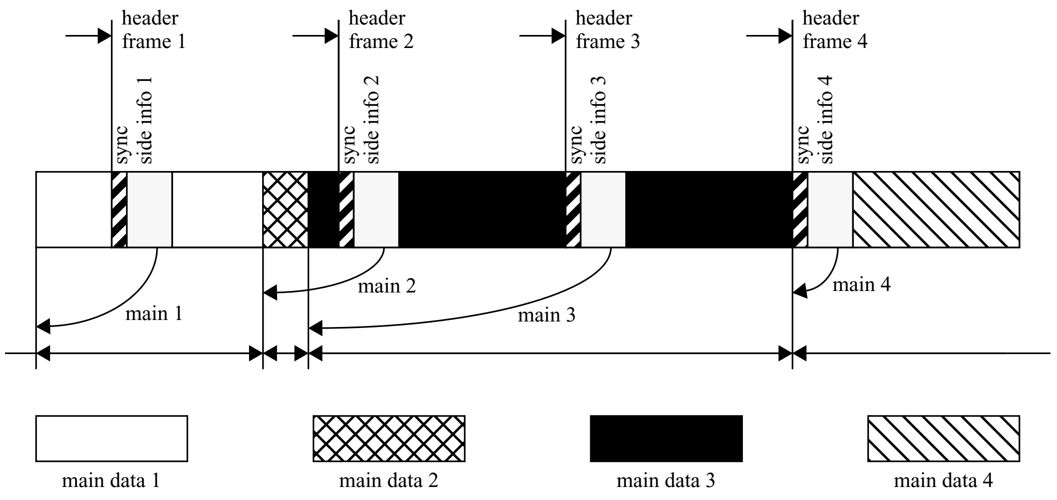

FIGURE 7. Main data buffer handling.

In this example frame 1 uses bits from frame 0 and 1. Frame 2 uses bits from frame 1. Frame 3 that has a high demand uses bits from frames 1, 2 and 3. Finally, frame 4 uses bits only from frame 4.

The “main_data_begin” parameter in the side information indicates whether bits from previous frames are needed. All the main data for one frame is stored in that and previous frames. The maximum size of the bit reservoir is 511 bytes.

2.4.2.4 Ancillary Data

This section is intended for user-defined data and is not specified further in the standard. It is not needed to decode the audio data.

2.4.3 Huffman Decoding

The Huffman data section contains the variable-length encoded samples. The Huffman coding scheme assumes that the large values occurs at the low spectral frequencies and mainly low values and zeroes occur at the high spectral frequencies. Therefore, the 576 spectral lines of each granule are partitioned into five regions as illustrated in this figure:

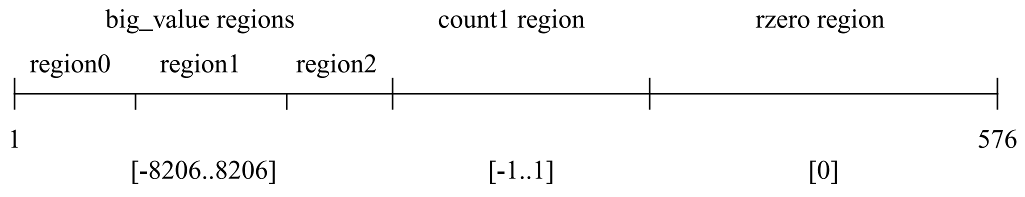

FIGURE 8. Huffman partitions.

The “rzero” region contains only zero values, while the “count1” region contains small values ranging from -1 to 1 and the “big_value” region contains values from -8206 to 8206.

Different Huffman code tables are used depending on the maximum quantized value and the local statistics of the signal. There are a total of 32 possible tables given in the standard. Each of the four regions in big_value and count1 can use a different Huffman table for decoding.

The count1 parameter that indicates the number of frequency lines in the count1 region is not explicitly coded in the bitstream. The end of the count1 region is known only when all bits for the granule (as specified by part2_3_length) have been exhausted, and the value of count1 is known implicitly after decoding the count1 region.

Chapter 3.2 contains a survey of different ways to implement the Huffman decoder.

2.4.4 Requantization

The sample requantization block uses the scale factors to convert the Huffman decoded values is_i back to their spectral values xr_i using the following formula:

$$xr_i = is_i^{\frac{4}{3}} * 2^{(0.25*C)}$$

Requantization of samples. (EQ 1)

The factor "*C*" in the equation consists of global and scalefactor band dependent gain factors from the side information and the scale factors.

Chapter 3.3 contains a survey of different ways to implement the sample requantization.

2.4.5 Reordering

The requantized samples must be reordered for the scalefactor bands that use short windows. In this example there are a total of 18 samples in a band that contains 3 windows of 6 samples each:

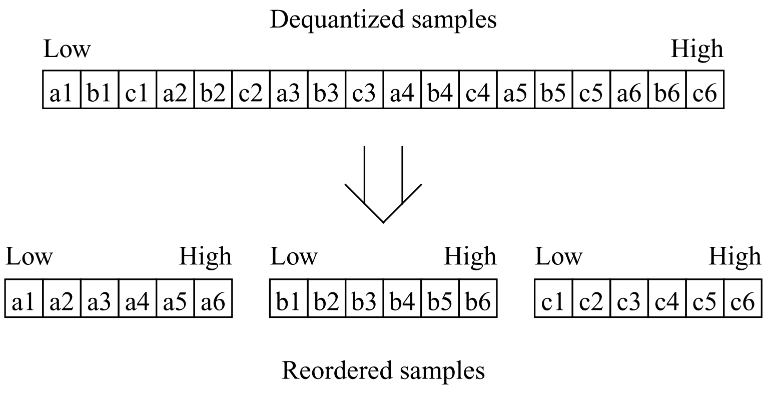

FIGURE 9. Reordering of samples.

The short windows are reordered in the encoder to make the Huffman coding more efficient, since the samples close in frequency (low or high) are more likely to have similar values.

2.4.6 Stereo Decoding

The compressed bitstream can support one or two audio channels in one of four possible modes [5]:

1. a monophonic mode for a single audio channel,
2. a dual-monophonic mode for two independent audio channels (functionally identical to the stereo mode),
3. a stereo mode for stereo channels that share bits but do not use joint-stereo coding, and

4. a joint-stereo mode that takes advantage of either the correlations between the stereo channels (MS stereo) or the irrelevancy of the phase difference between channels (intensity stereo), or both.

The stereo processing is controlled by the mode and mode_extension fields in the frame header.

2.4.6.1 MS Stereo Decoding

In the MS stereo mode the left and right channels are transmitted as the sum (M) and difference (S) of the two channels, respectively. This mode is suitable when the two channels are highly correlated, which means that the sum signal will contain much more information than the difference signal.

The stereo signal can therefore be compressed more efficiently compared to transmitting the two stereo channels independently of each other. In the decoder the left and right channels can be reconstructed using the following equation, where *i* is the frequency line index:

$$L_i = \frac{M_i + S_i}{\sqrt{2}} \quad \text{and} \quad R_i = \frac{M_i - S_i}{\sqrt{2}}$$

MS Stereo Decoding. (EQ 2)

The MS stereo processing is lossless.

2.4.6.2 Intensity Stereo Decoding

In intensity stereo mode the encoder codes some upper-frequency subband outputs with a single sum signal L+R instead of sending independent left (L) and right (R) subband signals. The balance between left and right is transmitted instead of scalefactors.

The decoder reconstructs the left and right channels based only on the single L+R ($=L'_i$) signal which is transmitted in the left channel and the balance which is transmitted instead of scalefactors ($ispos_{sfb}$) for the right channel:

$$isratio_{sfb} = \tan\left[ispos_{sfb}\frac{\pi}{12}\right]$$

$$L_i = L'_i\frac{isratio_{sfb}}{1 + isratio_{sfb}} \quad \text{and} \quad R_i = L'_i\frac{1}{1 + isratio_{sfb}}$$

Intensity Stereo Decoding. (EQ 3)

The $ispos_{sfb}$ parameter is limited to values between 0 and 6, so the tan() function is easily replaced by a small lookup table.

2.4.7 Alias Reduction

The alias reduction is required to negate the aliasing effects of the polyphase filterbank in the encoder. It is not applied to granules that use short blocks.

The alias reduction consists of eight butterfly calculations for each subband as illustrated by the figure below.

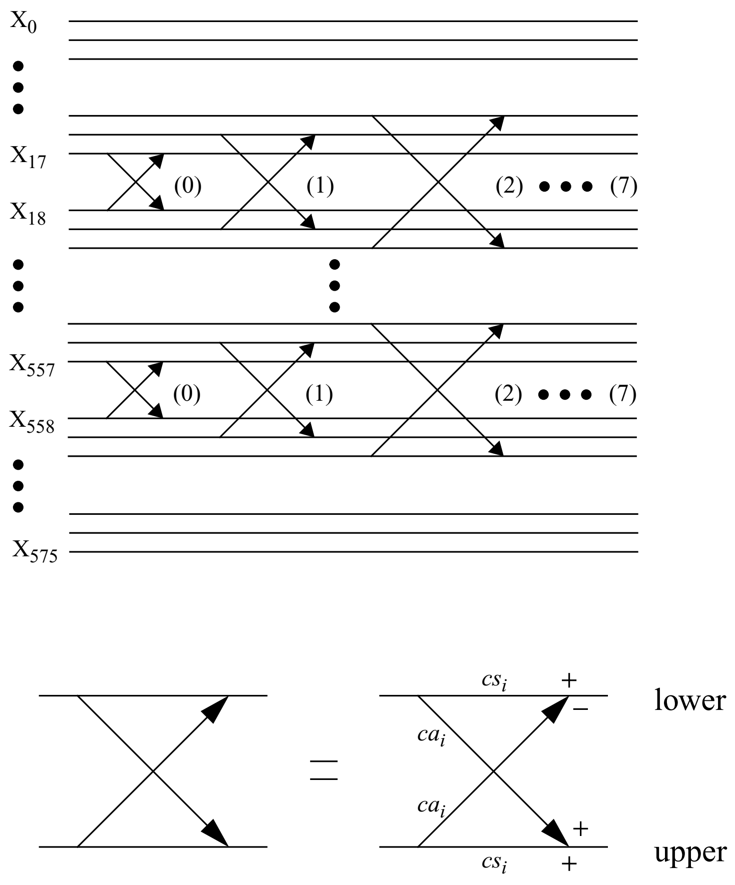

FIGURE 10. Alias reduction butterflies (Source: [4]). The *cs_i* and *ca_i* constants are tabulated in [2].

2.4.8 IMDCT

The IMDCT (Inverse Modified Discrete Cosine Transform) transforms the frequency lines (X_k) to polyphase filter subband samples (S_i). The analytical expression of the IMDCT is shown below, where n is 12 for short blocks and 36 for long blocks:

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left[\frac{\pi}{2n}\left[2i+1+\frac{n}{2}\right](2k+1)\right], \text{ for i=0 to n-1}$$

IMDCT Transform. (EQ 4)

In case of long blocks the IMDCT generates an output of 36 values for every 18 input values. The output is windowed depending on the block type (start, normal, stop) and the first half is overlapped with the second half of the previously saved block.

In case of short blocks three transforms are performed which produce 12 output values each. The three vectors are windowed and overlapped with each other. Concatenating 6 zeros on both ends of the resulting vector gives a vector of length 36, which is processed like the output of a long transform.

The overlapped addition operation is illustrated by the following figure:

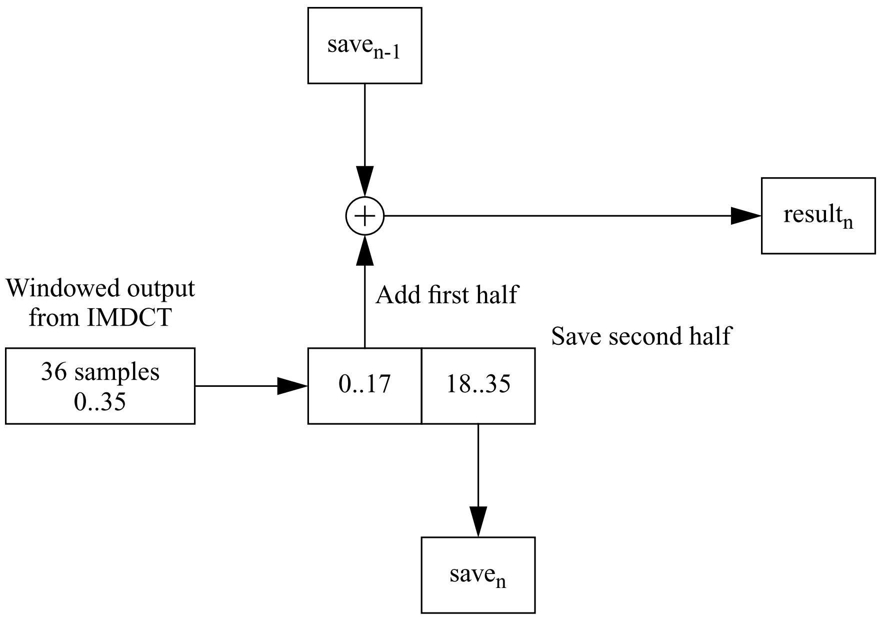

FIGURE 11. Overlapped add operation.

The output from the IMDCT operation is 18 time-domain samples for each of the 32 subband blocks.

Chapter 3.4 contains a survey of different ways to implement the IMDCT transform.

N.B.: It is important to clear the overlapped addition buffers when performing random seeking in the decoder in order to avoid noise in the output.

2.4.9 Frequency Inversion

In order to compensate for frequency inversions in the synthesis polyphase filterbank every odd time sample of every odd subband is multiplied with -1.

The subbands are numbered [0..31] and the time samples in each subband [0..17].

2.4.10 Synthesis Polyphase Filterbank

The synthesis polyphase filterbank transforms the 32 subband blocks of 18 time-domain samples in each granule to 18 blocks of 32 PCM samples. The filterbank operates on 32 samples at a time, one from each subband block, as illustrated by the following figure:

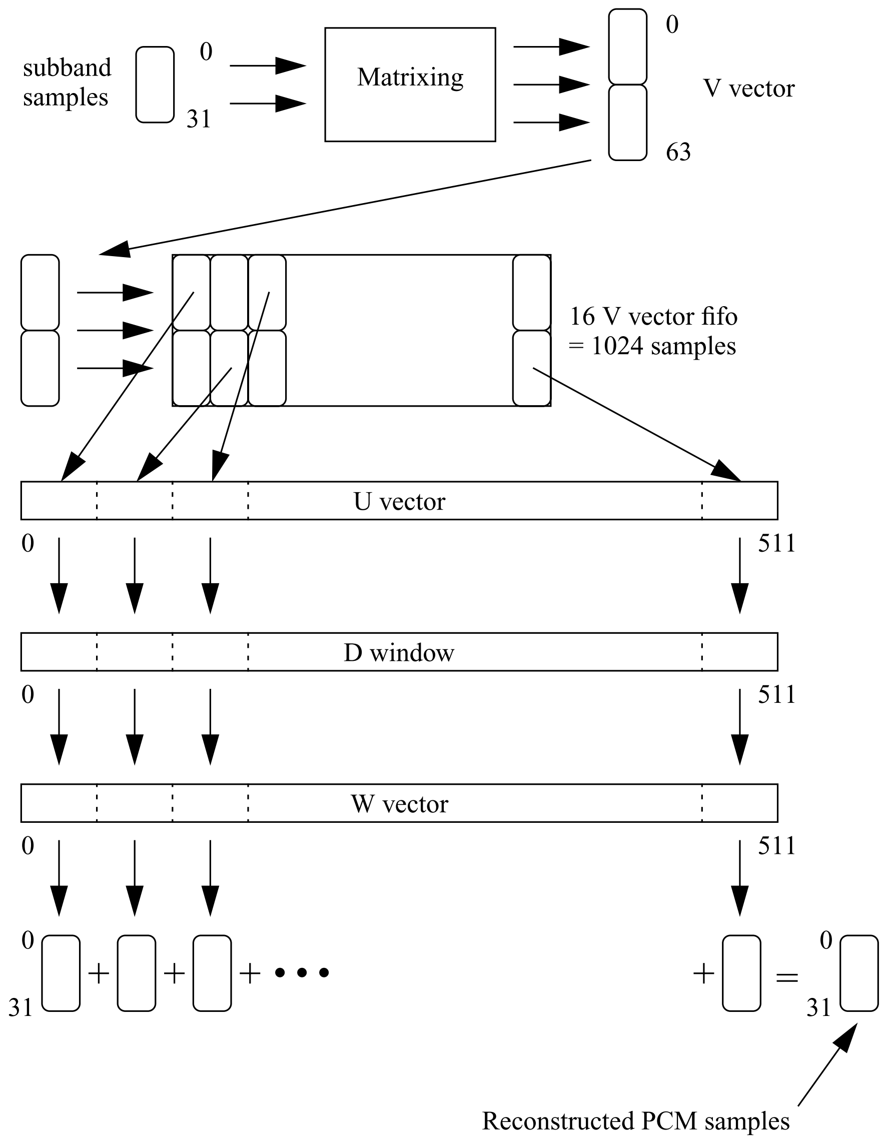

FIGURE 12. Synthesis polyphase filterbank (Source: [6]).

In the synthesis operation, the 32 subband values are transformed to the 64 value V vector using a variant of the IMDCT (matrixing). The V vector is pushed into a fifo which stores the last 16 V vectors.

A U vector is created from the alternate 32 component blocks in the fifo as illustrated and a window function D is applied to U to produce the W vector. The reconstructed samples are obtained from the W vector by decomposing it into 16 vectors each 32 values in size and summing these vectors.

Chapter 3.5 contains a survey of different ways to implement the filterbank.

N.B.: The vector V has to be cleared at the start of each song, or when performing random seeking in the bitstream.

3 Survey of Efficient MP3 Decoding Algorithms

3.1 Introduction

The standard ([2]) defines how the bitstream should be interpreted as well as the transformations needed to produce the PCM samples. For some parts (e.g. Huffman decoding) it lacks the necessary details on the transformation, and for other parts it does not use the most efficient algorithm possible. It might also be the case that the designer needs to place the emphasis on one aspect, such as memory requirements.

This chapter contains a survey of different ways to implement the parts of the decoder that are especially demanding.

We only look at general optimizations in this thesis. There are a number of things that should be taken into consideration for an optimal implementation, such as cache utilization, pipeline stalls, register starvation, etc. Depending on the processor architecture there might be features present that can be used for higher effiency, such as address generators, direct memory access, separate data and program memories, etc.

3.2 Huffman Decoding

3.2.1 Definition

The Huffman decoder translates the variable length codes in the bitstream to spectral lines. The decoder uses 32 fixed tables from the standard ([2]) that contain information about how the codes are designed.

The bitstream side information is used to select the tables for the different parts of the frequency spectrum.

There is a special mechanism in the decoder for decoding large values. Certain decoded values imply that a table dependent number of linear bits must be read and added to the value. Sign bits also have special handling.

The longest variable length codeword for any table is 19 bits. There are only 16 different tables actually defined by the standard.

3.2.2 Implementation Issues

A significant part of the processing will probably lie in the bitstream handling. We will not explore that further since it will probably be hand-coded in assembler for the particular CPU that is used.

3.2.3 Binary Tree Search

The Huffman decoder tables can be translated into binary trees. Each tree then represents a certain table.

The trees are traversed according to the bits in the bitstream, where a ‘0’ might mean go ‘left’ and a ‘1’ go ‘right’.

An entire code-word is fully decoded when a leaf is encountered. The leaves contains the values for the spectral lines.

3.2.4 Direct Table Lookup

For the direct table lookup method the decoder uses large tables. The length of each table is 2^b, where b is the maximum number of bits in the longest code-word for that table.

To decode a code-word, the decoder reads ‘b’ bits. The bits are used as a direct index into the table, where each entry contains the spectral line values and information about the real length of the code-word. The surplus bits must then be re-used for the next code-word.

3.2.5 Clustered Decoding

The clustered decoding method combines the binary tree and direct table methods.

A fixed number of bits (e.g. 4) is read from the bitstream and used as a lookup index into a table. Each table element contains a hit/miss bit that indicates whether the code-word has been fully decoded yet.

If a hit is detected the symbol is read from the table element as well as the number of bits that is used for the code-word. If it is a miss the decoding continues by using the information from the table element to determine how many more bits to read from the bitstream for the next index, as well as the starting address of the next table to use.

3.3 Requantizer

3.3.1 Definition

The requantization formula describes the processing to rescale the Huffman coded data. Global gain and subblock gain affect all values within one time window. Scalefactors and preflag further adjust the gain within each scalefactor band.

The following is the requantization equation for short windows. The Huffman decoded value at buffer index *i* is called is_i, the input to the reordering block at index *i* is called xr_i:

$$xr_i = sign(is_i) * |is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}A} * 2^{-B}$$

$$A = global\_gain[gr] - 210 - (8) * subblock\_gain[window][gr]$$

$$B = scalefac\_multiplier * scalefac\_s[gr][ch][sfb][window]$$

Requantization of samples, short blocks. (EQ 5)

For long blocks, the formula is:

$$xr_i = sign(is_i)*|is_i|^{\frac{4}{3}}*2^{\frac{1}{4}A}*2^{-B}$$

$$A = global\_gain[gr] - 210$$

$$B = scalefac\_multiplier*scalefac\_l[gr][ch][sfb] + preflag[gr]*pretab[sfb]$$

Requantization of samples, long blocks. (EQ 6)

Pretab[sfb] is tabulated in the standard. It is used to amplify high-frequency scalefactor bands. The value 210 is a system constant that is needed to scale the samples.

It is important to note that the maximum value of is_i is 8206, not 8191 as stated by the standard [2]. The reason for this is that the Huffman decoder adds 15 to the value of linbits. Linbits can be 13 bits long which gives a maximum value of 2^{13}-1 = 8191 for the linbits part alone.

3.3.2 Implementation Issues

Both the $is_i^{4/3}$ and the $2^{A,B}$ power functions are computationally expensive to implement using the standard math library function *pow()*. This is true even if it is calculated using a FPU or DSP.

The $is_i^{4/3}$ function in the requantizer can assume 8207 different values. A lookup table is fast, but would require approximately 256 kbits of memory. We therefore also look at an algorithmic approach in section 3.3.4 below.

The function $2^{0.25*A}*2^{-B}$ does not assume more than 384 different values. That means that a lookup table is probably the best choice even for memory-constrained implementations. It can be noted that the table can made even smaller (196 values) by rounding small values ($< 2^{-35}$) for the function down to zero since that will not affect the end result.

3.3.3 Table-based Approach for $y=x^{4/3}$

A lookup table for the $y=x^{4/3}$ function is easy to implement. The table could be included as part of the initialized data section, or it might be generated at run-time if the *pow()* function is available.

If there is enough memory the lookup table could include the negative values of is_i as well. This will speed up the decoding.

3.3.4 Newton's Method for $y=x^{4/3}$

The $y = x^{4/3}$ function can be rewritten as $y^3 - x^4 = 0$. This form is suitable for Newton's method of root-finding which will yield a value of y that approximates $x^{4/3}$.

The function result is calculated through repeated iterations that successively reduces the residual error $| y - x^{4/3} |$:

$$y_{n+1} = y_n - \frac{y_n^3 - x^4}{3y_n^2} = \frac{2y_n^3 + x^4}{3y_n^2}$$

Iteration formula for $y=x^{4/3}$. (EQ 7)

The formula is rewritten as the second form to avoid floating-point cancellation.

The starting value y_0 for the iteration formula affects the number for iterations needed to achieve the desired accuracy. For this application an accuracy larger than 16 bits is sufficient. A good starting value for y_0 is calculated by the polynomial fit function $y_0=a_0+a_1*x+a_2*x^2$. This function is designed to resemble $y = x^{4/3}$ as closely as possible for $0<x<8207$. The starting value will yield the desired accuracy in 3 iterations.

3.4 Inverse Modified Discrete Cosine Transform (IMDCT)

3.4.1 Definition

The IMDCT operation transforms the subband samples in a granule from the frequency domain to the time domain. The analytical expression of the IMDCT is:

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left[\frac{\pi}{2n}\left[2i + 1 + \frac{n}{2}\right](2k+1)\right], \text{ for i=0 to n-1}$$

IMDCT Transform. (EQ 8)

The value of n in the expression can be either 12 for short blocks or 36 for long blocks.

The output from the IMDCT must be windowed with a fixed function and overlapped with the data from the previous block.

3.4.2 Implementation Issues

The IMDCT operation is very computationally expensive to implement as it is defined by the standard. A lookup table can be used to replace the *cos()* function, but the inner loop of the equation will still require substantial processing. We therefore investigate faster algorithms for the IMDCT below.

The window functions can be replaced by a 4 kbit lookup table.

3.4.3 Direct Calculation

A direct calculation of the IMDCT operation is easy to implement since it only consists of two simple nested for-loops. A lookup table can be used to replace the *cos()* function call in the inner loop.

3.4.4 Fast IMDCT Implementation

Marovich has shown in [17] that Konstantinides' method ([16]) of accelerating the polyphase filterbank matrixing operation can also be applied to the 12- and 36-point IMDCT operations:

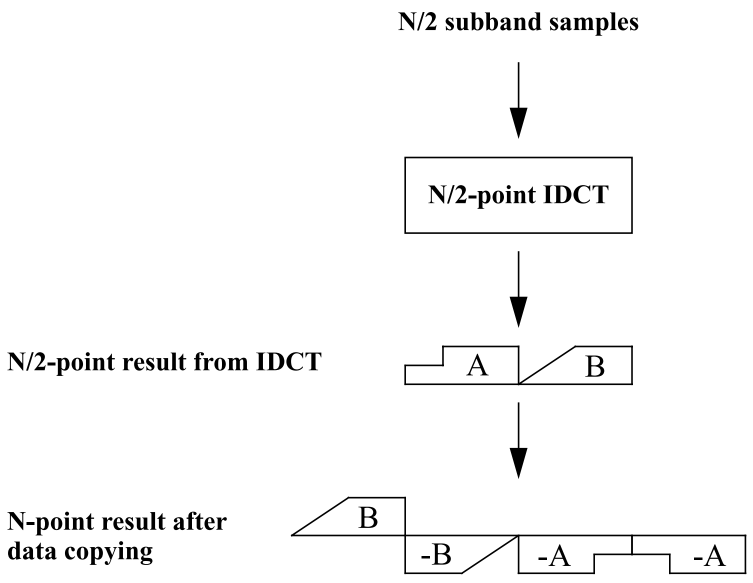

FIGURE 13. Symmetries present in the IMDCT transform (Source: [16]).

The N-point result is identical to the N-point IMDCT as defined by the standard.

This means that only 6 and 18 points need to be computed, respectively. These points can be computed from a modified version of the IDCT using a Lee-style ([18]) method for decomposing the 6- and 18-point transforms into 3-, 4-, and 5-point IDCT kernels.

The short block 6-point transform is decomposed into two 3-point transforms that can be evaluated directly:

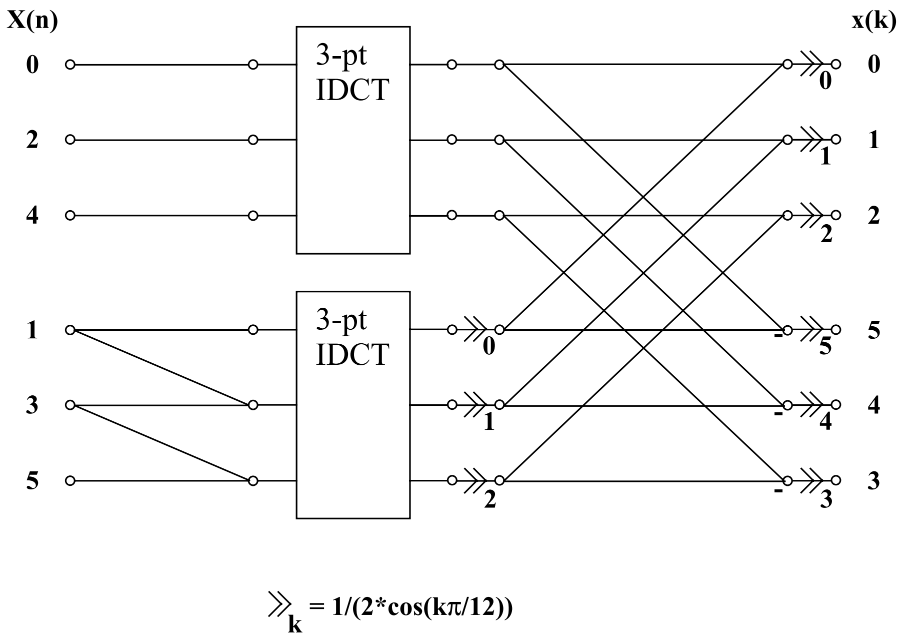

FIGURE 14. Decomposition of the 6-point IDCT into two 3-point kernels.

The long block 18-point transform is decomposed in a similar fashion into two 9-point parts. These 9-point parts are then decomposed further into a 4- and a 5-point part which are directly evaluated.

3.5 Polyphase Filterbank

3.5.1 Definition

The polyphase filterbank converts the time-domain samples from the IMDCT transform in each subband to PCM samples. The conversion involves the following steps:

1. “Matrixing” of 32 subband samples to produce a 64 values V vector and,
2. Windowing of selected samples from the V vector FIFO with a constant window function D to produce a W vector and,
3. Summing the W vector with itself to produce 32 output PCM samples.

Steps 2 and 3 above are straightforward to implement, especially in a DSP that has special addressing capabilities. Step 1 is also straightforward to implement as it is defined by the standard:

$$\text{for i = 0 to 63 do:}$$

$$V_i = \sum_{k=0}^{31} N_{ik} * \mathrm{S}_k$$

$$\left(\begin{array}{c} N_{ik} = \cos\left[(16+i)(2k+i)\frac{\pi}{64}\right] \\ \\ \mathrm{S}_k = \text{Subband samples} \end{array} \right)$$

Polyphase Filterbank Matrixing (EQ 9)

3.5.2 Implementation Issues

We have examined steps 2 and 3 for possible enhancements, but there are no obvious ways to improve upon them. A literature search for improvements did not yield any results either.

Two possible implementations for the matrixing operation (step 1) are described below.

3.5.3 Direct Calculation

A direct calculation of the matrixing operation is easy to implement since it only consists of two nested for-loops.

3.5.4 32-point Fast DCT Implementation

Konstantinides has shown in [16] that the matrixing operation in step 1 can be substantially improved by the use of a 32-point fast DCT transformation and some data copy operations:

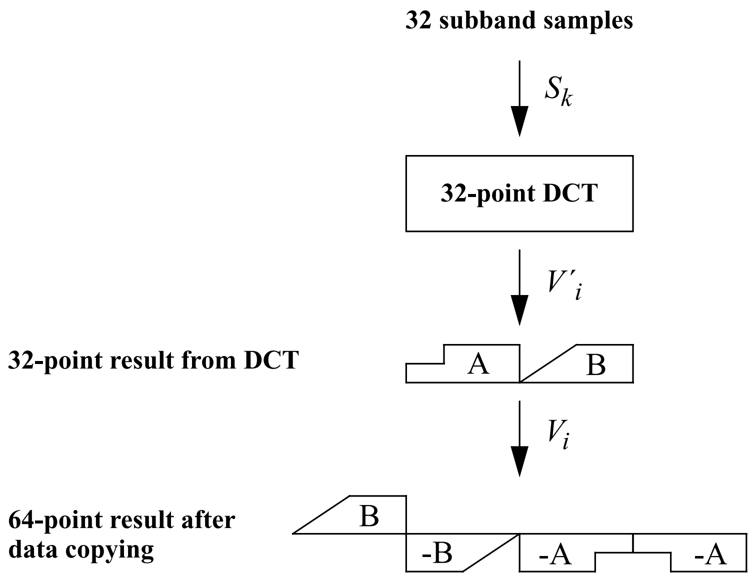

FIGURE 15. Symmetries present in the matrixing operation (Source: [16]).

The problem is then reduced to finding a good implementation of the 32-point DCT:

for i = 0 to 31 do:

$$V'_i = \sum_{k=0}^{31} S_k \cos\left[\frac{\pi}{64}(2k+1)i\right]$$

32-point DCT for Subband Synthesis (EQ 10)

One of the common fast DCT algorithms for 2^m points is described by Lee in [18]. It has a simple recursive structure where the transform is decomposed into even and odd parts:

$$X(n) = \sum_{k=0}^{N-1} x(k)\cos\left(\pi(2k+1)\frac{n}{2N}\right), \text{ for n = 0 to N-1}$$

$$g(k) = x(k) + x(N-1-k)$$

$$h(k) = \frac{1}{2\cos\left(\pi(2k+1)\frac{n}{2N}\right)}(x(k) - x(N-1-k))$$

$$G(n) = \sum_{k=0}^{\frac{N}{2}-1} g(k)\cos\left(\pi(2k+1)\frac{n}{N}\right), \text{ for n = 0 to N/2-1}$$

$$H(n) = \sum_{k=0}^{\frac{N}{2}-1} h(k)\cos\left(\pi(2k+1)\frac{n}{N}\right), \text{ for n = 0 to N/2-1}$$

for n = 0 to N/2-1:

$$X(2n) = G(n)$$

$$X(2n+1) = H(n) + H(n+1), \text{ H(N/2)=0}$$

Lee’s fast DCT algorithm. (EQ 11)

The even and odd parts can themselves be decomposed in the same way until the parts are small enough to be computed through direct evaluation, e.g. when N=2.

4 Evaluation of MP3 Decoding Algorithms

4.1 General

In this chapter we evaluate the algorithms listed in chapter 3. We look at the floating point processing and memory requirements, and also the relative implementation complexities.

4.1.1 Floating Point Performance

The processing performance evaluation for the DSP parts (all but Huffman decoding) is based on the number of floating point operations that has to be carried out. Overhead for looping and data transfers are not included since they would not affect the result for some architectures such as DSP's.

Floating point performance is measured in number of floating point operations, "flop". It is mainly taken as flop per second, "flops". This can also be written as kflops (1,000 flops) and Mflops (1,000,000 flops). A smaller flops number indicate a better algorithm.

An MP3 bitstream can contain a maximum of 48,000 samples per second and channel, and a maximum of two channels, for a total of 96,000 samples per second. That means that if an operation requires 6 flop the total performance will be roughly 600 kflops, or 0.6 Mflops.

4.1.2 Memory Requirements

The memory requirements are expressed in bits.

Each floating point number is 32 bits long, based on IEEE754 single-precision numbers.

The PCM samples require 16 bits of storage each.

4.1.3 Implementation Complexity

The implementation complexity is a subjective measure. We take into consideration how hard it is to understand the algorithm, to implement it, and to verify the correctness of the implementation.

4.2 Huffman Decoding

4.2.1 General

It is difficult to exactly determine the demands of the Huffman decoding process since it is entirely dependent upon the contents of the bitstream. We therefore concentrate on the clustered decoding method which is the best general-purpose decoding method found.

4.2.2 Binary Tree Search

This method is moderately simple to implement. The difficult part could be generating the tables.

The processing performance is clearly the worst of the three methods, since every bit of the code-word has to be handled individually. The memory requirements are moderate since the tables are efficiently stored.

4.2.3 Direct Table Lookup

This method is easy to implement, with the possible exception of generating the necessary tables.

The processing performance is clearly the best of the three methods, since every decoding operation will complete in a short fixed time. The drawback is the very large tables which will be on the order of several megabits.

4.2.4 Clustered Decoding

This method is moderately simple to implement. The difficult part could be generating the tables.

Salomonsen et al. have done a study of this method in conjunction with MP3 decoding [4].

They have shown that individual tables should be at most 16 elements long when decoding MP3. It is further shown that the processing requirements are approximately 1 MIPS for a RISC-based architecture and the memory requirements are 56 kbits for the lookup tables.

4.3 Requantizer

4.3.1 General

The requantization step must be performed once for each sample in the bitstream.

4.3.2 $y=x^{4/3}$, Table-based Implementation

This implementation is simple to realize. It requires only one table lookup operation for every sample which translates to a maximum of 100 kflops.

The drawback is that it requires a 256 kbit table (8207 floating point values), which could be too large for some applications.

4.3.3 $y=x^{4/3}$, Newton's Method

This method could be better than the table-based approach when it is too costly to add the memory needed for the table.

The drawback is that it requires 7 flop to calculate y_0 and x^4, and a further 5 flop by 3 iterations = 15 flop to calculate the final value for y. That results in a total of 22 flop per sample, for a maximum of 2.1 Mflops.

This method is not as straightforward as the table-based implementation, but is still relatively easy to realize.

4.4 Inverse Modified Discrete Cosine Transform (IMDCT)

4.4.1 General

The IMDCT operation operates on the 18 samples in each of the 32 subbands. For short blocks a 12-point IMDCT is performed three times on sets of 6 subband samples, and for long blocks a 36-point IMDCT is performed once on all of the 18 subband samples.

The 12-point IMDCT has to be performed a maximum of 16000 times a second for 48 kHz dual channel bitstreams (96000 samples/second / 6 samples per pass = 16000 passes).

For 36-point IMDCT's the corresponding number is 5333 times per second.

The ratio of short to long blocks will vary with the decoded bitstream, and the worst case is assumed here.

4.4.2 Direct Calculation

The IMDCT is uncomplicated in its' original form as described by the standard.

Each iteration requires 2 flop if a 17 kbit lookup table is used for the *cos(i,k)* values.

For short blocks each iteration has to be performed 72 times, for a total of 144 flop per pass * 16000 passes per second = 2.3 Mflops.

For long blocks each iteration has to be performed 648 times, for a total of 1296 flop per pass * 5333 passes per second = 6.9 Mflops. This is clearly the worst case for the direct calculation method.

If no lookup table is used for the *cos(i,k)* values each iteration would involve 5 flop and a *cos()* function evaluation. This method is clearly not suited for real-time decoders.

4.4.3 Fast IMDCT Implementation

The fast IMDCT implementation is relatively complex to realize. This is mostly due to its' irregular nature.

For short blocks, the 12-point IMDCT is calculated by the use of a 6-point IDCT and some data copying operations. The 6-point IDCT is composed of two 3-point IDCT's which each require 6 flop to calculate. The composition of the two IDCT's requires 34 flop of pre- and postprocessing. The total is then 46 flop per pass * 16000 passes per second = 736 kflops.

For long blocks, the 36-point IMDCT is calculated by the use of an 18-point IDCT and some data copying operations. The 18-point IDCT is composed of two 9-point IDCT's which in turn are composed of a 4- and a 5-point IDCT. The 4-point IDCT requires 18 flop to calculate, and the 5-point 26 flop. The composition of the two 4- and 5-point IDCT's requires 19 flop of pre- and postprocessing, and the 9- to 18-point composition 70 flop. The total is then 196 flop per pass * 5333 passes per second = 1.1 Mflops.

The fast IMDCT is clearly superior compared to the direct method in terms of processing requirements.

4.5 Polyphase Filterbank

4.5.1 General

The polyphase filterbank produces 32 output samples for each pass. The filterbank must then be operated a maximum of 3000 times per second for 48 kHz dual channel bitstreams (96000 samples/second / 32 samples per pass = 3000 passes).

4.5.2 Direct Calculation

The matrixing operation in the filterbank is uncomplicated in its' original form as described by the standard.

The drawback is that it requires 2048 iterations to produce the 32 samples for a pass. Each iteration requires 2 flop if a 64 kbit lookup table is used for the N_{ik} values, for a total of 4096 flop per pass * 3000 passes per second = 12.3 Mflops.

If no lookup table is used for the N_{ik} values each iteration would involve 6 flop and a *cos()* function evaluation. This method is clearly not suited for real-time decoders.

4.5.3 32-point Fast DCT Implementation

The fast DCT implementation is relatively complex to realize. This is mostly due to the recursion present which should be manually unrolled for best performance.

We assume that a 2-point DCT kernel is used as the base case for the recursion part. The number of flop to calculate the kernel is F(2) = 5 flop. The general formula for the number of flop for a certain number of points N is F(N)=2*N-1 + 2*F(N/2). Using this formula we can see that the number of flop needed to calculate the full 32-point DCT is F(32)=321 flop. The processing requirement for this part will then be 321 flop per pass * 3000 passes per second = 963 kflops.

The fast DCT is clearly superior compared to the direct method in terms of processing requirements.

5 MP3 Decoder Reference Design

5.1 Overview

The goal of this decoder implementation is to provide an unambigous and clear reference for the designers of MP3 decoders. The emphasis has been on clarity rather than optimizing every single line of the source code as is common for many decoders that can be found on the Internet [9], [10], [11]. These decoders are generally difficult to comprehend even for sections where performance is not an issue.

A few parts of the decoder account for most of the processing required. These parts have been implemented in two versions where one is optimized using the results of this thesis, and the other implements the algorithms as described by the standard.

The implementation is in ANSI C. The source code is listed in Appendix B, and it is also available on the Internet [12].

The compliance of the decoder to the standard has been proven by decoding the test bitstreams in [13] using the decoder in [9] as a reference.

5.2 ANSI-C MP3 Decoder

The structure of the reference decoder is closely modelled after the standard [2]. This makes it easier to use as a tutorial on the standard for parts that might otherwise be hard to understand.

The following source files contain the most important parts of the decoder:

- main.c: top level control of the decoder.
- MP3_Bitstream.c: bitstream functions for reading data from an abstract bitstream.
- MP3_Main.c: decoding of frame header, side information and Huffman data.
- MP3_Huffman[_opt].c: the actual Huffman decoding functions [optimized version].
- MP3_Decoding[_opt].c: all processing from the sample requantization to the filterbank [optimized version].

There are also various other source files for tables, audio output, and remote control.

The source has not been optimized for some parts that must be written specifically for the processor architecture, preferably in assembly language. This includes but is not limited to: bitstream handling, PCM sample output, bit reservoir implementation, and polyphase V and W buffer addressing. These parts should be relatively straightforward to optimize for a particular platform.

6 Discussion and Conclusion

6.1 Overview

The topic for this thesis was to design and implement a reference MP3 decoder, using optimized algorithms for the most demanding parts of the decoder. The source code should be easy to understand so that it can serve as a reference on the standard for designers that need to implement a decoder.

6.2 Algorithm Conclusions

We have investigated different ways to implement the most demanding parts of the decoder. In this section we compare the best algorithms found to the implementations described by the standard.

6.2.1 Huffman Decoding

The standard provides only the tables needed for decoding, so there is no comparison benchmark for the method we selected.

The clustered Huffman decoder is the best general-purpose decoder found, with a processing requirement of about 1 MIPS and a memory requirement of 56 kbit.

6.2.2 Requantizer

The requantizer is specified in the standard as several very time-consuming *pow(x,y)* function calls that are clearly not suited for an efficient implementation.

We have shown that the fastest way to implement the requantizer is to use of a 256 kbit lookup table that requires the equivalent of only 100 kflops. A memory efficient implementation is also shown, but it requires 2.1 Mflops.

6.2.3 Inverse Modified Discrete Cosine Transform (IMDCT)

The IMDCT algorithm suggested by the standard requires a 17 kbit lookup table and 6.9 Mflops.

We have shown that a more efficient IDCT-based algorithm needs no large lookup tables and requires only 1.1 Mflops.

6.2.4 Polyphase Filterbank

The filterbank algorithm suggested by the standard requires a 64 kbit lookup table and 12.3 Mflops.

We have shown that a more efficient DCT-based algorithm needs no large lookup tables and requires only 963 kflops.

6.3 Reference Design Conclusions

The goal of implementing a reference MP3 decoder in ANSI C was reached [12]. The motivation for the reference decoder and this thesis was not only to provide optimized algorithms but also to serve as a tutorial on the standard for other designers of MP3 decoders, since the standard document [2] was viewed as unclear and ambigous.

The standard document proved to be even harder to read than we first thought before starting the work. Some parts were next to impossible to decipher without referencing existing decoder sources (e.g. reordering), thus proving the need for the tutorial and reference parts of this thesis.

6.4 Concluding Remarks and Future Work

We have shown in this thesis how the most demanding parts of an MP3 decoder can be efficiently implemented. We have also written a reference decoder that can be used to gain a better understanding of the standard.

Future work could include looking at DSP specific implementation issues such as utilizing DSP specific bus structures and addressing capabilities. It would be especially interesting to investigate fixed-point algorithms that could be used for lower-cost fixed-point DSP's.

An area that has not been addressed at all by this thesis is hardware implementations of the decoder. That would probably include fixed-point algorithms, and also looking at how to utilize the parallellism available for hardware implementations.

7 References

[1] P. Noll, "MPEG Digital Audio Coding," IEEE Signal Processing Magazine, pp. 59-81, Sep. 1997.

[2] ISO/IEC 11 172-3, "Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio," first edition, Aug. 1993.

[3] T. Painter and A. Spanias, "Perceptual Coding of Digital Audio," Proceedings of the IEEE, vol. 88, no. 4, pp. 451-513, April 2000.

[4] K. Salomonsen et al., "Design and Implementation of an MPEG/Audio Layer III Bitstream Processor," Master's thesis, Aalborg University, Denmark, 1997.

[5] D. Pan, "A Tutorial on MPEG/Audio Compression," IEEE Multimedia, vol. 2, issue 2, pp. 60-74, Summer 1995.

[6] S. Shlien, "Guide to MPEG-1 Audio Standard," IEEE Transactions on Broadcasting, vol. 40, no. 4, Dec. 1994.

[7] R. Schäfer, "MPEG-4: a multimedia compression standard for interactive applications and services," Electronics and Communication Engineering Journal, pp. 253-262, Dec. 1998.

[8] R. Koenen, "MPEG-4: Multimedia for our time," IEEE Spectrum, pp. 26-33, Feb. 1999.

[9] D. Pan et al., "IIS MP3 Decoder Source Code," http://www.mp3-tech.org, April 1995.

[10] W. Jung, "SPLAY MP3 Decoder Source Code," http://splay.sourceforge.net, April 2001.

[11] M. Hipp et al., "MPG123 MP3 Decoder Source Code," http://www.mpg123.de, April 2001.

[12] K. Lagerström, "MP3 Reference Decoder Source Code," http://www.dtek.chalmers.se/~d2ksla, April 2001.

[13] M. Dietz et al., "MPEG-1 Audio Layer III test bitstream package," http://www.iis.fhg.de, May 1994.

[14] Analog Devices Inc., "ADSP-21061 SHARC DSP," http://www.analog.com, April 2001.

[15] Free Software Foundation, "GNU Compiler Collection," http://www.fsf.org, April 2001.

[16] K. Konstantinides, "Fast Subband Filtering in MPEG Audio Coding," IEEE Signal Processing Letters, vol.1, no. 2, Feb. 1994.

[17] S. Marovich, “Faster MPEG-1 Layer III Audio Decoding,” HP Laboratories Palo Alto, June 2000.

[18] B.G. Lee, “FCT - A Fast Cosine Transform,” IEEE International Conference on Acoustics, Speech and Signal Processing San Diego 1984, pp. 28A.3.1-28A3.4, March 1984.

Appendix A Glossary

ADC	Analog to Digital Converter.
CODEC	CODer/DECoder.
CPU	Central Processing Unit.
DCT	Discrete Cosine Transform
DSP	Digital Signal Processor.
F_S	Sampling Frequency, e.g. 44100 Hz for CD audio.
FFT	Fast Fourier Transform
FIFO	First in, first out.
FLOP	Floating-point operation.
FPU	Floating point unit. Hardware math acceleration inside a CPU.
ISO	International Standards Organisation.
MFLOPS	Million floating-point operations per second.
MPEG	Motion Picture Expert Group. Working group within ISO.
PCM	Pulse Code Modulation. Output from an ADC.