pLayer-i
An internet based muzik player

[CSEE W4840 Final Report – May 2009]

Maninder Singh
ms3770@columbia.edu

Nishant R. Shah
nrs2127@columbia.edu

Ramachandran Shankar
rs2857@columbia.edu

1. Abstract

Mp3 is a de facto standard of digital audio compression and is a very common audio format used for consumer audio storage. It is widely uses for the transfer and playback of music on digital audio players. Our aim was to develop an embedded application that streams Mp3 songs over a network and decode it on-the-fly to play it.

2. Introduction

To Process the multimedia data (images, audio etc...) and distribute over a network their compressed versions are used. The simple reasoning behind this approach is to raise the bandwidth capacity to process task in real time and allow the content of signals to be suitable for the band-width of processing systems. Software is the most common tool used to decompress and use the data. Several SOC solutions have been developed but they are built around a RISC processor with a suitable ISA. The Mp3 format though widely used is not very well understood by many people and hence in this Embedded Systems Design class we decided to develop a Mp3 player that does the computationally intensive decoding steps in dedicated hardware blocks and the complex and not so intense parts in software. Here we had a perfect opportunity to get the best of both the worlds. The choice of a network player was to get this decoder working real-time and hence adding a new dimension to the already complex problem.

3. Related Work

The work done on Mp3 is not very limited yet only few papers are available. The Mp3 has a lot of proprietary issues. Though there are many pieces of work that talk about the timing analysis of each part but none explain each block and its significance. We here have tried to do that and also are giving the overview of each block along with its purpose.

4. MP3 Standard

MPEG-1 Audio Layer 3, more commonly referred to as **MP3**, is a digital audio encoding format using a form of lossy data compression. It is a common audio format

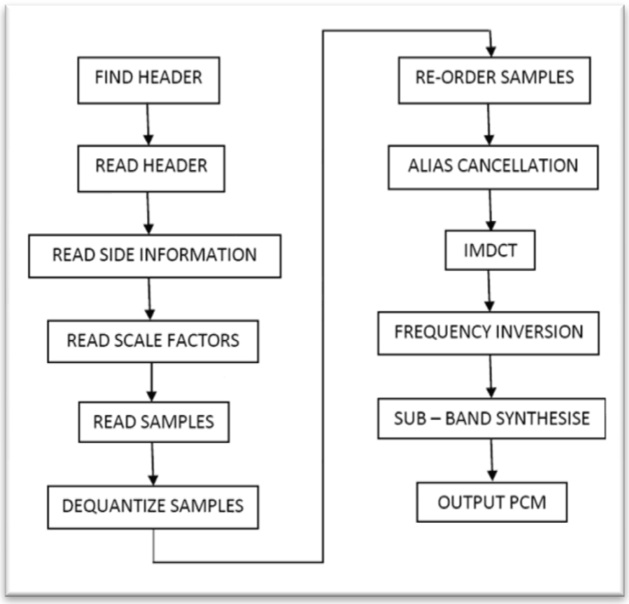
for consumer audio storage, as well as standard encoding for the transfer and playback of music on digital audio players. MP3 is an audio-specific format that was designed by the *Moving Picture Experts Group*. The MP3 standard describes a sound format with one or two sound channels sampled at 32 kHz, 44.1 kHz or 48 kHz, encoded at 32 kbit/s up to 320 kbit/s. In this format, a piece of music can be compressed down to approximately 1 Mb/minute and still sound virtually indistinguishable from the 10 Mb/minute original.

An MP3 file is made up of multiple MP3 frames, which consist of a header and a data block. This sequence of frames is called an elementary stream. Frames are not independent items ("byte reservoir") and therefore cannot be extracted on arbitrary frame boundaries. The MP3 Data blocks contain the (compressed) audio information in terms of frequencies and amplitudes. The diagram shows that the MP3 Header consists of a sync word, which is used to identify the beginning of a valid frame. This is followed by a bit indicating that this is the MPEG standard and two bits that indicate that layer 3 is used; hence MPEG-1 Audio Layer 3 or MP3. After this, the values will differ, depending on the MP3 file. *ISO/IEC 11172-3* defines the range of values for each section of the header along with the specification of the header.

The PCM input is divided into chunks of 576 samples called granules. For two-channel inputs, a sample represents two values. In this case, each granule will contain information about two channels, and the following steps will be repeated for the second channel. The samples are fed through a polyphase filter bank that splits the 576 samples into 32 subbands with 18 samples in each subband. A granule maybe initially silent, but may contain a sharp attack (a sudden loud sound) and so the masking thresholds might be improper for the silent part of the granule. This results in a brief burst of potentially audible noise.

4.2 MP3 DECODER

The decoder basically applies the inverse transformations on the incoming MP3 frames to restore the PCM audio stream for playback. The flowchart of the MP3 decoder is shown below:



4.3 Understanding Mp3 Decoding Scheme:

Using the properties of the human auditory system, lossy codecs and encoders remove inaudible signals to reduce the information content, thus compressing the signal. The job of the encoder is to remove some or all information from a signal component, while at the same time not changing the signal in such a way audible artifacts are introduced.

There's a threshold of hearing – once a signal is below a certain threshold it can't be heard, it's too quiet. Also a loud signal will "mask" other signals sufficiently close in frequency or time. This property is very useful: not only can the nearby masked signals be removed; the audible signal can also be compressed further as the noise introduced by heavy compression will be masked too.

The MP3 standard does not dictate how an encoder should be written (though it assumes the existence of critical bands), and implementers have plenty of freedom to remove content they deem imperceptible. We are taking a simplified engineering approach: for our purpose it's enough to think of these critical bands as fixed frequency regions where masking effects occur.

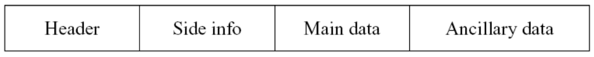
At a very high level, an MP3 encoder works like this: An input source, say a WAV file, is fed to the encoder. There the signal is split into parts (in the time domain), to be processed individually. The encoder then takes one of the short signals and transforms it to the frequency domain. The psychoacoustic model removes as much information as possible, based on the content and phenomena such as masking. The frequency samples, now with less information, are compressed in a generic lossless compression step. The samples, as well as parameters how the samples were compressed, are then written to disk in a binary file format.

The decoder works in reverse. It reads the binary file format, decompresses the frequency samples, reconstructs the samples based on information how content was removed by the model, and then transforms them to the time domain. Let's start with the binary file format.

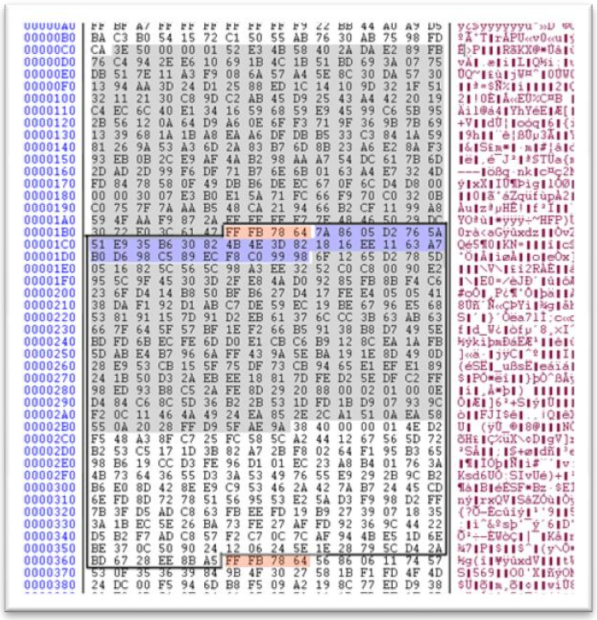
4.3.1 Mp3 Frame

Many computer users know that an MP3 are made up of several "frames", consecutive blocks of data. While important for unpacking the bit stream, frames are not fundamental and cannot be decoded individually. There are two nomenclatures that can be used for frames, physical and logical frames.

A logical frame has many parts: it has a 4 byte *header* easily distinguishable from other data in the bit stream, it has 17 or 32 bytes known as *side information*, and a few hundred bytes of *main data*.



A physical frame has a header, an optional 2 byte checksum, side information, but only some of the main data unless in very rare circumstances. The screenshot below shows a physical frame as a thick black border, the frame header as 4 red bytes, and the side information as blue bytes (this MP3 does not have the optional checksum). The grayed out bytes is the main data that corresponds to the highlighted header and side information. The header for the following physical frame is also highlighted, to show the header always begin at offset 0.



```
000000A0  FF BF A7 FF FF FF FF FF FF F9 22 BB 44 A0 A9 D5
000000B0  BA C3 B0 54 15 72 C1 50 55 AB 76 30 AB 75 98 FD
000000C0  CA 3E 50 00 00 01 52 E3 4B 58 40 2A DA E2 89 FB
000000D0  76 C4 94 2E E6 10 69 1B 4C 1B 51 BD 69 3A 07 75
000000E0  DB 51 7E 11 A3 F9 08 6A 57 A4 5E 8C 30 DA 57 30
000000F0  13 94 AA 3D 24 D1 25 88 ED 1C 14 10 9D 32 1F 51
0000100   32 11 21 30 C8 9D C2 AB 45 D9 25 43 A4 42 20 19
0000110   C4 EC 6C 40 E1 34 16 59 68 59 E9 45 99 C6 5B 95
0000120   2B 56 12 0A 64 D9 A6 0E 6F F3 71 9F 36 9B 7B 69
0000130   13 39 68 1A 1B A8 EA A6 DF DB B5 33 C3 84 1A 59
0000140   81 26 9A 53 A3 6D 2A 83 B7 6D 8B 23 A6 E2 8A F3
0000150   93 EB 0B 2C E9 AF 4A B2 98 AA A7 54 DC 61 7B 6D
0000160   2D AD 2D 99 F6 DF 71 B7 6E 6B 01 63 A4 E7 32 4D
0000170   FD 84 78 58 0F 49 DB B6 DE EC 67 0F 6C D4 D8 00
0000180   00 00 30 07 E3 B0 E1 5A 71 FC 66 F9 70 C0 32 0B
0000190   C0 75 7F 7A AA B5 48 CA 21 94 66 B2 CF 11 99 A8
00001A0   59 4F AA F9 87 2A FF FF FF F7 7E 48 46 50 29 DC
00001B0   30 72 E0 3C 61 47 FF FB 78 64 7A 86 05 D2 76 5A
00001C0   51 E9 35 B6 30 82 4B 4E 3D 82 18 16 EE 11 63 A7
00001D0   B0 D6 98 C5 89 EC F8 C0 99 98 6F 12 65 D2 78 5D
00001E0   05 16 82 5C 56 5C 98 A3 EE 32 52 C0 C8 00 90 E2
00001F0   95 5C 9F 45 30 3D 2F E8 4A D0 92 85 FB 8B F4 C6
0000200   23 6F D4 14 B8 50 BF B6 27 D4 17 FE E4 05 05 41
0000210   38 DA F1 92 D1 AB C7 DE 59 EC 19 BE 67 96 E5 68
0000220   53 81 91 15 7D 91 D2 EB 61 37 6C CC 3B 63 AB 63
0000230   66 7F 64 5F 57 BF 1E F2 66 B5 91 38 B8 D7 49 5E
0000240   BD FD 6B EC FE 6D D0 E1 CB C6 B9 12 8C EA 1A FB
0000250   5D AB E4 B7 96 6A FF 43 9A 5E BA 19 1E 8D 49 0D
0000260   28 E9 53 CB 15 5F 75 DF 73 CB 94 65 E1 EF E1 89
0000270   24 1B 50 D3 2A EB EE 18 81 7D FE D2 5E DF C2 FF
0000280   98 ED 93 B8 C5 2A FE 8D 29 20 88 00 02 01 00 0E
0000290   D4 84 C6 8C 5D 36 B2 2B 53 1D FD 1B D9 07 93 9C
00002A0   F2 0C 11 46 4A 49 24 EA 85 2E 2C A1 51 0A EA 58
00002B0   55 0A 20 28 FF D9 5F AE 9A 38 40 00 00 01 4E D2
00002C0   F5 48 A3 8F C7 25 FC 58 5C A2 44 12 67 56 5D 72
00002D0   B2 53 C5 17 1D 3B 82 A7 2B F8 02 64 F1 95 B3 65
00002E0   98 B6 19 CC D3 FE 96 D1 01 EC 23 A8 B4 01 76 3A
00002F0   4B 73 64 36 55 D3 3A 53 49 76 55 E9 29 2B 9C B2
0000300   B6 E0 8D 42 8E E9 C9 53 46 2A 42 7A B7 24 45 CD
0000310   6E FD 8D 72 78 51 56 95 53 E2 5A D3 F9 98 D2 FF
0000320   7B 3F D5 AD C8 63 FB EE FD 19 B9 27 39 07 18 35
0000330   3A 1B EC 5E 26 BA 73 FE 27 AF FD 92 36 9C 44 22
0000340   D5 B2 F7 AD C8 57 F2 C7 0C 7C AF 94 4B E5 1D 6E
0000350   BE 37 0C 50 90 24 12 06 24 5E 1E 28 79 5C D4 2A
0000360   BD 67 28 EE 8B A5 FF FB 78 64 56 86 06 11 74 57
0000370   53 0F 35 36 39 84 9B 4F 30 27 58 1B F1 FD 4F 4D
0000380   24 DC 00 F5 94 6D B8 F5 09 A2 19 8C 77 ED D9 38
```

The absolutely first thing we do when we decode the MP3 is to unpack the physical frames to logical frames – this is a means of abstraction, once we have a logical frame we can forget about everything else in the bit stream. We do this by reading an offset value in the side information that point to the beginning of the main data.

Why's not the main data for a logical frame kept within the physical frame? At first this seems unnecessarily clumsy, but it has some advantages. The length of a physical frame is constant (within a byte) and solely based on the bit rate and other values stored in the easily found header. This makes seeking to arbitrary frames in the MP3 efficient for media players. Additionally, as frames are not limited to a fixed size in bits, parts of the audio signal with complex sounds can use bytes from preceding frames, in essence giving all MP3:s variable bit rate.

There are some limitations though: a frame can save its main data in several preceding frames, but not following frames – this would make streaming difficult

Before that, we have to make sense of the logical frame, especially the side information and the main data. Unpacking the logical frame requires some information about the different parts. The 4-byte header stores some properties about the audio signal, most importantly the sample rate and the channel mode (mono, stereo etc). The information in the header is useful both for media player software, and for decoding the audio. Note that the header does not store many parameters used by the decoder

The side information is 17 bytes for mono, 32 bytes otherwise. There's lots of information in the side info. Most of the bits describe how the main data should be parsed, but there are also some parameters saved here used by other parts of the decoder.

The first few bits of a chunk are the so-called *scale factors* – basically 21 numbers, which are used for decoding the frame later. The reason the scale factors are stored in the main data and not the side information, as many other parameters, is the scale factors take up quite a lot of space. How the scale factors should be parsed, for example how long a scale factor is in bits, is described in the side information.

Following the scale factors is the actual compressed audio data for this frame. These are a few hundred numbers, and take up most of the space in a frame.

4.3.2 Huffman Decoding

The Huffman coding is actually one of the biggest reasons an MP3 file is so small. The basic idea of Huffman coding is simple. We take some data we want to compress, say a list of 8 bit characters. We then create a value table where we order the characters by frequency. If we don't know beforehand how our list of characters will look, we can order the characters by probability of occurring in the string. We then assign code words to the value table, where we assign the short code words to the most probable values. A code word is simply an n-bit integer designed in such a way there are no ambiguities or clashes with shorter code words.

Decoding is the reverse of coding. If we have a bit string, say 00011111010, we read bits until there's a match in the table.

The standard method of decoding a Huffman coded string is by walking a binary tree, created from the code word table.Instead of walking a tree, we use a lookup table in a clever way.

To understand how Huffman coding is used by MP3, it is necessary to understand exactly what is being coded or decoded. The compressed data that we are about to decompress is frequency domain samples. Each logical frame has up to four chunks – two per channel – each containing up to 576 frequency samples. For a 44100 Hz audio signal, the first frequency sample (index 0) represent frequencies at around 0 Hz, while the last sample (index 575) represent a frequency around 22050 Hz.

These samples are divided into five different regions of variable length. The first three regions are known as the *big values* regions, the fourth region is known as the *count1 region* (or *quad region*), and the fifth is known as the *zero region*. The samples in the zero region are all zero, so these are not actually Huffman coded. If the big values regions and the quad region decode to 400 samples, the remaining 176 are simply padded with 0.

The three big values regions represent the important lower frequencies in the audio. The name big values refer to the information content: when we are done decoding the regions will contain integers in the range –8206 to 8206.

These three big values regions are coded with three different Huffman tables, defined in the MP3 standard. The standard defines 15 large tables for these regions, where each table outputs two frequency samples for a given code word. The tables are designed to compress the “typical” content of the frequency regions as much as possible.

To further increase compression, the 15 tables are paired with another parameter for a total of 29 different ways each of the three regions can be compressed. The side information contains information which of the 29 possibilities to use. Somewhat confusingly, the standard calls these possibilities “tables”. We will call them table pairs instead.

Here’s where it gets interesting: The largest code table defined in the standard has samples no larger than 15. This is enough to represent most signals satisfactory, but sometimes a larger value is required. The second value in the table pair is known as the *linbits* (for some reason), and whenever we have found an output sample that is the maximum value (15) we read linbits number of bits, and add them to the sample. For table pair 1, the linbits is 0, and the maximum sample value is never 15, so we ignore it in this case. For some samples, linbits may be as large as 13, so the maximum value is 15+8191.

4.3.3 Re-quantizing

Having successfully unpacked a frame, we now have a data structure containing audio to be processed further, and parameters how this should be done. Here are our types, what we got from mp3Unpack:

MP3Data is simply an unpacked and parsed logical frame. It contains some useful information, first is the sample rate, second is the channel mode, third are the stereo modes (more about them later). Then are the two-four data chunks, decoded separately. What the values stored in an MP3DataChunk represent will be described soon. For now it’s enough to know frames store the (at most) 576 frequency domain samples.

Due to Fourier: all continuous signals can be created by adding sinusoids together – even the square wave! This means that if we take a pure sine wave, say at 440 Hz, and quantize it, the quantization error will manifest itself as new frequency components in the signal. This makes sense – the quantized sine is not really a pure sine, so there must be something else in the signal. These new frequencies will be all over the spectra, and is noise. If the quantization error is small, the magnitude of the noise will be small.

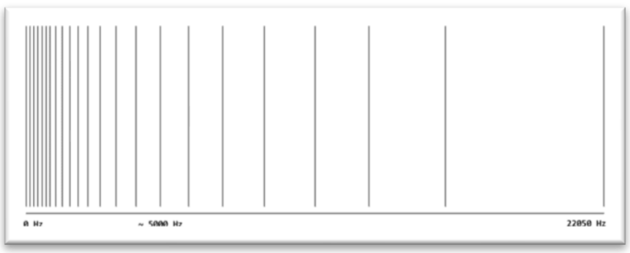
If there’s a strong signal within a critical band, the noise due to quantization errors will be masked, up to the threshold. The encoder can thus throw away as much information as possible from the samples within the critical band, up to the point were discarding more information would result in noise passing the audible threshold. This is the key insight of lossy audio encoding.

After we unpacked the MP3 bit stream and Huffman decoded the frequency samples in a chunk, we ended up with quantized frequency samples between –8206 and 8206. When we take a 16-bit PCM sample and turn it to a float. When we’re done we have a sample in the range –1 to 1, much smaller than 8206. However our new sample has a *much* higher resolution, thanks to the information the encoder left in the frame how the sample should be reconstructed.

The MP3 encoder uses a *non-linear quantizer*, meaning the difference between consecutive re-quantized values is not constant. This is because low amplitude signals are more sensitive to noise, and thus require more bits than stronger signals – think of it as using more bits for small values, and fewer bits for large values. To achieve this non-linearity, the different scaling quantities are non-linear.

The encoder will first raise all samples by 3/4, that is newsample = oldsample$^{3/4}$. The purpose is, according to the literature, to make the signal-to-noise ratio more consistent.

Some frequency regions, partitioned into several *scale factor bands*, are further scaled individually. This is what the scale factors are for: the frequencies in the first scale factor band are all multiplied by the first scale factor, etc. The bands are designed to approximate the critical bands. Here's an illustration of the scale factor bandwidths for a 44100 Hz MP3. The astute reader may notice there are 22 bands, but only 21 scale factors. This is a design limitation that affects the very high frequencies.



The reason these bands are scaled individually is to better control quantization noise. If there's a strong signal in one band, it will mask the noise in this band but not others. The values within a scale factor band are thus quantized independently from other bands by the encoder, depending on the masking effects.

4.3.4 Re-ordering

Before quantizing the frequency samples, the *encoder* will in certain cases reorder the samples in a predefined way. We have already encountered this above: after the reordering by the encoder the "short" chunks with three small chunks of 192 samples each are combined to 576 samples ordered by frequency. This is to improve the efficiency of the Huffman coding, as the method with big values and different tables assume the lower frequencies are first in the list.

When we're done re-quantizing in our decoder, we will reorder the "short" samples back to their original position. After this reordering, the samples in these chunks are no longer ordered by frequency.

4.3.5 IMDCT and Filter-bank Analysis

Now to understand the next blocks, we have to first look at the encoder in order to understand their functionalities.

The input to an encoder is probably a time domain PCM WAV file. The encoder takes 576 time samples, from here on called a granule, and encodes two of these granules to a frame. For an input source with two channels, two granules per channel are stored in the frame. The encoder also saves information how the audio was compressed in the frame. The time domain samples are transformed to the frequency domain in several steps, one granule a time.

Analysis filter bank:
First the 576 samples are fed to a set of 32 band pass filters, where each band pass filter outputs 18 *time domain* samples representing 1/32:th of the frequency spectra of the input signal. If the sample rate is 44100 Hz each band will be approximately 689 Hz wide (22050/32 Hz). Note that there's down-sampling going on here: Common band pass filters will output 576 output samples for 576 input samples, however the MP3 filters also reduce the number of samples by 32, so the combined output of *all* 32 filters is the same as the number of inputs.

This part of the encoder is known as the *analysis filter bank*, and it's a part of the encoder common to all the MPEG-1 layers. Our decoder will do the reverse at the very end of the decoding process, combining the sub-bands to the original signal. These two filter banks are simple conceptually, but real mammoths mathematically – at least the synthesis filter bank.

MDCT:
The output of each band pass filter is further transformed by the MDCT, the modified discrete cosine transform. This transform is just a method of transforming the time domain samples to the

frequency domain. This makes sense: simply dividing the whole frequency spectra in fixed size blocks means the decoder has to take several critical bands into account when quantizing the signal, which results in a worse compression ratio.

The MDCT takes a signal and represents it as a sum of cosine waves, turning it to the frequency domain. Compared to the DFT/FFT and other well-known transforms, the MDCT has a few properties that make it very suited for audio compression.

First of all, the MDCT has the *energy compaction property* common to several of the other discrete cosine transforms. This means most of the information in the signal is concentrated to a few output samples with high energy. If you take an input sequence, do an (M)DCT transform on it, set the "small" output values to 0, then do the inverse transform – the result is a fairly small change in the original input.

Secondly, the MDCT is designed to be performed on consecutive blocks of data, so it has smaller discrepancies at block boundaries compared to other transforms. This also makes it very suited for audio, as we're almost always working with really long signals.

Technically, the MDCT is a so-called *lapped* transform, which means we use input samples from the previous input data when we work with the current input data. The input is 2N time samples and the output is N frequency samples. Instead of transforming 2N length blocks separately, consecutive blocks are overlapped. This overlapping helps reducing artifacts at block boundaries. First we perform the MDCT on say samples 0-35 (inclusive), then 18-53, then 36-71… To smoothen the boundaries between consecutive blocks, the MDCT is usually combined with a windowing function that is performed prior to the transform. A windowing function is simply a sequence of values that are zero outside some region, and often between 0 and 1 within the region, that are to be multiplied with another sequence. For the MDCT smooth, arc-like window functions are usually used, which makes the boundaries of the input block go smoothly to zero at the edges.

In the case of MP3, the MDCT is done on the subbands from the analysis filter bank. In order to get all the nice properties of the MDCT, the transform is not done on the 18 samples directly, but on a windowed signal formed by the concatenation of the 18 previous and the current samples. The combination of the analysis filter bank and the MDCT is known as the *hybrid filter bank*, and it's a very confusing part of the decoder. The analysis filter bank is used by all MPEG-1 layers, but as the frequency bands does not reflect the critical bands, layer 3 added the MDCT on top of the analysis filter bank.

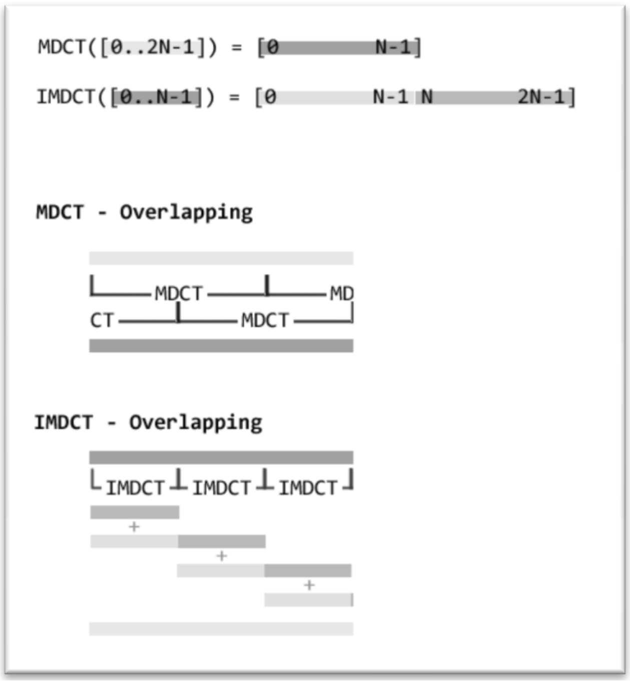
The Decoder Section:
Digesting this information about the encoder leads to a startling realization: we can't actually decode granules, or frames, independently! Due to the overlapping nature of the MDCT we need the inverse-MDCT output of the previous granule to decode the current granule.

Before we do the inverse MDCT, we have to take some deficiencies of the encoder's analysis filter bank into account. The down-sampling in the filter bank introduces some aliasing (where signals are indistinguishable from other signals), but in such a way the synthesis filter bank cancels the aliasing. After the MDCT, the encoder will remove some of this aliasing. This, of course, means we have to undo this alias reduction in our decoder, prior the IMDCT. Otherwise the alias cancellation property of the synthesis filter bank will not work.

When we've dealt with the aliasing, we can IMDCT and then window, remembering to overlap with the output from the previous granule. For short blocks, the three small individual IMDCT inputs are overlapped directly, and this result is then treated as a long block.

The word "overlap" requires some clarifications in the context of the inverse transform. When we speak of the MDCT, a function from 2N inputs to N outputs, this just means we use half the previous samples as inputs to the function. If we've just MDCT-ed 36 input samples from offset 0 in a long sequence, we then MDCT 36 new samples from offset 18.

When we speak of the IMDCT, a function from N inputs to 2N outputs, there's an addition step needed to reconstruct the original sequence. We do the IMDCT on the first 18 samples from the output sequence above. This gives us 36 samples. Output 18..35 are added, element wise, to output 0..17 of the IMDCT output of the next 18 samples.



Before we pass the time domain signal to the synthesis filter bank, there's one final step. Some subbands from the analysis filter bank have inverted frequency spectra, which the encoder corrects. We have to undo this, as with the alias reduction.

A typical MP3 decoder will spend most of its time in the synthesis filter bank – it is by far the most computationally heavy part of the decoder.

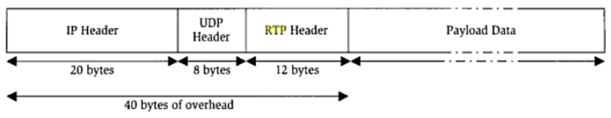
5. Networking

RTP was developed by the Audio/Video Transport working group of the IETF standards organization, and it has since been adopted by several other standards organization. The RTP standard defines a pair of protocols, RTP and the Real-time Transport Control Protocol (RTCP). The former is used for exchange of multimedia data, while the latter is used to periodically send control information and Quality of service parameters.[2]

RTP protocol is designed for end-to-end, real-time, audio or video data flow transport.[3] It allows the recipient to compensate for the jitter and breaks in sequence that may occur during the transfer on an IP network. RTP supports data transfer to multiple destinations by using multicast. RTP provides no guarantee of the delivery, but sequencing of the data makes it possible to detect missing packets. RTP is regarded as the primary standard for audio/video transport in IP networks and is used with an associated profile and payload format.

Multimedia applications need timely delivery and can tolerate some loss in packets. For example, loss of a packet in audio application results may result in loss of a fraction of a second of audio data, which, with suitable error concealment can be made unnoticeable. Multimedia applications require timeliness over reliability.

For our networking needs we used the very basic RTP-2250 protocol. Though the protocol is primitive, its simplicity is what attracted us. The latter versions have a lot QoS tweaks which we really did not require, as loss of a few packets was not a big issue. We desired zero latency while decoding the Ethernet frames. The data link layer frame received over the internet looks as follows:



The needed Mp3 frame lies in the Payload Data.

6. Our approach

Our Simple software decoder

Firstly we implemented the whole mp3 decoder in software. We took reference from mpg123 library and did a one file mp3 decoder which took mp3 file as input and gave output in the form of a .wav file.

There is feature in mpg123 to convert mp3 to wav but it spans through lots of files because mpg123 supports not just mp3 but all MPEG I, II layer 1,2,2.5 and 3. So, the source code s huge. So, we had two options:

1. Run some small Linux kernel (like ucLinux etc.) on Altera board and then port the mpg123 on that kernel.
2. Take all the snippets of code from the mpg123 library required for our mp3 decoding and add all the missing stuff (like the way mp3 data is provided) and then use that

small piece of code with some hardware blocks.

The option 1 seems feasible and required practical no knowledge of mp3 decoding but just the process of running LINUX on our board and after that it was just using the available library. In option 2 we needed to dig into the code but the final output would have been marvelous, we would gain the full knowledge of mp3 decoding process and a simple file would be available for future students to know learn about the concept in a very fast way which we felt was difficult as we could not find any working simple mp3 decoder program.

The main software Blocks designed were :

I. Getting MP3 Data :

In our mp3 decoder version of the program, running on a simple Linux machine, the mp3 data is obtained from a mp3 file residing on the disk. For our FPGA version we streamed the mp3 data over ethernet using the RTP 2250 protocol (poc-2250 streamer program was used). The streamed data is accepted into a buffer after verifying it to be authentic mp3 data. In one RTP packet there was one mp3 frame (418 bytes as we used 44.1 MHz and 128 kbps bitrate).

At the input we used ping-pong approach with two 150 X 420 buffers. First the first 150 frames are input and the decoder does nothing and then after first buffer is filled the ethernet starts filling in the second buffer and the decoder is given the signal to start decoding from the first buffer. In this way after the initial delay the decoder starts its work without any further interruptions.

II. Parsing the Data :

The frame is first tested to be valid mp3 frame by inspecting the first 4 header bytes which have 11 sync bits. After passing from this test the parsing of header and the side information data is done and the internal data structures are filled in with the information on the current mp3 file like – sampling rate, bitrate, mode, etc. Once that is done the side information field in parsed and more data structures are filled that are needed for the decoding process (like scale factors, global gains, windows switching flag and huffman table selection flag, etc). The main data begin field tells us the offset from where the main data begins which can be few frames before the current frame. The pointer is set accordingly in the data field to begin the decoding process.

III. Actual Decoding :

After this the actual decoding of the frame starts. All the blocks from here on are extracted from the mpg123 library. The input to this Block is 380 (418 - 4 - 34) bytes. The output is 4608 bytes of decoded PCM data. This PCM data is actually 2304 , 16 bit samples each. These can be given directly to the Audio PCM output.

Enhancements in the MP3 decoder code :

We used the code with only integer (long 32 bit) calculations without any floating point operations, which were very expensive in terms of time.
Secondly all the table initializations, which took 20 sec when run on the NIOS, were done statically and then the values were entered in the form of arrays. This practically removed any sort of initialization delay.

DCT64 Block in Hardware:

The DCT64 has the following code in c. It takes in 32 32-bit data values and does butterfly operation on them and returns 32 16-bit values out :

```
for(i=15;i>=0;i--)
  *bs++ = (*b1++ + *--b2);
for(i=15;i>=0;i--)
  *bs++ = REAL_MUL((*--b2 - *b1++), *--costab);
b1 = bufs;
costab = pnts[1]+8;
b2 = b1 + 16;
{
 for(i=7;i>=0;i--)
   *bs++ = (*b1++ + *--b2);
 for(i=7;i>=0;i--)
   *bs++ = REAL_MUL((*--b2 - *b1++), *--costab);
 b2 += 32;
 costab += 8;
 for(i=7;i>=0;i--)
   *bs++ = (*b1++ + *--b2);
 for(i=7;i>=0;i--)
   *bs++ = REAL_MUL((*b1++ - *--b2), *--costab);
 b2 += 32;
}
bs = bufs;
costab = pnts[2];
b2 = b1 + 8;
for(j=2;j;j--)
{
 for(i=3;i>=0;i--)
   *bs++ = (*b1++ + *--b2);
 for(i=3;i>=0;i--)
   *bs++ = REAL_MUL((*--b2 - *b1++), costab[i]);
   b2 += 16;
   for(i=3;i>=0;i--)
     *bs++ = (*b1++ + *--b2);
   for(i=3;i>=0;i--)
     *bs++ = REAL_MUL((*b1++ - *--b2), costab[i]);
   b2 += 16;
  }
  b1 = bufs;
  costab = pnts[3];
  b2 = b1 + 4;
for(j=4;j;j--)
  {
   *bs++ = (*b1++ + *--b2);
   *bs++ = (*b1++ + *--b2);
   *bs++ = REAL_MUL((*--b2 - *b1++), costab[1]);
   *bs++ = REAL_MUL((*--b2 - *b1++), costab[0]);
   b2 += 8;
   *bs++ = (*b1++ + *--b2);
   *bs++ = (*b1++ + *--b2);
   *bs++ = REAL_MUL((*b1++ - *--b2), costab[1]);
   *bs++ = REAL_MUL((*b1++ - *--b2), costab[0]);
   b2 += 8;
  }
  bs = bufs;
  costab = pnts[4];
  for(j=8;j;j--)
  {
   real v0,v1;
   v0=*b1++; v1 = *b1++;
   *bs++ = (v0 + v1);
   *bs++ = REAL_MUL((v0 - v1), (*costab));
   v0=*b1++; v1 = *b1++;
   *bs++ = (v0 + v1);
   *bs++ = REAL_MUL((v1 - v0), (*costab));
  }
```

As is very clear from this code it has lots of 32-bit additions and multiplications and the timing analysis showed us that this is one of the main time consuming blocks, so we built a dedicated hardware block for the same. The hardware block takes in 32 32-bit samples and then software does a busy wait for one register bit to go high and then reads back 32 32-bit samples. So, the code below is does that :

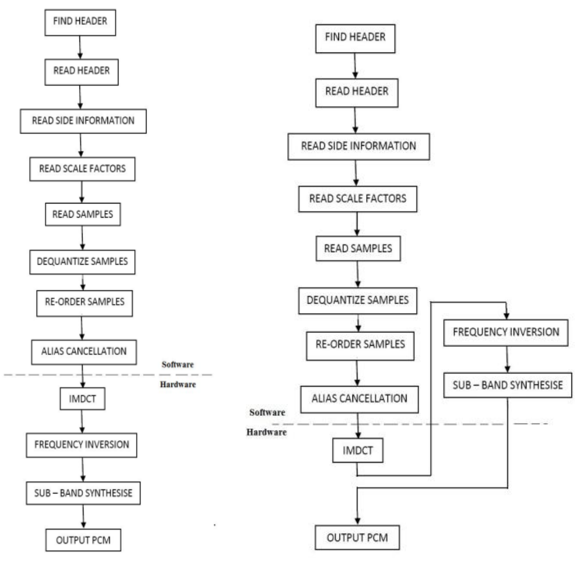
```
for(i=0;i<32;i++)
       IOWR_DCT_DATA(DCT_RAM_DCT_BASE        ,0,
*samples++);
val   =   IORD_DCT_DATA(DCT_RAM_REG_BASE,   0);
while(!val)
{
    val  =  IORD_DCT_DATA(DCT_RAM_REG_BASE,  0);
}
for(i=0;i<32;i++)
   bufs[i] = IORD_DCT_DATA(DCT_RAM_DCT_BASE, 0);
```

We saved 60 milliseconds by this modification, which is significant gain but not enough for playing the mp3 in real time.

Details	Time
1 Mp3 frame	26ms
Full Software Decoder	245 ms
Software + Hardware DCT	185 ms
De-quantize + Huffman	3.5 ms
Anti-alising + Re-ordering	35 ms

Timing Analysis

Polyphase filterbank in hardware:
The next most time consuming Block is polyphase filterbanking, it took 157msec out of 185msec of our final version of mp3 player (that was supposed to take just 26 msec). We made the whole of polyphase filterbanking module in hardware but never got time to test that piece of hardware. Our final goal was to do the decoding phase till reordering in software and then send the data to hardware, which should do the IMDCT+polyphase filter banking and then put the decoded data in CD FIFO for Audio decoder to consume.



Utopian Approach   Our Final Design

6. Hindrances:

Our choice to make the Mp3 decoder looked a bad one when it was difficult to understand the underlying concepts. Not understanding the process was partially because of unavailability of a complete documentation or the “spec sheet” for the Mpeg I, Layer III.

All this turned around once we wrote software Mp3 Player. The flow was now getting clearer and we could now charter our path to completing the decoding process on FPGA.

We faced a few problems because of our own mistakes as well. Trying to compile and use someone else’s code is a no-no. We tried and got stuck debugging and trying to make it work. Starting over with a clean slate was a choice that we advice everyone to make.

Also making the sound work with all its fancy clocks and getting the correct sine-wave, it was a challenge. It took us some sleepless nights and gallons of coffee to make it happen.

Other than times where we tried to shoot ourselves in the foot, it was not very difficult to make the hardware blocks.

7. Conclusion

7.1 Contributions

Ramchandran Shankar:
-Made the windowing hardware block
-Modified Lab2 to set-up the Ethernet
-Initial setup of the Ethernet streaming

Nishant R Shah:
-Modified the DM9000A driver to our needs
-Made the Audio Stack
-Made the DCT hardware block

Maninder Singh:
-Made the Software Mp3 code
-Modified the DM9000A driver to our needs
-Made the Audio Stack
-

7.2 Lessons Learnt

Ramchandran Shankar:
1. Truck loads of VHDL.

2. It is important to have good knowledge of both C and vhdl to implement the mp3 decoder.

3. Timing analysis is of paramount importance.

And....The human ear has more endurance power than we realize. After hearing crap from the FPGA for almost 3 months, trash metal sounds pleasing to the ear!

Nishant R Shah:
This class has taught a few important lessons about embedded systems and also in life. The class taught me how to approach such a huge problem and break it into parts to make it simple and easy sections. Learning about the work was important but learning how to tackle the work is also as important.

My advice to the future warriors of the ESD course is to take a challenging project and get as much out of it. I have learnt a lot from this course and the reason was precisely that. Also don’t be shy to bring your own “weapons”.

On a lighter note I learnt that coffee does not buy you any extra time and also the errors come thick and fast when no help is available.

Maninder Singh:
I learnt all about the MP3 decoding process and learnt how difficult it is to play the nice music I always listen to. Compressing something 12 times... making 50 Mb wave file to below 4 Mb is not so simple and then decoding it back is even more difficult. The worst thing is that you can get something functionally correct but making it work with real time constraints is a real pain.

Secondly, I was working for the first time on FPGAs so I learnt how to make simple hardware blocks and actually understood the importance of hardware blocks (which does things really fast) in the time constraint applications.

8. References

[1] FPGA based Architecture of MP3 Decoding Core for Multimedia Systems, Thuong Le- Tien, Vu Cao-Tuan, Chien Hoang-Dinh

[2] A hardware implementation of an MP3 decoder , Irina F¨altman, Marcus Hast, Andreas Lundgren, Suleyman Malki, Erik Montnemery, Anders

Rangevall, Johannes Sandvall, Milan Stamenkovic

[3] A hardware MP3 decoder with low precision floating point intermediate storage , Andreas Ehliar, Johan Eilert

[4] INTERNATIONAL STANDARD ISO/IEC 1117203:1993 TECHNICAL CORRIGENDUM 1 Published 1996-04-15

[5] www.mp3-tech.org

[6]www.mpg123.de

[7] Praveen Sripada, Mp3 decoder in theory Practice, March 2006.

[8]KRISTER LAGERSTRÖM, Design and Implementation of an MPEG-1 Layer III Audio Decoder, 2001.

Appendix A:

```
/*Software for Mp3 decoder including Ehternet control for NIOS*/

/*mp3decoder  */

#include "basic_io.h"
#include "DM9000A.h"
#include <alt_types.h>
#include "string.h"
#include "huffman.h"
#include "initialize.h"


//#define PLAY_IN_LOOP

#define IOWR_DCT_DATA(base,offset,data) \
  IOWR_32DIRECT(base, offset, data)

#define IORD_DCT_DATA(base, offset) \
  IORD_32DIRECT(base, offset)


#define MAX_MSG_LENGTH 128
#define MAX_X 74
// Ethernet MAC address.  Choose the last three bytes yourself
unsigned char mac_address[6] = { 0x01, 0x60, 0x6E, 0x11, 0x02, 0x0F  };
unsigned int interrupt_number;
unsigned int receive_buffer_length;
unsigned char receive_buffer[1600];
#define MAXARRAYS 75
#define MAXAUDBUF 150
unsigned  char Aud_buffer[MAXAUDBUF][4906];
unsigned char temp_receive_buffer[2][MAXARRAYS][420];
unsigned char rec_count = 0;
unsigned char FLAG_SAFE = 0;
volatile unsigned char START = 0;
unsigned long counter1 = 0, counter2 = 0;
char ShowBuff[23][75];
char *latest = NULL,*oldest = NULL;



#define UDP_PACKET_PAYLOAD_OFFSET 42
#define UDP_PACKET_LENGTH_OFFSET 38
```

```
#define UDP_PACKET_PAYLOAD (transmit_buffer + UDP_PACKET_PAYLOAD_OFFSET)
unsigned char transmit_buffer[] = {
 // Ethernet MAC header
 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, // Destination MAC address
 0x01, 0x60, 0x6E, 0x11, 0x02, 0x0F, // Source MAC address
 0x08, 0x00,                         // Packet Type: 0x800 = IP


 // IP Header
 0x45,              // version (IPv4), header length = 20 bytes
 0x00,              // differentiated services field
 0x00,0x9C,         // total length: 20 bytes for IP header +
                  // 8 bytes for UDP header + 128 bytes for payload
 0x3d, 0x35,        // packet ID
 0x00,              // flags
 0x00,              // fragment offset
 0x80,              // time-to-live
 0x11,              // protocol: 11 = UDP
 0xa3,0x43,         // header checksum: incorrect
 0xc0,0xa8,0x04,0x01, // source IP address
 0xc0,0xa8,0x01,0xff, // destination IP address

 // UDP Header
 0x67,0xd9, // source port port (26585: garbage)
 0x27,0x2b, // destination port (10027: garbage)
 0x00,0x88, // length (136: 8 for UDP header + 128 for data)
 0x00,0x00, // checksum: 0 = none

 // UDP payload
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
 0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67,
  0x74, 0x65, 0x73, 0x74, 0x20, 0x6d, 0x73, 0x67
};



static void ethernet_interrupt_handler() {
  unsigned int receive_status;
  int i;
  unsigned char temp;

    receive_status = ReceivePacket(receive_buffer, &receive_buffer_length);
  if (receive_status == DMFE_SUCCESS) {
    if (receive_buffer_length >= 14) {
      //  A real Ethernet packet
      if (receive_buffer[12] == 8 && receive_buffer[13] == 0 &&
      receive_buffer_length >= 34) {
    // An IP packet
    if (receive_buffer[23] == 0x11) {
      // A UDP packet
      if (receive_buffer_length >= UDP_PACKET_PAYLOAD_OFFSET) {
        if(receive_buffer_length < (419+62) )
        {
            for(i=0; i < (receive_buffer_length - 58) && i < 420 ; i++)
            {
                temp_receive_buffer[FLAG_SAFE][rec_count][i] = receive_buffer[i + 58];
            }

            rec_count++;
            if(rec_count >= MAXARRAYS)
            {

                if(START == 0 && FLAG_SAFE == 1)
                {
                    START = 1;
                }
                else
                    FLAG_SAFE ^= 1;
                rec_count = 0;

            }

        }

      }
    } else {
     //printf("Received non-UDP packet   %x  \n",receive_buffer[23]);
    }
      } else {

    //printf("Received non-IP packet  %x  %x  %d
\n",receive_buffer[12],receive_buffer[13],receive_buffer_length);
    #if 1
    if (receive_buffer[12] == 8 && receive_buffer[13] == 6) //ARP packet
    {


        for(i=0;i<6;i++)
        {
            temp = receive_buffer[i];
            receive_buffer[i] = receive_buffer[i+6];
            receive_buffer[i+6] = temp;
        }

        for(i=22;i<(22+10);i++)
        {
            temp = receive_buffer[i];
            receive_buffer[i] = receive_buffer[i+10];
            receive_buffer[i+10] = temp;
        }

        for(i=0;i<6;i++)
        {
            receive_buffer[i+22] = mac_address[i];
            receive_buffer[i+6] = mac_address[i];
        }



        if (TransmitPacket(receive_buffer, receive_buffer_length)==DMFE_SUCCESS) {
            //printf("\nMessage sent successfully\n");
          } else {
            //printf("\nMessage sending failed\n");
        }
    }
        #endif
     }

        } else {
          printf("Malformed Ethernet packet\n");
      }

  } else {
    printf("Error receiving packet\n");
  }

  /* Display the number of interrupts on the LEDs */
  interrupt_number++;
  outport(SEG7_DISPLAY_BASE, interrupt_number);

  /* Clear the DM9000A ISR: PRS, PTS, ROS, ROOS 4 bits, by RW/C1 */
  dm9000a_iow(ISR, 0x3F);

  dm9000a_iow(IMR, INTR_set);
}

/////////////////////////////////////////////////////////////////////////////////

# define real long

# define REAL_RADIX           15
# define REAL_FACTOR          (32.0 * 1024.0)

# define REAL_PLUS_32767      ( 32767 << REAL_RADIX )
# define REAL_MINUS_32768     ( -32768 << REAL_RADIX )

/* I just changed the (int) to (long) there... seemed right. */
# define DOUBLE_TO_REAL(x)    ((long)((x) * REAL_FACTOR))
# define REAL_TO_SHORT(x)     ((x) >> REAL_RADIX)
# define REAL_MUL(x, y)               (((long long)(x) * (long long)(y)) >> REAL_RADIX)
#  define REAL_SCANF "%ld"
#  define REAL_PRINTF "%ld"


#define MAXFRAMESIZE 3456
#define NEW_DCT9
#define DECODE_DONE 0
#define DECODE_OK 1


#define       MPG_MD_STEREO          0
#define       MPG_MD_JOINT_STEREO    1
#define       MPG_MD_DUAL_CHANNEL    2
#define       MPG_MD_MONO            3
```

```
#define MAXSCALE 32768

#define SINGLE_STEREO -1
#define SINGLE_LEFT    0
#define SINGLE_RIGHT   1
#define SINGLE_MIX     3

#define        MAX_NAME_SIZE          81
#define        SBLIMIT                32
#define        SCALE_BLOCK            12
#define        SSLIMIT                18

#define BLOCK 0x40 /* One decoding block is 64 samples. */

#define WRITE_SHORT_SAMPLE(samples,sum,clip) \
  if( (sum) > REAL_PLUS_32767) { *(samples) = 0x7fff; (clip)++; } \
  else if( (sum) < REAL_MINUS_32768) { *(samples) = -0x8000; (clip)++; } \
  else { *(samples) = REAL_TO_SHORT(sum); }

#define SAMPLE_T short
#define WRITE_SAMPLE(samples,sum,clip) WRITE_SHORT_SAMPLE(samples,sum,clip)


#define backbits(fr,nob) ((void)( \
  fr->bitindex    -= nob, \
  fr->wordpointer += (fr->bitindex>>3), \
  fr->bitindex    &= 0x7 ))

#define getbitoffset(fr) ((-fr->bitindex)&0x7)
#define getbyte(fr)      (*fr->wordpointer++)

#define skipbits(fr, nob) fr->ultmp = ( \
  fr->ultmp = fr->wordpointer[0], fr->ultmp <<= 8, fr->ultmp |= fr->wordpointer[1], \
  fr->ultmp <<= 8, fr->ultmp |= fr->wordpointer[2], fr->ultmp <<= fr->bitindex, \
  fr->ultmp &= 0xffffff, fr->bitindex += nob, \
  fr->ultmp >>= (24-nob), fr->wordpointer += (fr->bitindex>>3), \
  fr->bitindex &= 7 )

#define getbits(fr, nob) ( \
  fr->ultmp = fr->wordpointer[0], fr->ultmp <<= 8, fr->ultmp |= fr->wordpointer[1], \
  fr->ultmp <<= 8, fr->ultmp |= fr->wordpointer[2], fr->ultmp <<= fr->bitindex, \
  fr->ultmp &= 0xffffff, fr->bitindex += nob, \
  fr->ultmp >>= (24-nob), fr->wordpointer += (fr->bitindex>>3), \
  fr->bitindex &= 7,fr->ultmp)

#define get1bit(fr) ( \
  fr->uctmp = *fr->wordpointer << fr->bitindex, fr->bitindex++, \
  fr->wordpointer += (fr->bitindex>>3), fr->bitindex &= 7, fr->uctmp>>7 )

#define getbits_fast(fr, nob) ( \
  fr->ultmp = (unsigned char) (fr->wordpointer[0] << fr->bitindex), \
  fr->ultmp |= ((unsigned long) fr->wordpointer[1]<<fr->bitindex)>>8, \
  fr->ultmp <<= nob, fr->ultmp >>= 8, \
  fr->bitindex += nob, fr->wordpointer += (fr->bitindex>>3), \
  fr->bitindex &= 7, fr->ultmp )


struct mp3_header{
    int error_protection;
    int channels;
    int bitrate;
    int sampling_frequency;
    int padding;
    int extension;
    int mode;
    int mode_ext;
    int stereo;
    int single;
    int copyright;
    int original;
    int emphasis;
    int framesize; /* computed framesize */
    int fd;
    int bo;
    unsigned char Isfirstflag;
    unsigned char parsebuffer[MAXFRAMESIZE+512];;
    unsigned long buffersize;
    int longLimit[9][23];
    int shortLimit[9][14];
    real gainpow2[256+118+4];
    /* bitstream info; bsi */
    int bitindex;
    unsigned char *wordpointer;
    /* temporary storage for getbits stuff */
    unsigned long ultmp;
    unsigned char uctmp;
    real hybrid_block[2][2][SBLIMIT*SSLIMIT];
    int hybrid_blc[2];
    real *real_buffs[2][2];
    unsigned char *rawbuffs;
    real *decwin; /* _the_ decode table */
    unsigned char* rawdecwin; /* the block with all decwins */
};
typedef struct mp3_header header;

#ifdef NEW_DCT9
static real cos9[3] = {30791, -5690, -25101 };
static real cos18[3] = {32270, -11207, -21062 };
#endif

//static real ispow[8207];
static real aa_ca[8] = {-16858, -15457, -10268, -5960, -3099, -1342, -465, -121};
static real aa_cs[8]= {28098, 28892, 31117, 32221, 32621, 32740, 32764, 32767};

static real win[4][36] =
{

    {
        1057, 3512, 6599, 10669, 16383, 25158, 40672, 76412, 253752,
        -276939, -99584, -63843, -48329, -39554, -33840, -29770, -26683, -24228,
        -22201, -20474, -18965, -17616, -16384, -15238, -14153, -13110, -12091,
        -11079, -10060, -9016, -7932, -6786, -5554, -4204, -2695, -969
    },
    {
        1057, 3512, 6599, 10669, 16383, 25158, 40672, 76412, 253752,
        -276939, -99584, -63843, -48329, -39554, -33840, -29770, -26683, -24228,
        -22222, -20651, -19426, -18471, -17733, -17179, -16638, -15267, -13010,
        -9983, -6324, -2190,0, 0, 0, 0, 0, 0
    },
    {
        3512, 16383, 76412, -99584, -39554, -26683, -20474, -16384, -13110, -10060, -6786, -2695
        ,0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0

    },
    {
        0, 0, 0, 0, 0, 0,9880, 48034, 228651, -298002, -115969, -75050,
        -54485, -42813, -35482, -30493, -26913, -24251,
        -22201, -20474, -18965, -17616, -16384, -15238, -14153, -13110, -12091,
        -11079, -10060, -9016, -7932, -6786, -5554, -4204, -2695, -969
    }
};
```

```
static real win1[4][36] =
{
{
	1057, -3512, 6599, -10669, 16383, -25158, 40672, -76412, 253752, 276939, -99584, 63843, -48329, 39554,
-33840, 29770, -26683, 24228, -22201, 20474, -18965, 17616, -16384, 15238, -14153, 13110, -12091, 11079, -
10060, 9016, -7932, 6786, -5554, 4204, -2695, 969
},

{
	1057, -3512, 6599, -10669, 16383, -25158, 40672, -76412, 253752, 276939, -99584, 63843, -48329, 39554,
-33840, 29770, -26683, 24228, -22222, 20651, -19426, 18471, -17733, 17179, -16638, 15267, -13010, 9983, -
6324, 2190, 0, 0, 0, 0, 0, 0
},

{
	3512, -16383, 76412, 99584, -39554, 26683, -20474, 16384, -13110, 10060, -6786, 2695, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
},

{
	0, 0, 0, 0, 0, 0, 9880, -48034, 228651, 298002, -115969, 75050, -54485, 42813, -35482,30493, -26913,
24251, -22201, 20474, -18965, 17616, -16384, 15238, -14153, 13110, -12091, 11079, -10060, 9016, -7932,
6786, -5554, 4204, -2695, 969
}
};

real COS9[9] = {32768, 32270, 30791, 28377, 25101, 21062, 16384, 11207, 5690 }; /* dct36_3dnow wants to
use that */
static real COS6_1 = 28377,COS6_2 = 16384;
real tfcos36[9] = {16446, 16961, 18077, 20001, 23170, 28564, 38767, 63302, 187982 }; /* dct36_3dnow wants
to use that */
static real tfcos12[3] = {16961, 23170, 63302};

static int mapbuf0[9][152];
static int mapbuf1[9][156];
static int mapbuf2[9][44];
static int *map[9][3];
static int *mapend[9][3];



struct bandInfoStruct
{
  int longIdx[23];
```

```
    int longDiff[22];
    int shortIdx[14];
    int shortDiff[13];
};

struct gr_info_s
{
    int scfsi;
    unsigned part2_3_length;
    unsigned big_values;
    unsigned scalefac_compress;
    unsigned block_type;
    unsigned mixed_block_flag;
    unsigned table_select[3];
    unsigned subblock_gain[3];
    unsigned maxband[3];
    unsigned maxbandl;
    unsigned maxb;
    unsigned region1start;
    unsigned region2start;
    unsigned preflag;
    unsigned scalefac_scale;
    unsigned count1table_select;
    real *full_gain[3];
    real *pow2gain;
};

typedef struct III_sideinfo
{
    unsigned main_data_begin;
    unsigned private_bits;
    struct { struct gr_info_s gr[2]; } ch[2 ];
}sideinfo ;


const struct bandInfoStruct bandInfo[9] =
{
    { /* MPEG 1.0 */
        {0,4,8,12,16,20,24,30,36,44,52,62,74, 90,110,134,162,196,238,288,342,418,576},
        {4,4,4,4,4,4,6,6,8, 8,10,12,16,20,24,28,34,42,50,54, 76,158},
        {0,4*3,8*3,12*3,16*3,22*3,30*3,40*3,52*3,66*3, 84*3,106*3,136*3,192*3},
        {4,4,4,4,6,8,10,12,14,18,22,30,56}
    },
    {
```

```
    {0,4,8,12,16,20,24,30,36,42,50,60,72, 88,106,128,156,190,230,276,330,384,576},
    {4,4,4,4,4,4,6,6,6, 8,10,12,16,18,22,28,34,40,46,54, 54,192},
    {0,4*3,8*3,12*3,16*3,22*3,28*3,38*3,50*3,64*3, 80*3,100*3,126*3,192*3},
    {4,4,4,4,6,6,10,12,14,16,20,26,66}
  },
  {
    {0,4,8,12,16,20,24,30,36,44,54,66,82,102,126,156,194,240,296,364,448,550,576},
    {4,4,4,4,4,4,6,6,8,10,12,16,20,24,30,38,46,56,68,84,102, 26},
    {0,4*3,8*3,12*3,16*3,22*3,30*3,42*3,58*3,78*3,104*3,138*3,180*3,192*3},
    {4,4,4,4,6,8,12,16,20,26,34,42,12}
  },
  { /* MPEG 2.0 */
    {0,6,12,18,24,30,36,44,54,66,80,96,116,140,168,200,238,284,336,396,464,522,576},
    {6,6,6,6,6,6,8,10,12,14,16,20,24,28,32,38,46,52,60,68,58,54 } ,
    {0,4*3,8*3,12*3,18*3,24*3,32*3,42*3,56*3,74*3,100*3,132*3,174*3,192*3} ,
    {4,4,4,6,6,8,10,14,18,26,32,42,18 }
  },
  { /* Twiddling 3 values here (not just 330->332!) fixed bug 1895025. */
    {0,6,12,18,24,30,36,44,54,66,80,96,114,136,162,194,232,278,332,394,464,540,576},
    {6,6,6,6,6,6,8,10,12,14,16,18,22,26,32,38,46,54,62,70,76,36 },
    {0,4*3,8*3,12*3,18*3,26*3,36*3,48*3,62*3,80*3,104*3,136*3,180*3,192*3},
    {4,4,4,6,8,10,12,14,18,24,32,44,12 }
  },
  {
    {0,6,12,18,24,30,36,44,54,66,80,96,116,140,168,200,238,284,336,396,464,522,576},
    {6,6,6,6,6,6,8,10,12,14,16,20,24,28,32,38,46,52,60,68,58,54 },
    {0,4*3,8*3,12*3,18*3,26*3,36*3,48*3,62*3,80*3,104*3,134*3,174*3,192*3},
    {4,4,4,6,8,10,12,14,18,24,30,40,18 }
  },
  { /* MPEG 2.5 */
    {0,6,12,18,24,30,36,44,54,66,80,96,116,140,168,200,238,284,336,396,464,522,576},
    {6,6,6,6,6,6,8,10,12,14,16,20,24,28,32,38,46,52,60,68,58,54},
    {0,12,24,36,54,78,108,144,186,240,312,402,522,576},
    {4,4,4,6,8,10,12,14,18,24,30,40,18}
  },
  {
    {0,6,12,18,24,30,36,44,54,66,80,96,116,140,168,200,238,284,336,396,464,522,576},
    {6,6,6,6,6,6,8,10,12,14,16,20,24,28,32,38,46,52,60,68,58,54},
    {0,12,24,36,54,78,108,144,186,240,312,402,522,576},
    {4,4,4,6,8,10,12,14,18,24,30,40,18}
  },
  {
    {0,12,24,36,48,60,72,88,108,132,160,192,232,280,336,400,476,566,568,570,572,574,576},
    {12,12,12,12,12,12,16,20,24,28,32,40,48,56,64,76,90,2,2,2,2,2},
        {0, 24, 48, 72,108,156,216,288,372,480,486,492,498,576},
        {8,8,8,12,16,20,24,28,36,2,2,2,26}
    }
};

static real cos64[16] = {16403 ,16563 ,16890 ,17401 ,18124 ,19101 ,20398 ,22112 ,24396 ,27503 ,31869 ,38320
,48632 ,67429 ,111659 ,333897 };
static real cos32[8] = {16463 ,17121 ,18577 ,21195 ,25826 ,34756 ,56440 ,167152};
static real cos16[4] = {16704 ,19704 ,29490 ,83981};
static real cos8[2] = {17733 ,42813};
static real cos4[1] = {23170};
real *pnts[] = { cos64,cos32,cos16,cos8,cos4 };

static long intwinbase[] = {
     0,    -1,    -1,    -1,    -1,    -1,    -1,    -2,    -2,    -2,
    -2,    -3,    -3,    -4,    -4,    -5,    -5,    -6,    -7,    -7,
    -8,    -9,   -10,   -11,   -13,   -14,   -16,   -17,   -19,   -21,
   -24,   -26,   -29,   -31,   -35,   -38,   -41,   -45,   -49,   -53,
   -58,   -63,   -68,   -73,   -79,   -85,   -91,   -97,  -104,  -111,
  -117,  -125,  -132,  -139,  -147,  -154,  -161,  -169,  -176,  -183,
  -190,  -196,  -202,  -208,  -213,  -218,  -222,  -225,  -227,  -228,
  -228,  -227,  -224,  -221,  -215,  -208,  -200,  -189,  -177,  -163,
  -146,  -127,  -106,   -83,   -57,   -29,     2,    36,    72,   111,
   153,   197,   244,   294,   347,   401,   459,   519,   581,   645,
   711,   779,   848,   919,   991,  1064,  1137,  1210,  1283,  1356,
  1428,  1498,  1567,  1634,  1698,  1759,  1817,  1870,  1919,  1962,
  2001,  2032,  2057,  2075,  2085,  2087,  2080,  2063,  2037,  2000,
  1952,  1893,  1822,  1739,  1644,  1535,  1414,  1280,  1131,   970,
   794,   605,   402,   185,   -45,  -288,  -545,  -814, -1095, -1388,
 -1692, -2006, -2330, -2663, -3004, -3351, -3705, -4063, -4425, -4788,
 -5153, -5517, -5879, -6237, -6589, -6935, -7271, -7597, -7910, -8209,
 -8491, -8755, -8998, -9219, -9416, -9585, -9727, -9838, -9916, -9959,
 -9966, -9935, -9863, -9750, -9592, -9389, -9139, -8840, -8492, -8092,
 -7640, -7134, -6574, -5959, -5288, -4561, -3776, -2935, -2037, -1082,
   -70,   998,  2122,  3300,  4533,  5818,  7154,  8540,  9975, 11455,
 12980, 14548, 16155, 17799, 19478, 21189, 22929, 24694, 26482, 28289,
 30112, 31947, 33791, 35640, 37489, 39336, 41176, 43006, 44821, 46617,
 48390, 50137, 51853, 53534, 55178, 56778, 58333, 59838, 61289, 62684,
 64019, 65290, 66494, 67629, 68692, 69679, 70590, 71420, 72169, 72835,
 73415, 73908, 74313, 74630, 74856, 74992, 75038 };

int bitrate[] = {32,40,48,56,64,80,96,112,128,160,192,224,256,320};
int freq[] = {44100,48000,32000};
```

```
void init(void);
void initbuffer(header *hptr);
void make_decode_tables(header *fr);
void init_stuff(header *fr);
int gethead(header *hptr, char * buf);
int mpg123_read( header *hptr, unsigned char *buffer, unsigned long int buffer_size,unsigned long int
*done,unsigned char *data );
int Get_side_info(header *hptr,int mode,unsigned char *sbuff,int ms_stereo,sideinfo *si);
void decodehead(header *hptr,unsigned char *headbuf);
int gethead(header *hptr, char *buf);
void set_pointer(header *hptr,unsigned backstep,int cur_data_size,unsigned char *);
static int Dequantize_sample(header *fr, real xr[SBLIMIT][SSLIMIT],int *scf, struct gr_info_s *gr_info,int
sfreq,int part2bits);
static int Get_scale_factors(header *fr, int *scf,struct gr_info_s *gr_info,int ch,int gr);
static void dct12(real *in,real *rawout1,real *rawout2,register real *wi,register real *ts);
void dct36(real *inbuf,real *o1,real *o2,real *wintab,real *tsbuf);
void Hybrid(real fsIn[SBLIMIT][SSLIMIT], real tsOut[SSLIMIT][SBLIMIT], int ch,struct gr_info_s *gr_info, header
*fr);
static void Antialias(real xr[SBLIMIT][SSLIMIT],struct gr_info_s *gr_info);
void dct64(real *out0,real *out1,real *samples);
int synth_1to1_my(real *bandPtr, int channel,header *fr, int final,unsigned char *buffer, unsigned long int
*done);

int main()
{
header *hptr,head;
unsigned char buffer[1152 * 2 * 2];
unsigned int done;
unsigned char myflag = 0, mycount = 0;
unsigned char Ahead[4];
int i = 0;
int j = 0;
unsigned short Tmp1;

  unsigned char Buffer[512]={0};
  int k = 0;

  outport(SEG7_DISPLAY_BASE, 0);

 printf("Start  with Init\n");

    hptr = &head;
    init();
    initbuffer(hptr);
    make_decode_tables(hptr);

    hptr->hybrid_blc[0] = hptr->hybrid_blc[1] = 0;
    hptr->bo = 1;
    init_stuff(hptr);

printf("Done  with Init without any parsing requirements\n");




  // Initalize the DM9000 and the Ethernet interrupt handler
  DM9000_init(mac_address);
  interrupt_number = 0;
 alt_irq_register(DM9000A_IRQ, NULL, (void*)ethernet_interrupt_handler);

while(!START)
    ;
k=0;
if(START)
{ dm9000a_iow(IMR, PAR_set);

    printf("inside start\n");
    while(START)
    {
       Ahead[0] = temp_receive_buffer[myflag][mycount][0];
       Ahead[1] = temp_receive_buffer[myflag][mycount][1];
       Ahead[2] = temp_receive_buffer[myflag][mycount][2];
       Ahead[3] = temp_receive_buffer[myflag][mycount][3];
       if(Ahead[0] == 0xff)
       {
          if((Ahead[1] & 0xfe) == 0xfa)
          {
            if(((Ahead[2] & 0xf0) == 0xf0) | //1111 means bad bitrate
                   ((Ahead[2] & 0xf0) == 0x00) | //0000 means free format
                    ((Ahead[2] & 0x0c) == 0x0c)) //sampling ferq: 11 is reserved
                printf("BAD FRAME\n");
             else
             {
               decodehead(hptr, Ahead);


                //counter1 = IORD(COUN_BASE,0);
          mpg123_read( hptr, buffer, (1152 * 2 * 2), &done, (unsigned char
*)&temp_receive_buffer[myflag][mycount][4]);


#ifdef PLAY_IN_LOOP
       //done = 4608;
       if(done > 0)
       {
          for(i=0;i<done;i++)
            Aud_buffer[k][i] = buffer[i];
         k++;
         if(k>=MAXAUDBUF)
          k=0;
       }
#else  //send the info to audio FIFO
      for(i=0;i<done;)
      {
        if(!IORD(AUDIO_0_BASE,0))
        {
          IOWR(AUDIO_0_BASE,0,(unsigned short)((buffer[i+1]<<8)|buffer[i]));
          i+=2;
        }
      }
#endif

       }
     }
    }
    mycount++;
    if(mycount >= MAXARRAYS)
    {
#ifdef PLAY_IN_LOOP
      if(myflag == 1)
      {
        break;
      }
#endif
      mycount = 0;
      myflag ^= 1;
    }
  }
}
j=0;
k=0;
```

```
i=0;
/* mask NIC interrupts IMR: PAR only */
//dm9000a_iow(IMR, PAR_set);
#ifdef PLAY_IN_LOOP
printf("Came here\n");
while(1)
{
   while(j<1024)
   {
     if(!IORD(AUDIO_0_BASE,0))
     {
       Tmp1=(Aud_buffer[k][i+1]<<8)| Aud_buffer[k][i] ;
       IOWR(AUDIO_0_BASE,0,Tmp1);
       j+=2;
       i+=2;
       if(i>=4608)
       {
          i=0;
          k++;
       }
       if(k == (MAXAUDBUF-60))
       {
          i=0;
          k=0;
       }
     }
   }
   j=0;
}
#endif

  return 0;

 ErrorExit:
  printf("Program terminated with an error condition\n");

  return 1;
}

//MAIN LAYER3 FUNTION : Give it a frame and get back the PCM data... :)

int mpg123_read( header *hptr, unsigned char *buffer, unsigned long int buffer_size,unsigned long int *done,
unsigned char *data )
{
   sideinfo  si;
   int stereo = hptr->stereo;
   int single = hptr->single;
   int ms_stereo,i_stereo,stereo1;
   //int sfreq = hptr->sampling_frequency;
   unsigned char CRC[16], Side[32] ; //Sideinfo can have maximum 32 bytes
   int i=0,CRC_bytes = 0, side_size = 0;
   int scalefacs[2][39]; /* max 39 for short[13][3] mode, mixed: 38, long: 22 */
   int ch,gr,clip = 0;

   static int flag = 0;

   float countersum = 0;

   if(done)
       *done = 0;


   int ss;

   if(stereo == 1)
   { /* stream is mono */
       stereo1 = 1;
       single = SINGLE_LEFT;
   }
   else if(single != SINGLE_STEREO) /* stream is stereo, but force to mono */
       stereo1 = 1;
   else
       stereo1 = 2;


    if(hptr->stereo == 1)
       side_size = 17;
    else
       side_size = 32;

       //skipping the CRC 16 bits ... its weird but there are no crc bits even if error_protection is 1!!!
   #if 0
       if(hptr->error_protection)
       {
          printf("read crc bytes\n");
          CRC_bytes = read(hptr->fd, CRC, 2);
          if (2 != CRC_bytes)
          {
            printf("Error in reading file\n");
            exit(0);
         }
      }
   #endif

   if(hptr->mode == MPG_MD_JOINT_STEREO)
   {
      ms_stereo = (hptr->mode_ext & 0x2)>>1;
      i_stereo  = hptr->mode_ext & 0x1;
   }
   else
      ms_stereo = i_stereo = 0;

   //printf("fd -> %x\n",hptr->fd);
   {
      for(i = 0;i<side_size;i++)
         Side[i] = data[i];

      Get_side_info(hptr,hptr->mode,Side,ms_stereo,&si);

   }

   //printf("data_begin - %d\n",si.main_data_begin);

   //return 0;

   //printf("ya\n");

   set_pointer(hptr,si.main_data_begin,(hptr->framesize - 4 - side_size),(unsigned char *)(data + side_size));

//   if(flag < 20)
//{
//    for(i=0;i<hptr->buffersize;i++)
//       printf("%x %x %x \n",(unsigned char) *(hptr->wordpointer + i),hptr->parsebuffer[i],data[side_size+i]);
//    flag++;
//}


   //printf("na\n");
   for(gr = 0;gr < 2;gr++)
   {
      real hybridIn[2][SBLIMIT][SSLIMIT];
      real hybridOut[2][SSLIMIT][SBLIMIT];
```

```
//printf("\n####################################\n");
    {
       struct gr_info_s *gr_info = &(si.ch[0].gr[gr]);
       long part2bits;


       part2bits = Get_scale_factors(hptr, scalefacs[0],gr_info,0,gr);

       //counter1 = IORD(COUN_BASE,0);
       if(Dequantize_sample(hptr, hybridIn[0], scalefacs[0],gr_info,0,part2bits))
       {
          //printf("dequantization failed!");
          return clip;
       }
       //counter2 = IORD(COUN_BASE,0);
       //countersum = countersum + (counter2 - counter1);
    }
    if(stereo == 2)
    {
       struct gr_info_s *gr_info = &(si.ch[1].gr[gr]);
       long part2bits;

       part2bits = Get_scale_factors(hptr, scalefacs[1],gr_info,1,gr);

       //counter1 = IORD(COUN_BASE,0);
       if(Dequantize_sample(hptr, hybridIn[1],scalefacs[1],gr_info,0,part2bits))
       {
          //printf("dequantization failed!");
          return clip;
       }
       //counter2 = IORD(COUN_BASE,0);
       //countersum = countersum + (counter2 - counter1);
       if(ms_stereo)
       {
          int i;
          int maxb = si.ch[0].gr[gr].maxb;
          if(si.ch[1].gr[gr].maxb > maxb)
             maxb = si.ch[1].gr[gr].maxb;

          for(i=0;i<SSLIMIT*maxb;i++)
          {
             real tmp0 = ((real *)hybridIn[0])[i];
             real tmp1 = ((real *)hybridIn[1])[i];
             ((real *)hybridIn[0])[i] = tmp0 + tmp1;
```

```
            ((real *)hybridIn[1])[i] = tmp0 - tmp1;
        }
    }

    //i_stereo was zero!!!
    //if(i_stereo) III_i_stereo(hybridIn,scalefacs[1],gr_info,sfreq,ms_stereo,fr->lsf);

    if(ms_stereo || i_stereo || (single == SINGLE_MIX) )
    {
        if(gr_info->maxb > si.ch[0].gr[gr].maxb)
            si.ch[0].gr[gr].maxb = gr_info->maxb;
        else
            gr_info->maxb = si.ch[0].gr[gr].maxb;
    }

    switch(single)
    {
        case SINGLE_MIX:
        {
            register int i;
            register real *in0 = (real *) hybridIn[0],*in1 = (real *) hybridIn[1];
            for(i=0;i<SSLIMIT*gr_info->maxb;i++,in0++)
            *in0 = (*in0 + *in1++); /* *0.5 done by pow-scale */
        }
        break;
        case SINGLE_RIGHT:
        {
            register int i;
            register real *in0 = (real *) hybridIn[0],*in1 = (real *) hybridIn[1];
            for(i=0;i<SSLIMIT*gr_info->maxb;i++)
            *in0++ = *in1++;
        }
        break;
    }
}

for(ch=0; ch<stereo1; ch++)
{
    struct gr_info_s *gr_info = &(si.ch[ch].gr[gr]);
    Antialias(hybridIn[ch],gr_info);
    Hybrid(hybridIn[ch], hybridOut[ch], ch,gr_info, hptr);
}

//counter1 = IORD(COUN_BASE,0);
```

```
        for(ss=0;ss<SSLIMIT;ss++)
        {
                clip += synth_1to1_my(hybridOut[0][ss], 0, hptr, 0,buffer,done);
                clip += synth_1to1_my(hybridOut[1][ss], 1, hptr, 1,buffer,done);
        }
        //counter2 = IORD(COUN_BASE,0);
        //countersum = countersum + (counter2 - counter1);
    }

    //printf("after : %f\n",(float)((float)((float)(countersum))/100000));
return DECODE_OK;
}
#define BACKPEDAL 0x10
int synth_1to1_my(real *bandPtr, int channel,header *fr, int final,unsigned char *buffer, unsigned long int
*done)
{
//printf("\nGeneric synth.h\n");

    static const int step = 2;
    SAMPLE_T *samples = (SAMPLE_T *) (buffer + *done);
    int i =1;
    real *b0, **buf; /* (*buf)[0x110]; */
    int clip = 0;
    int bo1;
    static int flag = 0;
    static float diff = 0;


    if(!channel)
    {
        fr->bo--;
        fr->bo &= 0xf;
        buf = fr->real_buffs[0];
    }
    else
    {
        samples++;
        buf = fr->real_buffs[1];
    }



    if(fr->bo & 0x1)
    {
        b0 = buf[0];
        bo1 = fr->bo;
        dct64(buf[1]+((fr->bo+1)&0xf),buf[0]+fr->bo,bandPtr);
    }
    else
    {
        b0 = buf[1];
        bo1 = fr->bo+1;
        dct64(buf[0]+fr->bo,buf[1]+fr->bo+1,bandPtr);
    }


    {
        register int j;
        real *window = fr->decwin + 16 - bo1;

//printf("\nflag = %d\n ",flag++);
        for(j=(BLOCK/4); j; j--, b0+=0x400/BLOCK-BACKPEDAL, window+=0x800/BLOCK-BACKPEDAL,
samples+=step)
        {
            real sum;

            sum  = REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);
            sum += REAL_MUL(*window++, *b0++);
            sum -= REAL_MUL(*window++, *b0++);


//printf("%ld  ",sum);
i++;
if(i%5 == 0);
//printf("\n");
```

```
            WRITE_SAMPLE(samples,sum,clip);
        }

        {
            real sum;
            sum  = REAL_MUL(window[0x0], b0[0x0]);
            sum += REAL_MUL(window[0x2], b0[0x2]);
            sum += REAL_MUL(window[0x4], b0[0x4]);
            sum += REAL_MUL(window[0x6], b0[0x6]);
            sum += REAL_MUL(window[0x8], b0[0x8]);
            sum += REAL_MUL(window[0xA], b0[0xA]);
            sum += REAL_MUL(window[0xC], b0[0xC]);
            sum += REAL_MUL(window[0xE], b0[0xE]);


            WRITE_SAMPLE(samples,sum,clip);
            samples += step;
            b0-=0x400/BLOCK;
            window-=0x800/BLOCK;
        }
        window += bo1<<1;

        for(j=(BLOCK/4)-1; j; j--, b0-=0x400/BLOCK+BACKPEDAL, window-=0x800/BLOCK-BACKPEDAL,
samples+=step)
        {
            real sum;

            sum = -REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            sum -= REAL_MUL(*(--window), *b0++);
            WRITE_SAMPLE(samples,sum,clip);
        }
    }

    if(final) *done += BLOCK*sizeof(SAMPLE_T);

    return clip;


}
void decodehead(header *hptr,unsigned char *headbuf)
{
    int index;
    long temp;

    index = (((headbuf[2]&0xf0) >> 4) - 1);
    hptr->bitrate = bitrate[index];

    if(headbuf[3] & 0xc0 == 0xc0)
        hptr->channels = 1;
    else
        hptr->channels = 2;

    index = ((headbuf[2] & 0x0c) >> 2);
    hptr->sampling_frequency = freq[index];

    hptr->error_protection = (headbuf[1] & 0x01);
    hptr->padding          = ((headbuf[2] & 0x02) >> 1);
    hptr->extension        = (headbuf[2] & 0x01);
    hptr->mode             = ((headbuf[3] & 0xc0) >> 6);
    hptr->mode_ext         = ((headbuf[3] & 0x30) >> 4);
    hptr->copyright        = ((headbuf[3] & 0x08) >> 3);
    hptr->original         = ((headbuf[3] & 0x04) >> 2);
    hptr->emphasis         = (headbuf[3] & 0x03);

    hptr->stereo    = (hptr->mode == MPG_MD_MONO) ? 1 : 2;

    //Hardcoded as of now... need to understand mh->p.flags ... line 580 libmpg123.c
    hptr->single = SINGLE_STEREO;



//  printf("%x%x%x%x \n",headbuf[0],headbuf[1],headbuf[2],headbuf[3] );
```

```
   temp = (long)(144000 * hptr->bitrate);
//  printf("%d\n",temp);
   temp /= hptr->sampling_frequency;
//  printf("%d\n",temp);
   temp +=  hptr->padding;
//  printf("%d\n",temp);

   hptr->framesize = (int) temp;


   //printf("%d\n",hptr->framesize);
}


static void Antialias(real xr[SBLIMIT][SSLIMIT],struct gr_info_s *gr_info)
{
   int sblim;

   if(gr_info->block_type == 2)
   {
         if(!gr_info->mixed_block_flag)
            return;
         sblim = 1;
   }
   else
      sblim = gr_info->maxb-1;

   /* 31 alias-reduction operations between each pair of sub-bands */
   /* with 8 butterflies between each pair                         */

   {
      int sb;
      real *xr1=(real *) xr[1];

      for(sb=sblim; sb; sb--,xr1+=10)
      {
         int ss;
         real *cs=aa_cs,*ca=aa_ca;
         real *xr2 = xr1;

         for(ss=7;ss>=0;ss--)
         { /* upper and lower butterfly inputs */
            register real bu = *--xr2,bd = *xr1;
            *xr2   = REAL_MUL(bu, *cs) - REAL_MUL(bd, *ca);
            *xr1++ = REAL_MUL(bd, *cs++) + REAL_MUL(bu, *ca++);
          }
        }
    }
}

void Hybrid(real fsIn[SBLIMIT][SSLIMIT], real tsOut[SSLIMIT][SBLIMIT], int ch,struct gr_info_s *gr_info, header
*fr)
{
   real (*block)[2][SBLIMIT*SSLIMIT] = fr->hybrid_block;
   int *blc = fr->hybrid_blc;

   real *tspnt = (real *) tsOut;
   real *rawout1,*rawout2;
   int bt,sb = 0;

   static int flag = 0;

   {
      int b = blc[ch];
      rawout1=block[b][ch];
      b=-b+1;
      rawout2=block[b][ch];
      blc[ch] = b;
   }
//printf("\ngr_info->mixed_block_flag = %u\n",gr_info->mixed_block_flag);
   if(gr_info->mixed_block_flag)
   {
      sb = 2;
      dct36(fsIn[0],rawout1,rawout2,win[0],tspnt);
      dct36(fsIn[1],rawout1+18,rawout2+18,win1[0],tspnt+1);
      rawout1 += 36; rawout2 += 36; tspnt += 2;
   }

   if(flag < 10)
   {
      int i = 0;
      real *temp = tspnt;
      //printf("\n-----------------------------------------------------------   \n");
      for(i=0;i<300;i++)
      {
         //printf("%f",*temp);
         temp++;
      }
      flag++;
   }

   bt = gr_info->block_type;
   if(bt == 2)
   {
      for(; sb<gr_info->maxb; sb+=2,tspnt+=2,rawout1+=36,rawout2+=36)
      {
         dct12(fsIn[sb]  ,rawout1   ,rawout2   ,win[2] ,tspnt);
         dct12(fsIn[sb+1],rawout1+18,rawout2+18,win1[2],tspnt+1);
      }
   }
   else
   {
      for(; sb<gr_info->maxb; sb+=2,tspnt+=2,rawout1+=36,rawout2+=36)
      {
         dct36(fsIn[sb],rawout1,rawout2,win[bt],tspnt);
         dct36(fsIn[sb+1],rawout1+18,rawout2+18,win1[bt],tspnt+1);
      }
   }

   for(;sb<SBLIMIT;sb++,tspnt++)
   {
      int i;
      for(i=0;i<SSLIMIT;i++)
      {
         tspnt[i*SBLIMIT] = *rawout1++;
         *rawout2++ = DOUBLE_TO_REAL(0.0);
      }
   }
}

/* Calculation of the inverse MDCT
  used to be static without 3dnow - does that really matter? */

void dct36(real *inbuf,real *o1,real *o2,real *wintab,real *tsbuf)
{
#ifdef NEW_DCT9
   real tmp[18];
#endif
//printf("\nInside DCT36\n");
   {
      register real *in = inbuf;
```

```
    in[17]+=in[16]; in[16]+=in[15]; in[15]+=in[14];
    in[14]+=in[13]; in[13]+=in[12]; in[12]+=in[11];
    in[11]+=in[10]; in[10]+=in[9];  in[9] +=in[8];
    in[8] +=in[7];  in[7] +=in[6];  in[6] +=in[5];
    in[5] +=in[4];  in[4] +=in[3];  in[3] +=in[2];
    in[2] +=in[1];  in[1] +=in[0];

    in[17]+=in[15]; in[15]+=in[13]; in[13]+=in[11]; in[11]+=in[9];
    in[9] +=in[7];  in[7] +=in[5];  in[5] +=in[3];  in[3] +=in[1];


#ifdef NEW_DCT9

    {
      real t3;
      {
        real t0, t1, t2;

        t0 = REAL_MUL(COS6_2, (in[8] + in[16] - in[4]));
        t1 = REAL_MUL(COS6_2, in[12]);

        t3 = in[0];
        t2 = t3 - t1 - t1;
        tmp[1] = tmp[7] = t2 - t0;
        tmp[4]          = t2 + t0 + t0;
        t3 += t1;

        t2 = REAL_MUL(COS6_1, (in[10] + in[14] - in[2]));
        tmp[1] -= t2;
        tmp[7] += t2;
      }
      {
        real t0, t1, t2;

        t0 = REAL_MUL(cos9[0], (in[4] + in[8] ));
        t1 = REAL_MUL(cos9[1], (in[8] - in[16]));
        t2 = REAL_MUL(cos9[2], (in[4] + in[16]));

        tmp[2] = tmp[6] = t3 - t0      - t2;
        tmp[0] = tmp[8] = t3 + t0 + t1;
        tmp[3] = tmp[5] = t3      - t1 + t2;
      }
    }
```

```
{
  real t1, t2, t3;

  t1 = REAL_MUL(cos18[0], (in[2]  + in[10]));
  t2 = REAL_MUL(cos18[1], (in[10] - in[14]));
  t3 = REAL_MUL(COS6_1,    in[6]);

  {
    real t0 = t1 + t2 + t3;
    tmp[0] += t0;
    tmp[8] -= t0;
  }

  t2 -= t3;
  t1 -= t3;

  t3 = REAL_MUL(cos18[2], (in[2] + in[14]));

  t1 += t3;
  tmp[3] += t1;
  tmp[5] -= t1;

  t2 -= t3;
  tmp[2] += t2;
  tmp[6] -= t2;
}

{
  real t0, t1, t2, t3, t4, t5, t6, t7;

  t1 = REAL_MUL(COS6_2, in[13]);
  t2 = REAL_MUL(COS6_2, (in[9] + in[17] - in[5]));

  t3 = in[1] + t1;
  t4 = in[1] - t1 - t1;
  t5 = t4 - t2;

  t0 = REAL_MUL(cos9[0], (in[5] + in[9]));
  t1 = REAL_MUL(cos9[1], (in[9] - in[17]));

  tmp[13] = REAL_MUL((t4 + t2 + t2), tfcos36[17-13]);
  t2 = REAL_MUL(cos9[2], (in[5] + in[17]));

  t6 = t3 - t0 - t2;
```

```
            t0 += t3 + t1;
            t3 += t2 - t1;

            t2 = REAL_MUL(cos18[0], (in[3]  + in[11]));
            t4 = REAL_MUL(cos18[1], (in[11] - in[15]));
            t7 = REAL_MUL(COS6_1, in[7]);

            t1 = t2 + t4 + t7;
            tmp[17] = REAL_MUL((t0 + t1), tfcos36[17-17]);
            tmp[9]  = REAL_MUL((t0 - t1), tfcos36[17-9]);
            t1 = REAL_MUL(cos18[2], (in[3] + in[15]));
            t2 += t1 - t7;

            tmp[14] = REAL_MUL((t3 + t2), tfcos36[17-14]);
            t0 = REAL_MUL(COS6_1, (in[11] + in[15] - in[3]));
            tmp[12] = REAL_MUL((t3 - t2), tfcos36[17-12]);

            t4 -= t1 + t7;

            tmp[16] = REAL_MUL((t5 - t0), tfcos36[17-16]);
            tmp[10] = REAL_MUL((t5 + t0), tfcos36[17-10]);
            tmp[15] = REAL_MUL((t6 + t4), tfcos36[17-15]);
            tmp[11] = REAL_MUL((t6 - t4), tfcos36[17-11]);
        }

#define MACRO(v) { \
        real tmpval; \
        tmpval = tmp[(v)] + tmp[17-(v)]; \
        out2[9+(v)] = REAL_MUL(tmpval, w[27+(v)]); \
        out2[8-(v)] = REAL_MUL(tmpval, w[26-(v)]); \
        tmpval = tmp[(v)] - tmp[17-(v)]; \
        ts[SBLIMIT*(8-(v))] = out1[8-(v)] + REAL_MUL(tmpval, w[8-(v)]); \
        ts[SBLIMIT*(9+(v))] = out1[9+(v)] + REAL_MUL(tmpval, w[9+(v)]); }

        {
            register real *out2 = o2;
            register real *w = wintab;
            register real *out1 = o1;
            register real *ts = tsbuf;

            MACRO(0);
            MACRO(1);
            MACRO(2);
            MACRO(3);
```

```
          MACRO(4);
          MACRO(5);
          MACRO(6);
          MACRO(7);
          MACRO(8);
        }

#else
#endif

    }
}


/* new DCT12 */
static void dct12(real *in,real *rawout1,real *rawout2,register real *wi,register real *ts)
{
#define DCT12_PART1 \
    in5 = in[5*3];  \
    in5 += (in4 = in[4*3]); \
    in4 += (in3 = in[3*3]); \
    in3 += (in2 = in[2*3]); \
    in2 += (in1 = in[1*3]); \
    in1 += (in0 = in[0*3]); \
    \
    in5 += in3; in3 += in1; \
    \
    in2 = REAL_MUL(in2, COS6_1); \
    in3 = REAL_MUL(in3, COS6_1);

#define DCT12_PART2 \
    in0 += REAL_MUL(in4, COS6_2); \
    \
    in4 = in0 + in2; \
    in0 -= in2;      \
    \
    in1 += REAL_MUL(in5, COS6_2); \
    \
    in5 = REAL_MUL((in1 + in3), tfcos12[0]); \
    in1 = REAL_MUL((in1 - in3), tfcos12[2]); \
    \
    in3 = in4 + in5; \
    in4 -= in5;      \
    \
in2 = in0 + in1; \
in0 -= in1;

{
   real in0,in1,in2,in3,in4,in5;
   register real *out1 = rawout1;
   ts[SBLIMIT*0] = out1[0]; ts[SBLIMIT*1] = out1[1]; ts[SBLIMIT*2] = out1[2];
   ts[SBLIMIT*3] = out1[3]; ts[SBLIMIT*4] = out1[4]; ts[SBLIMIT*5] = out1[5];

   DCT12_PART1

   {
      real tmp0,tmp1 = (in0 - in4);
      {
         real tmp2 = REAL_MUL((in1 - in5), tfcos12[1]);
         tmp0 = tmp1 + tmp2;
         tmp1 -= tmp2;
      }
      ts[(17-1)*SBLIMIT] = out1[17-1] + REAL_MUL(tmp0, wi[11-1]);
      ts[(12+1)*SBLIMIT] = out1[12+1] + REAL_MUL(tmp0, wi[6+1]);
      ts[(6 +1)*SBLIMIT] = out1[6 +1] + REAL_MUL(tmp1, wi[1]);
      ts[(11-1)*SBLIMIT] = out1[11-1] + REAL_MUL(tmp1, wi[5-1]);
   }

   DCT12_PART2

   ts[(17-0)*SBLIMIT] = out1[17-0] + REAL_MUL(in2, wi[11-0]);
   ts[(12+0)*SBLIMIT] = out1[12+0] + REAL_MUL(in2, wi[6+0]);
   ts[(12+2)*SBLIMIT] = out1[12+2] + REAL_MUL(in3, wi[6+2]);
   ts[(17-2)*SBLIMIT] = out1[17-2] + REAL_MUL(in3, wi[11-2]);

   ts[(6 +0)*SBLIMIT]  = out1[6+0] + REAL_MUL(in0, wi[0]);
   ts[(11-0)*SBLIMIT] = out1[11-0] + REAL_MUL(in0, wi[5-0]);
   ts[(6 +2)*SBLIMIT]  = out1[6+2] + REAL_MUL(in4, wi[2]);
   ts[(11-2)*SBLIMIT] = out1[11-2] + REAL_MUL(in4, wi[5-2]);
}

in++;

{
   real in0,in1,in2,in3,in4,in5;
   register real *out2 = rawout2;

   DCT12_PART1
```

```
    {
        real tmp0,tmp1 = (in0 - in4);
        {
            real tmp2 = REAL_MUL((in1 - in5), tfcos12[1]);
            tmp0 = tmp1 + tmp2;
            tmp1 -= tmp2;
        }
        out2[5-1] = REAL_MUL(tmp0, wi[11-1]);
        out2[0+1] = REAL_MUL(tmp0, wi[6+1]);
        ts[(12+1)*SBLIMIT] += REAL_MUL(tmp1, wi[1]);
        ts[(17-1)*SBLIMIT] += REAL_MUL(tmp1, wi[5-1]);
    }

    DCT12_PART2

    out2[5-0] = REAL_MUL(in2, wi[11-0]);
    out2[0+0] = REAL_MUL(in2, wi[6+0]);
    out2[0+2] = REAL_MUL(in3, wi[6+2]);
    out2[5-2] = REAL_MUL(in3, wi[11-2]);

    ts[(12+0)*SBLIMIT] += REAL_MUL(in0, wi[0]);
    ts[(17-0)*SBLIMIT] += REAL_MUL(in0, wi[5-0]);
    ts[(12+2)*SBLIMIT] += REAL_MUL(in4, wi[2]);
    ts[(17-2)*SBLIMIT] += REAL_MUL(in4, wi[5-2]);
}

in++;

{
    real in0,in1,in2,in3,in4,in5;
    register real *out2 = rawout2;
    out2[12]=out2[13]=out2[14]=out2[15]=out2[16]=out2[17]=0.0;

    DCT12_PART1

    {
        real tmp0,tmp1 = (in0 - in4);
        {
            real tmp2 = REAL_MUL((in1 - in5), tfcos12[1]);
            tmp0 = tmp1 + tmp2;
            tmp1 -= tmp2;
        }
        out2[11-1] = REAL_MUL(tmp0, wi[11-1]);
```

```
            out2[6 +1] = REAL_MUL(tmp0, wi[6+1]);
            out2[0+1] += REAL_MUL(tmp1, wi[1]);
            out2[5-1] += REAL_MUL(tmp1, wi[5-1]);
        }

        DCT12_PART2

        out2[11-0] = REAL_MUL(in2, wi[11-0]);
        out2[6 +0] = REAL_MUL(in2, wi[6+0]);
        out2[6 +2] = REAL_MUL(in3, wi[6+2]);
        out2[11-2] = REAL_MUL(in3, wi[11-2]);

        out2[0+0] += REAL_MUL(in0, wi[0]);
        out2[5-0] += REAL_MUL(in0, wi[5-0]);
        out2[0+2] += REAL_MUL(in4, wi[2]);
        out2[5-2] += REAL_MUL(in4, wi[5-2]);
    }
}


static int Get_scale_factors(header *fr, int *scf,struct gr_info_s *gr_info,int ch,int gr)
{
    const unsigned char slen[2][16] =
    {
        {0, 0, 0, 0, 3, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4},
        {0, 1, 2, 3, 0, 1, 2, 3, 1, 2, 3, 1, 2, 3, 2, 3}
    };
    int numbits;
    int num0 = slen[0][gr_info->scalefac_compress];
    int num1 = slen[1][gr_info->scalefac_compress];

    if(gr_info->block_type == 2)
    {
        int i=18;
        numbits = (num0 + num1) * 18;

        if(gr_info->mixed_block_flag)
        {
            for (i=8;i;i--)
            *scf++ = getbits_fast(fr, num0);

            i = 9;
            numbits -= num0; /* num0 * 17 + num1 * 18 */
        }
    for(;i;i--) *scf++ = getbits_fast(fr, num0);

    for(i = 18; i; i--) *scf++ = getbits_fast(fr, num1);

    *scf++ = 0; *scf++ = 0; *scf++ = 0; /* short[13][0..2] = 0 */
}
else
{
    int i;
    int scfsi = gr_info->scfsi;

    if(scfsi < 0)
    { /* scfsi < 0 => granule == 0 */
        for(i=11;i;i--) *scf++ = getbits_fast(fr, num0);

        for(i=10;i;i--) *scf++ = getbits_fast(fr, num1);

        numbits = (num0 + num1) * 10 + num0;
        *scf++ = 0;
    }
    else
    {
        numbits = 0;
        if(!(scfsi & 0x8))
        {
            for (i=0;i<6;i++) *scf++ = getbits_fast(fr, num0);

            numbits += num0 * 6;
        }
        else scf += 6;

        if(!(scfsi & 0x4))
        {
            for (i=0;i<5;i++) *scf++ = getbits_fast(fr, num0);

            numbits += num0 * 5;
        }
        else scf += 5;

        if(!(scfsi & 0x2))
        {
            for(i=0;i<5;i++) *scf++ = getbits_fast(fr, num1);
            numbits += num1 * 5;
        }
        else scf += 5;

        if(!(scfsi & 0x1))
        {
            for (i=0;i<5;i++) *scf++ = getbits_fast(fr, num1);

            numbits += num1 * 5;
        }
        else scf += 5;

        *scf++ = 0;  /* no l[21] in original sources */
      }
  }
  //printf("\nGet_scale_factors numbits = %d\n",numbits);
  return numbits;
}
static const int pretab1[22] = {0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,2,2,3,3,3,2,0};
static const int pretab2[22] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
/*
  Dequantize samples
  ...includes Huffman decoding
*/

/* 24 is enough because tab13 has max. a 19 bit huffvector */
#define BITSHIFT ((sizeof(long)-1)*8)
#define REFRESH_MASK \
  while(num < BITSHIFT) { \
      mask |= ((unsigned long)getbyte(fr))<<(BITSHIFT-num); \
      num += 8; \
      part2remain -= 8; }

static int Dequantize_sample(header *fr, real xr[SBLIMIT][SSLIMIT],int *scf, struct gr_info_s *gr_info,int
sfreq,int part2bits)
{
  int shift = 1 + gr_info->scalefac_scale;
  real *xrpnt = (real *) xr;
  int l[3],l3;
  int part2remain = gr_info->part2_3_length - part2bits;
  int *me;

  /* mhipp tree has this split up a bit... */
  int num=getbitoffset(fr);
```

```
    long mask;
    /* We must split this, because for num==0 the shift is undefined if you do it in one step. */
    mask  = ((unsigned long) getbits(fr, num))<<BITSHIFT;
    mask <<= 8-num;
    part2remain -= num;

//  printf("\nDequantize_sample 1 \n");
//  printf("table_select[2] -> %u, maxband[0] =      %u\n",gr_info->table_select[2],gr_info->maxband[0]);
//  printf("maxband[1] ->      %u, maxband[2] =      %u\n",gr_info->maxband[1],gr_info->maxband[2]);
//  printf("maxbandl ->        %u, maxb =            %u\n",gr_info->maxbandl,gr_info->maxb);


    {
        int bv       = gr_info->big_values;
        int region1  = gr_info->region1start;
        int region2  = gr_info->region2start;
        if(region1 > region2)
        {
            /*
                That's not optimal: it fixes a segfault with fuzzed data, but also apparently triggers where it shouldn't,
see bug 1641196.
                The benefit of not crashing / having this security risk is bigger than these few frames of a lame-3.70
file that aren't audible anyway.
                But still, I want to know if indeed this check or the old lame is at fault.
            */
//          error("You got some really nasty file there... region1>region2!");
            return 1;
        }
        l3 = ((576>>1)-bv)>>1;

        /* we may lose the 'odd' bit here !! check this later again */
        if(bv <= region1)
        {
            l[0] = bv;
            l[1] = 0;
            l[2] = 0;
        }
        else
        {
            l[0] = region1;
            if(bv <= region2)
            {
                l[1] = bv - l[0];
                l[2] = 0;
        }
        else
        {
           l[1] = region2 - l[0];
           l[2] = bv - region2;
        }
     }
  }
  //printf("l 1-2 = %d %d %d , l3 = %d\n",l[0],l[1],l[2],l3);
  if(gr_info->block_type == 2)
  {
     /* decoding with short or mixed mode BandIndex table */
     int i,max[4];
     int step=0,lwin=3,cb=0;
     register real v = 0.0;
     register int *m,mc;

     if(gr_info->mixed_block_flag)
     {
        max[3] = -1;
        max[0] = max[1] = max[2] = 2;
        m = map[sfreq][0];
        me = mapend[sfreq][0];
     }
     else
     {
        max[0] = max[1] = max[2] = max[3] = -1;
        /* max[3] not really needed in this case */
        m = map[sfreq][1];
        me = mapend[sfreq][1];
     }

     mc = 0;
     for(i=0;i<2;i++)
     {
        int lp = l[i];
        struct newhuff *h = ht+gr_info->table_select[i];
        for(;lp;lp--,mc--)
        {
           register int x,y;
           if( (!mc) )
           {
              mc    = *m++;
              xrpnt = ((real *) xr) + (*m++);
```

```
    lwin  = *m++;
    cb    = *m++;
    if(lwin == 3)
    {
       v = gr_info->pow2gain[(*scf++) << shift];
       step = 1;
    }
    else
    {
       v = gr_info->full_gain[lwin][(*scf++) << shift];
       step = 3;
    }
 }
 {
    register short *val = h->table;
    REFRESH_MASK;
    while((y=*val++)<0)
    {
       if (mask < 0) val -= y;

       num--;
       mask <<= 1;
    }
    x = y >> 4;
    y &= 0xf;
 }
 if(x == 15 && h->linbits)
 {
    max[lwin] = cb;
    REFRESH_MASK;
    x += ((unsigned long) mask) >> (BITSHIFT+8-h->linbits);
    num -= h->linbits+1;
    mask <<= h->linbits;
    if(mask < 0) *xrpnt = REAL_MUL(-ispow[x], v);
    else         *xrpnt = REAL_MUL( ispow[x], v);

    mask <<= 1;
 }
 else if(x)
 {
    max[lwin] = cb;
    if(mask < 0) *xrpnt = REAL_MUL(-ispow[x], v);
    else         *xrpnt = REAL_MUL( ispow[x], v);
```

```
            num--;
            mask <<= 1;
          }
          else *xrpnt = DOUBLE_TO_REAL(0.0);

          xrpnt += step;
          if(y == 15 && h->linbits)
          {
            max[lwin] = cb;
            REFRESH_MASK;
            y += ((unsigned long) mask) >> (BITSHIFT+8-h->linbits);
            num -= h->linbits+1;
            mask <<= h->linbits;
            if(mask < 0) *xrpnt = REAL_MUL(-ispow[y], v);
            else         *xrpnt = REAL_MUL( ispow[y], v);

            mask <<= 1;
          }
          else if(y)
          {
            max[lwin] = cb;
            if(mask < 0) *xrpnt = REAL_MUL(-ispow[y], v);
            else         *xrpnt = REAL_MUL( ispow[y], v);

            num--;
            mask <<= 1;
          }
          else *xrpnt = DOUBLE_TO_REAL(0.0);

          xrpnt += step;
        }
      }

      for(;l3 && (part2remain+num > 0);l3--)
      {
        struct newhuff* h;
        register short* val;
        register short a;
        /*
          This is only a humble hack to prevent a special segfault.
          More insight into the real workings is still needed.
          Especially why there are (valid?) files that make xrpnt exceed the array with 4 bytes without
segfaulting, more seems to be really bad, though.
        */
```

```
if(!(xrpnt < &xr[SBLIMIT][0]+5))
{
   printf("attempted xrpnt overflow (%p !< %p)", (void*) xrpnt, (void*) &xr[SBLIMIT][0]);
   return 2;
}
h = htc+gr_info->count1table_select;
val = h->table;

REFRESH_MASK;
while((a=*val++)<0)
{
   if(mask < 0) val -= a;

   num--;
   mask <<= 1;
}
if(part2remain+num <= 0)
{
   num -= part2remain+num;
   break;
}

for(i=0;i<4;i++)
{
   if(!(i & 1))
   {
      if(!mc)
      {
         mc = *m++;
         xrpnt = ((real *) xr) + (*m++);
         lwin = *m++;
         cb = *m++;
         if(lwin == 3)
         {
            v = gr_info->pow2gain[(*scf++) << shift];
            step = 1;
         }
         else
         {
            v = gr_info->full_gain[lwin][(*scf++) << shift];
            step = 3;
         }
      }
      mc--;
```

```
        }
        if( (a & (0x8>>i)) )
        {
          max[lwin] = cb;
          if(part2remain+num <= 0)
          break;

          if(mask < 0) *xrpnt = -v;
          else        *xrpnt =  v;

          num--;
          mask <<= 1;
        }
        else *xrpnt = DOUBLE_TO_REAL(0.0);

        xrpnt += step;
      }
    }

    if(lwin < 3)
    { /* short band? */
      while(1)
      {
        for(;mc > 0;mc--)
        {
          *xrpnt = DOUBLE_TO_REAL(0.0); xrpnt += 3; /* short band -> step=3 */
          *xrpnt = DOUBLE_TO_REAL(0.0); xrpnt += 3;
        }
        if(m >= me)
        break;

        mc    = *m++;
        xrpnt = ((real *) xr) + *m++;
        if(*m++ == 0)
        break; /* optimize: field will be set to zero at the end of the function */

        m++; /* cb */
      }
    }

    gr_info->maxband[0] = max[0]+1;
    gr_info->maxband[1] = max[1]+1;
    gr_info->maxband[2] = max[2]+1;
    gr_info->maxbandl   = max[3]+1;
    {
      int rmax = max[0] > max[1] ? max[0] : max[1];
      rmax = (rmax > max[2] ? rmax : max[2]) + 1;
      gr_info->maxb = rmax ? fr->shortLimit[sfreq][rmax] : fr->longLimit[sfreq][max[3]+1];
    }

  }
  else
  {
    /* decoding with 'long' BandIndex table (block_type != 2) */
    const int *pretab = gr_info->preflag ? pretab1 : pretab2;
    int i,max = -1;
    int cb = 0;
    int *m = map[sfreq][2];
    register real v = 0.0;
    int mc = 0;

    /* long hash table values */
    for(i=0;i<3;i++)
    {
      int lp = l[i];
      struct newhuff *h = ht+gr_info->table_select[i];

      for(;lp;lp--,mc--)
      {
        int x,y;
        if(!mc)
        {
          mc = *m++;
          cb = *m++;
          if(cb == 21)
            v = 0.0;
          else
            v = gr_info->pow2gain[((*scf++) + (*pretab++)) << shift];

        }
        {
          register short *val = h->table;
          REFRESH_MASK;
          while((y=*val++)<0)
          {
            if (mask < 0) val -= y;
            num--;
            mask <<= 1;
          }
          x = y >> 4;
          y &= 0xf;
        }

        if(x == 15 && h->linbits)
        {
          max = cb;
          REFRESH_MASK;
          x += ((unsigned long) mask) >> (BITSHIFT+8-h->linbits);
          num -= h->linbits+1;
          mask <<= h->linbits;
          if(mask < 0) *xrpnt++ = REAL_MUL(-ispow[x], v);
          else         *xrpnt++ = REAL_MUL( ispow[x], v);

          mask <<= 1;
        }
        else if(x)
        {
          max = cb;
          if(mask < 0) *xrpnt++ = REAL_MUL(-ispow[x], v);
          else         *xrpnt++ = REAL_MUL( ispow[x], v);
          num--;

          mask <<= 1;
        }
        else *xrpnt++ = DOUBLE_TO_REAL(0.0);

        if(y == 15 && h->linbits)
        {
          max = cb;
          REFRESH_MASK;
          y += ((unsigned long) mask) >> (BITSHIFT+8-h->linbits);
          num -= h->linbits+1;
          mask <<= h->linbits;
          if(mask < 0) *xrpnt++ = REAL_MUL(-ispow[y], v);
          else         *xrpnt++ = REAL_MUL(ispow[y], v);

          mask <<= 1;
        }
        else if(y)
        {
```

```
          max = cb;
          if(mask < 0) *xrpnt++ = REAL_MUL(-ispow[y], v);
          else        *xrpnt++ = REAL_MUL(ispow[y], v);

          num--;
          mask <<= 1;
        }
        else *xrpnt++ = DOUBLE_TO_REAL(0.0);
      }
    }

    /* short (count1table) values */
    for(;l3 && (part2remain+num > 0);l3--)
    {
      struct newhuff *h = htc+gr_info->count1table_select;
      register short *val = h->table,a;

      REFRESH_MASK;
      while((a=*val++)<0)
      {
        if (mask < 0) val -= a;

        num--;
        mask <<= 1;
      }
      if(part2remain+num <= 0)
      {
        num -= part2remain+num;
        break;
      }

      for(i=0;i<4;i++)
      {
        if(!(i & 1))
        {
          if(!mc)
          {
            mc = *m++;
            cb = *m++;
            if(cb == 21) v = 0.0;
            else v = gr_info->pow2gain[((*scf++) + (*pretab++)) << shift];
          }
          mc--;
        }
```

```
          if( (a & (0x8>>i)) )
          {
            max = cb;
            if(part2remain+num <= 0)
            break;

            if(mask < 0) *xrpnt++ = -v;
            else        *xrpnt++ =  v;

            num--;
            mask <<= 1;
          }
          else *xrpnt++ = DOUBLE_TO_REAL(0.0);
        }
      }

      gr_info->maxbandl = max+1;
      gr_info->maxb = fr->longLimit[sfreq][gr_info->maxbandl];
  }

//  printf("\nDequantize_sample 2\n");
//  printf("table_select[2] -> %u, maxband[0] =      %u\n",gr_info->table_select[2],gr_info->maxband[0]);
//  printf("maxband[1] ->      %u, maxband[2] =      %u\n",gr_info->maxband[1],gr_info->maxband[2]);
//  printf("maxbandl ->        %u, maxb =            %u\n",gr_info->maxbandl,gr_info->maxb);



    part2remain += num;
    backbits(fr, num);
    num = 0;

    while(xrpnt < &xr[SBLIMIT][0])
    *xrpnt++ = DOUBLE_TO_REAL(0.0);

    while( part2remain > 16 )
    {
      skipbits(fr, 16); /* Dismiss stuffing Bits */
      part2remain -= 16;
    }
    if(part2remain > 0)
      skipbits(fr, part2remain);
    else if(part2remain < 0)
    {
      //printf("Can't rewind stream by %d bits!",-part2remain);
```

```
        return 1; /* -> error */
    }
    return 0;
}

void set_pointer(header *hptr,unsigned backstep, int cur_datasize,unsigned char *cur_data)
{
int index;
//int bytes_read;
//unsigned temp[800];

static unsigned count = 0,i;
    index = (hptr->buffersize - backstep);
 if(count <200)
{
    //printf("%d -> backstep -> %d wordpointer -> %d, Index = %d, Curdata size = %d\n",backstep,backstep,hptr-
>wordpointer[0],index,cur_datasize);
    count ++;
}
    if(index <= 0) //something is wrong just copy this data
    {
        hptr->bitindex = 0;
        hptr->buffersize = cur_datasize;
        for(i=0;i<cur_datasize;i++)
            hptr->parsebuffer[i] = cur_data[i];
        //memcpy(&(hptr->parsebuffer[0]),cur_data,cur_datasize);
        hptr->wordpointer = hptr->parsebuffer ;
    }
    else
    {
        //memcpy(temp,&(hptr->parsebuffer[index]),backstep);
        //memcpy(&(hptr->parsebuffer[0]),temp,backstep);

        for(i=0;i<backstep;i++)
            hptr->parsebuffer[i] = hptr->parsebuffer[i+index];

        hptr->bitindex = 0;
        hptr->buffersize = backstep;
        //memcpy(&(hptr->parsebuffer[backstep]),cur_data,cur_datasize);

        for(i=0;i<cur_datasize;i++)
            hptr->parsebuffer[backstep+i] = cur_data[i];

        hptr->buffersize += cur_datasize;
        hptr->wordpointer = hptr->parsebuffer ;
    }
//        for(i=0;i<cur_datasize;i++)
//            printf("%x ",cur_data[i]);

       //printf("\n");
}



int Get_side_info(header *hptr,int mode,unsigned char *sbuff,int ms_stereo,sideinfo *si)
{

    //struct gr_info_s *gr_ptr;
    int ch,gr;
    struct gr_info_s *gr_info;



    hptr->wordpointer = sbuff;
    hptr->bitindex = 0;

    si->main_data_begin = getbits(hptr,9);

    //printf("here I am!!   %d\n",si->main_data_begin);

    if(MPG_MD_MONO == mode)
        si->private_bits = getbits(hptr,5);
    else
        si->private_bits = getbits(hptr,3);

    for(ch=0; ch<(hptr->stereo); ch++)
    {
        si->ch[ch].gr[0].scfsi = -1;
        si->ch[ch].gr[1].scfsi = getbits(hptr, 4);
    }

    //printf("111\n");


    for (gr = 0; gr < 2; gr++)
    for (ch = 0; ch < (hptr->stereo); ch++)
    {
        //printf("[%d][%d]\n",gr,ch);
```

```
gr_info = &(si->ch[ch].gr[gr]);

gr_info->part2_3_length = getbits(hptr, 12);
gr_info->big_values = getbits(hptr, 9);
if(gr_info->big_values > 288)
{
   printf("big_values too large!");
   gr_info->big_values = 288;
}
gr_info->pow2gain = hptr->gainpow2+256 - getbits_fast(hptr, 8);
if(ms_stereo)
   gr_info->pow2gain += 2;

gr_info->scalefac_compress = getbits(hptr, 4);
//printf("222\n");
if(get1bit(hptr))
{ /* window switch flag  */
   //printf("333\n");
   int i;
   gr_info->block_type       = getbits_fast(hptr, 2);
   //printf("block_type %u\n",gr_info->block_type);
   gr_info->mixed_block_flag = get1bit(hptr);
   gr_info->table_select[0]  = getbits_fast(hptr, 5);
   gr_info->table_select[1]  = getbits_fast(hptr, 5);
   /*
      table_select[2] not needed, because there is no region2,
      but to satisfy some verification tools we set it either.
   */
   gr_info->table_select[2] = 0;

   for(i=0;i<3;i++)
   {
      //printf("%d\n",i);
      gr_info->full_gain[i] = gr_info->pow2gain + (getbits_fast(hptr, 3)<<3);
      //printf("%d\n",i);
   }



   if(gr_info->block_type == 0)
   {
      printf("Blocktype == 0 and window-switching == 1 not allowed.\n");
      return 1;
   }
```

```
            /* region_count/start parameters are implicit in this case. */

            //printf("aaa\n");
                gr_info->region1start = 36>>1;
                gr_info->region2start = 576>>1;
                //printf("555\n");

        }
        else
        {
            //printf("333\n");
            int i,r0c,r1c;
            for (i=0; i<3; i++)
            gr_info->table_select[i] = getbits_fast(hptr, 5);

            r0c = getbits_fast(hptr, 4);
            //printf("444\n");
            r1c = getbits_fast(hptr, 3);
            //printf("555\n");
            gr_info->region1start = bandInfo[0].longIdx[r0c+1] >> 1 ;
            gr_info->region2start = bandInfo[0].longIdx[r0c+1+r1c+1] >> 1;

            if(r0c + r1c + 2 > 22)
                gr_info->region2start = 576>>1;
            else
                gr_info->region2start = bandInfo[0].longIdx[r0c+1+r1c+1] >> 1;
            //printf("666\n");
            //printf("%d",sfreq);

            gr_info->block_type = 0;
            gr_info->mixed_block_flag = 0;
        }
        //printf("777\n");
        gr_info->preflag = get1bit(hptr);

        gr_info->scalefac_scale = get1bit(hptr);
        gr_info->count1table_select = get1bit(hptr);
    }
    //printf("end\n");
return 0;

}
```

```
void dct64(real *out0,real *out1,real *samples)
{
  real bufs[64];
//printf("our dct64");

  volatile i,j;
  register real *b1,*b2,*bs,*costab;
  volatile long val;
  static int flag = 0;

  b1 = samples;
  bs = bufs;
  costab = pnts[0]+16;
  b2 = b1 + 32;


//if(flag < 4)
{
   //printf("\n");
   //for(i=0;i<5;i++)
      //printf("%d, ",(real)samples[i]);
}


#if 0

for(i=0;i<32;i++)
   IOWR_DCT_DATA(DCT_RAM_DCT_BASE ,0, *samples++);


val = IORD_DCT_DATA(DCT_RAM_REG_BASE, 0);
//printf("%x   ", val);
while(!val)
{
   //printf("%x   ", val);
   val = IORD_DCT_DATA(DCT_RAM_REG_BASE, 0);
   //val++;
}

//printf("hello11");

for(i=0;i<32;i++)
   bufs[i] = IORD_DCT_DATA(DCT_RAM_DCT_BASE, 0);
```

```
#else
  for(i=15;i>=0;i--)
    *bs++ = (*b1++ + *--b2);
  for(i=15;i>=0;i--)
    *bs++ = REAL_MUL((*--b2 - *b1++), *--costab);

  b1 = bufs;
  costab = pnts[1]+8;
  b2 = b1 + 16;

  {
    for(i=7;i>=0;i--)
      *bs++ = (*b1++ + *--b2);
    for(i=7;i>=0;i--)
      *bs++ = REAL_MUL((*--b2 - *b1++), *--costab);
    b2 += 32;
    costab += 8;
    for(i=7;i>=0;i--)
      *bs++ = (*b1++ + *--b2);
    for(i=7;i>=0;i--)
      *bs++ = REAL_MUL((*b1++ - *--b2), *--costab);
    b2 += 32;
  }

  bs = bufs;
  costab = pnts[2];
  b2 = b1 + 8;

  for(j=2;j;j--)
  {
    for(i=3;i>=0;i--)
      *bs++ = (*b1++ + *--b2);
    for(i=3;i>=0;i--)
      *bs++ = REAL_MUL((*--b2 - *b1++), costab[i]);
    b2 += 16;
    for(i=3;i>=0;i--)
      *bs++ = (*b1++ + *--b2);
    for(i=3;i>=0;i--)
      *bs++ = REAL_MUL((*b1++ - *--b2), costab[i]);
    b2 += 16;
  }
  b1 = bufs;
  costab = pnts[3];
  b2 = b1 + 4;

  for(j=4;j;j--)
  {
    *bs++ = (*b1++ + *--b2);
    *bs++ = (*b1++ + *--b2);
    *bs++ = REAL_MUL((*--b2 - *b1++), costab[1]);
    *bs++ = REAL_MUL((*--b2 - *b1++), costab[0]);
    b2 += 8;
    *bs++ = (*b1++ + *--b2);
    *bs++ = (*b1++ + *--b2);
    *bs++ = REAL_MUL((*b1++ - *--b2), costab[1]);
    *bs++ = REAL_MUL((*b1++ - *--b2), costab[0]);
    b2 += 8;
  }
  bs = bufs;
  costab = pnts[4];

  for(j=8;j;j--)
  {
    real v0,v1;
    v0=*b1++; v1 = *b1++;
    *bs++ = (v0 + v1);
    *bs++ = REAL_MUL((v0 - v1), (*costab));
    v0=*b1++; v1 = *b1++;
    *bs++ = (v0 + v1);
    *bs++ = REAL_MUL((v1 - v0), (*costab));
  }

 #endif
//if(flag < 4)
{
   //printf("\n");
   //for(i=0;i<32;i++)
       //printf("%d, ",(real)bufs[i]);

   //flag++;
}

 {
  real *b1;
  int i;
 for(b1=bufs,i=8;i;i--,b1+=4)
   b1[2] += b1[3];

 for(b1=bufs,i=4;i;i--,b1+=8)
 {
   b1[4] += b1[6];
   b1[6] += b1[5];
   b1[5] += b1[7];
 }

 for(b1=bufs,i=2;i;i--,b1+=16)
 {
   b1[8]  += b1[12];
   b1[12] += b1[10];
   b1[10] += b1[14];
   b1[14] += b1[9];
   b1[9]  += b1[13];
   b1[13] += b1[11];
   b1[11] += b1[15];
 }
}


 out0[0x10*16] = bufs[0];
 out0[0x10*15] = bufs[16+0]  + bufs[16+8];
 out0[0x10*14] = bufs[8];
 out0[0x10*13] = bufs[16+8]  + bufs[16+4];
 out0[0x10*12] = bufs[4];
 out0[0x10*11] = bufs[16+4]  + bufs[16+12];
 out0[0x10*10] = bufs[12];
 out0[0x10* 9] = bufs[16+12] + bufs[16+2];
 out0[0x10* 8] = bufs[2];
 out0[0x10* 7] = bufs[16+2]  + bufs[16+10];
 out0[0x10* 6] = bufs[10];
 out0[0x10* 5] = bufs[16+10] + bufs[16+6];
 out0[0x10* 4] = bufs[6];
 out0[0x10* 3] = bufs[16+6]  + bufs[16+14];
 out0[0x10* 2] = bufs[14];
 out0[0x10* 1] = bufs[16+14] + bufs[16+1];
 out0[0x10* 0] = bufs[1];

 out1[0x10* 0] = bufs[1];
 out1[0x10* 1] = bufs[16+1]  + bufs[16+9];
```

```
  out1[0x10* 2] = bufs[9];
  out1[0x10* 3] = bufs[16+9]  + bufs[16+5];
  out1[0x10* 4] = bufs[5];
  out1[0x10* 5] = bufs[16+5]  + bufs[16+13];
  out1[0x10* 6] = bufs[13];
  out1[0x10* 7] = bufs[16+13] + bufs[16+3];
  out1[0x10* 8] = bufs[3];
  out1[0x10* 9] = bufs[16+3]  + bufs[16+11];
  out1[0x10*10] = bufs[11];
  out1[0x10*11] = bufs[16+11] + bufs[16+7];
  out1[0x10*12] = bufs[7];
  out1[0x10*13] = bufs[16+7]  + bufs[16+15];
  out1[0x10*14] = bufs[15];
  out1[0x10*15] = bufs[16+15];
}



void init_stuff(header *fr)
{
   int i,j;
real gainpow2[256+118+4] =
{
   94906265, 79806338, 67108864, 56431603, 47453132, 39903169, 33554432, 28215801, 23726566,
19951584, 16777216, 14107900, 11863283, 9975792, 8388608,
   7053950, 5931641, 4987896, 4194304, 3526975, 2965820, 2493948, 2097152, 1763487, 1482910, 1246974,
1048576, 881743, 741455, 623487,
   524288, 440871, 370727, 311743, 262144, 220435, 185363, 155871, 131072, 110217, 92681, 77935, 65536,
55108, 46340,
   38967, 32768, 27554, 23170, 19483, 16384, 13777, 11585, 9741, 8192, 6888, 5792, 4870, 4096, 3444,
   2896, 2435, 2048, 1722, 1448, 1217, 1024, 861, 724, 608, 512, 430, 362, 304, 256,
   215, 181, 152, 128, 107, 90, 76, 64, 53, 45, 38, 32, 26, 22, 19,
   16, 13, 11, 9, 8, 6, 5, 4, 4, 3, 2, 2, 2, 1, 1,
   1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0
};

   for(i=0;i<(256+118+4);i++)
      fr->gainpow2[i] = gainpow2[i];

   for(j=0;j<9;j++)
   {
      for(i=0;i<23;i++)
      {
         fr->longLimit[j][i] = (bandInfo[j].longIdx[i] - 1 + 8) / 18 + 1;
         //if(fr->longLimit[j][i] > (fr->down_sample_sblimit) )
         if(fr->longLimit[j][i] > SBLIMIT )  //hardcode to 32 as downsample freq is equal to the sampling freq. aq
seen by me ... has to be changed in future
            fr->longLimit[j][i] = SBLIMIT;
      }
      for(i=0;i<14;i++)
      {
         fr->shortLimit[j][i] = (bandInfo[j].shortIdx[i] - 1) / 18 + 1;
         //if(fr->shortLimit[j][i] > (fr->down_sample_sblimit) )
         if(fr->shortLimit[j][i] > SBLIMIT ) //hardcode to 32 as downsample freq is equal to the sampling freq. aq
seen by me ... has to be changed in future
            fr->shortLimit[j][i] = SBLIMIT;
      }
   }
}

void make_decode_tables(header *fr)
{

  int i,j;
  int idx = 0;
  /* Scale is always based on 1.0 . */
  double scaleval = -0.5; //fixing the scale value to be -0.5... TODO see how it changes with songs.. i think its just
volume control stuff...
  //printf("make_decode_tables  scaleval %g  ",scaleval);
  #if 1
```

```
 real decwin[] =
 {
 0, 475136, -3489792, 7520256, -33374208, 84426752, -107708416, 614219776, -1229422592, -614219776, -
107708416, -84426752, -33374208, -7520256, -3489792,
 -475136, 0, 475136, -3489792, 7520256, -33374208, 84426752, -107708416, 614219776, -1229422592, -
614219776, -107708416, -84426752, -33374208, -7520256,
 -3489792, -475136, 16384, 507904, -3571712, 8503296, -32768000, 90390528, -97632256, 644481024, -
1228668928, -583925760, -116883456, -78446592, -33800192,
 -6569984, -3407872, -425984, 16384, 507904, -3571712, 8503296, -32768000, 90390528, -97632256,
644481024, -1228668928, -583925760, -116883456, -78446592,
 -33800192, -6569984, -3407872, -425984, 16384, 573440, -3637248, 9519104, -31981568, 96321536, -
86638592, 674627584, -1226440704, -553631744, -125173760,
 -72499200, -34078720, -5685248, -3309568, -393216, 16384, 573440, -3637248, 9519104, -31981568,
96321536, -86638592, 674627584, -1226440704, -553631744,
 -125173760, -72499200, -34078720, -5685248, -3309568, -393216, 16384, 622592, -3686400, 10567680, -
31014912, 102187008, -74727424, 704610304, -1222737920,
 -523419648, -132579328, -66568192, -34193408, -4816896, -3211264, -344064, 16384, 622592, -3686400,
10567680, -31014912, 102187008, -74727424, 704610304,
 -1222737920, -523419648, -132579328, -66568192, -34193408, -4816896, -3211264, -344064, 16384,
671744, -3719168, 11649024, -29851648, 107954176, -61865984,
 734347264, -1217544192, -493355008, -139132928, -60702720, -34160640, -3997696, -3112960, -311296,
16384, 671744, -3719168, 11649024, -29851648, 107954176,
 -61865984, 734347264, -1217544192, -493355008, -139132928, -60702720, -34160640, -3997696, -3112960,
-311296, 16384, 737280, -3735552, 12763136, -28491776,
 113623040, -48087040, 763772928, -1210908672, -463486976, -144834560, -54902784, -33996800, -
3227648, -2998272, -278528, 16384, 737280, -3735552, 12763136,
 -28491776, 113623040, -48087040, 763772928, -1210908672, -463486976, -144834560, -54902784, -
33996800, -3227648, -2998272, -278528, 16384, 802816, -3735552,
 13893632, -26935296, 119128064, -33374208, 792821760, -1202831360, -433881088, -149733376, -
49217536, -33701888, -2506752, -2883584, -262144, 16384, 802816,
 -3735552, 13893632, -26935296, 119128064, -33374208, 792821760, -1202831360, -433881088, -
149733376, -49217536, -33701888, -2506752, -2883584, -262144, 32768,
 868352, -3719168, 15056896, -25149440, 124469248, -17727488, 821444608, -1193328640, -404586496, -
153829376, -43630592, -33292288, -1818624, -2768896, -229376,
 32768, 868352, -3719168, 15056896, -25149440, 124469248, -17727488, 821444608, -1193328640, -
404586496, -153829376, -43630592, -33292288, -1818624, -2768896,
 -229376, 32768, 950272, -3670016, 16236544, -23166976, 129597440, -1146880, 849559552, -1182416896, -
375668736, -157155328, -38174720, -32784384, -1179648,
 -2637824, -212992, 32768, 950272, -3670016, 16236544, -23166976, 129597440, -1146880, 849559552, -
1182416896, -375668736, -157155328, -38174720, -32784384,
 -1179648, -2637824, -212992, 32768, 1032192, -3620864, 17432576, -20971520, 134496256, 16351232,
877101056, -1170145280, -347160576, -159744000, -32866304,
 -32145408, -589824, -2523136, -180224, 32768, 1032192, -3620864, 17432576, -20971520, 134496256,
16351232, 877101056, -1170145280, -347160576, -159744000,
  -32866304, -32145408, -589824, -2523136, -180224, 32768, 1114112, -3522560, 18628608, -18530304,
139116544, 34766848, 904036352, -1156546560, -319127552,
  -161595392, -27721728, -31440896, -32768, -2408448, -163840, 32768, 1114112, -3522560, 18628608, -
18530304, 139116544, 34766848, 904036352, -1156546560,
  -319127552, -161595392, -27721728, -31440896, -32768, -2408448, -163840, 49152, 1196032, -3407872,
19824640, -15892480, 143441920, 54067200, 930250752,
  -1141620736, -291618816, -162775040, -22740992, -30638080, 475136, -2277376, -147456, 49152, 1196032,
-3407872, 19824640, -15892480, 143441920, 54067200,
  930250752, -1141620736, -291618816, -162775040, -22740992, -30638080, 475136, -2277376, -147456,
49152, 1294336, -3276800, 21020672, -13008896, 147423232,
  74268672, 955727872, -1125449728, -264683520, -163282944, -17940480, -29769728, 933888, -2162688, -
131072, 49152, 1294336, -3276800, 21020672, -13008896,
  147423232, 74268672, 955727872, -1125449728, -264683520, -163282944, -17940480, -29769728, 933888, -
2162688, -131072, 65536, 1392640, -3096576, 22216704,
  -9912320, 151044096, 95322112, 980385792, -1108033536, -238354432, -163168256, -13336576, -
28819456, 1359872, -2048000, -114688, 65536, 1392640, -3096576,
  22216704, -9912320, 151044096, 95322112, 980385792, -1108033536, -238354432, -163168256, -13336576,
-28819456, 1359872, -2048000, -114688, 65536, 1490944,
  -2899968, 23396352, -6586368, 154271744, 117211136, 1004158976, -1089437696, -212664320, -
162463744, -8929280, -27820032, 1736704, -1916928, -114688, 65536,
  1490944, -2899968, 23396352, -6586368, 154271744, 117211136, 1004158976, -1089437696, -212664320, -
162463744, -8929280, -27820032, 1736704, -1916928, -114688,
  81920, 1589248, -2670592, 24543232, -3031040, 157040640, 139919360, 1027014656, -1069711360, -
187678720, -161185792, -4718592, -26771456, 2080768, -1818624,
  -98304, 81920, 1589248, -2670592, 24543232, -3031040, 157040640, 139919360, 1027014656, -1069711360,
-187678720, -161185792, -4718592, -26771456, 2080768,
  -1818624, -98304, 81920, 1703936, -2392064, 25673728, 737280, 159367168, 163430400, 1048887296, -
1048887296, -163430400, -159367168, -737280, -25673728,
  2392064, -1703936, -81920, 81920, 1703936, -2392064, 25673728, 737280, 159367168, 163430400,
1048887296, -1048887296, -163430400, -159367168, -737280,
  -25673728, 2392064, -1703936, -81920,
  };
  //fr->decwin = decwin;

  for(i=0;i<(512+32);i++)
  {
     fr->decwin[i] = decwin[i];
  }

#else



  for(i=0,j=0;i<256;i++,j++,idx+=32)
```

```
 {
   if(idx < 512+16)
     fr->decwin[idx+16] = fr->decwin[idx] = DOUBLE_TO_REAL((double) intwinbase[j] * scaleval);

   if(i % 32 == 31)
     idx -= 1023;
   if(i % 64 == 63)
     scaleval = - scaleval;
 }

 for( /* i=256 */ ;i<512;i++,j--,idx+=32)
 {
   if(idx < 512+16)
     fr->decwin[idx+16] = fr->decwin[idx] = DOUBLE_TO_REAL((double) intwinbase[j] * scaleval);

   if(i % 32 == 31)
     idx -= 1023;
   if(i % 64 == 63)
     scaleval = - scaleval;
 }

printf("\n{\n");
for(i=0;i<(512+32);i++)
{
   printf("%ld, ",fr->decwin[i]);
   if((i+1) % 15 == 0)
       printf("\n");
}
printf("\n};\n");
#endif
}




void initbuffer(header *hptr)
{
   //unsigned count =0;
   int buffssize = 0;
   int decwin_size = (512+32)*sizeof(real);

   hptr->bitindex = 0;
   hptr->buffersize = 0;
   //memset(hptr->parsebuffer, 0, MAXFRAMESIZE + 512);
   //for(count=0;count <MAXFRAMESIZE + 512;count++)
     //  *(hptr->parsebuffer + count) = 0;

   buffssize = 2*2*0x110*sizeof(real);
   buffssize += 15;

   hptr->rawbuffs = (unsigned char*) malloc(buffssize);

   hptr->rawdecwin = (unsigned char*) malloc(decwin_size);
   hptr->decwin = (real*) hptr->rawdecwin;;

   hptr->real_buffs[0][0] = (real *)hptr->rawbuffs;
   hptr->real_buffs[0][1] = hptr->real_buffs[0][0] + 0x110;
   hptr->real_buffs[1][0] = hptr->real_buffs[0][1] + 0x110;
   hptr->real_buffs[1][1] = hptr->real_buffs[1][0] + 0x110;
}


void init(void)
{
   //printf("init    ");
   int i,j;

   for(j=0;j<9;j++)
   {
      const struct bandInfoStruct *bi = &bandInfo[j];
      int *mp;
      int cb,lwin;
      const int *bdf;

      mp = map[j][0] = mapbuf0[j];
      bdf = bi->longDiff;
      for(i=0,cb = 0; cb < 8 ; cb++,i+=*bdf++)
      {
         *mp++ = (*bdf) >> 1;
         *mp++ = i;
         *mp++ = 3;
         *mp++ = cb;
      }
      bdf = bi->shortDiff+3;
      for(cb=3;cb<13;cb++)
      {
         int l = (*bdf++) >> 1;
         for(lwin=0;lwin<3;lwin++)
```

```
        {
          *mp++ = l;
          *mp++ = i + lwin;
          *mp++ = lwin;
          *mp++ = cb;
        }
        i += 6*l;
      }
      mapend[j][0] = mp;

      mp = map[j][1] = mapbuf1[j];
      bdf = bi->shortDiff+0;
      for(i=0,cb=0;cb<13;cb++)
      {
        int l = (*bdf++) >> 1;
        for(lwin=0;lwin<3;lwin++)
        {
          *mp++ = l;
          *mp++ = i + lwin;
          *mp++ = lwin;
          *mp++ = cb;
        }
        i += 6*l;
      }
      mapend[j][1] = mp;

      mp = map[j][2] = mapbuf2[j];
      bdf = bi->longDiff;
      for(cb = 0; cb < 22 ; cb++)
      {
        *mp++ = (*bdf++) >> 1;
        *mp++ = cb;
      }
      mapend[j][2] = mp;
    }
}
```

Appendix B:

--VHDL implementation of IMDCT

```
ibrary ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dct is
port(
avs_dct_clk          : in std_logic;
    --avs_dct_reset_n     : in std_logic;
    avs_dct_read        : in std_logic;
    avs_dct_write       : in std_logic;
    avs_dct_chipselect  : in std_logic;
    avs_dct_address     : in unsigned(4 downto 0);
    avs_dct_readdata    : out std_logic_vector(31 downto 0);
    avs_dct_writedata   : in std_logic_vector(31 downto 0);

    data_dct_out        : out std_logic_vector(31 downto 0);
    over                : out std_logic;

avs_reg_read         : in std_logic;
avs_reg_chipselect   : in std_logic;
avs_reg_readdata     : out std_logic_vector(31 downto 0);
avs_reg_write        : in std_logic;
avs_reg_writedata    : in STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end dct;

architecture rtl of dct is

component ram_2_port
PORT
(
address_a    : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
address_b    : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
clock        : IN STD_LOGIC ;
data_a       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
data_b       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
wren_a       : IN STD_LOGIC;
wren_b       : IN STD_LOGIC;
q_a          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
q_b          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end component;
```

```
component add
PORT
(
clock        : IN STD_LOGIC ;
dataa        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
datab        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
result        : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end component;

component sub
PORT
(
clock        : IN STD_LOGIC ;
dataa        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
datab        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
result        : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end component;




component mul
PORT
(
clock        : IN STD_LOGIC ;
dataa        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
datab        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
result        : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
);
end component;

component delay_add
PORT
(
clock        : IN STD_LOGIC ;
dataa        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
result        : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end component;


--component shift_reg
```

```
--    PORT
--    (
--         clock         : IN STD_LOGIC ;
--         shiftin        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
--         shiftout        : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
--         taps         : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
--    );
--end component;


component cos_tab
PORT
(
address        : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
clock        : IN STD_LOGIC ;
q        : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
end component;


component add_1
PORT
(
address        : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
clock        : IN STD_LOGIC ;
q        : OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
);
end component;


component add_2
PORT
(
address        : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
clock        : IN STD_LOGIC ;
q        : OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
);
end component;

component one_zero
PORT
(
address        : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
clock        : IN STD_LOGIC ;
q        : OUT STD_LOGIC_VECTOR (0 DOWNTO 0)
);
end component;

signal stop_global_count:  std_logic:='0';
signal stop_count_c        :  std_logic:='0';
signal stop_count_g        :  std_logic:='0';
signal stop_count_w        :  std_logic:='0';
signal stop_count_r        :  std_logic:='0';
signal strt1_inv0        :  STD_LOGIC_vector(0 downto 0);
signal clock_sig           :  STD_LOGIC;
signal start               :  STD_LOGIC:='0';
signal over_sig            :  STD_LOGIC:='0';
signal switch              :  std_logic;
signal global_counter   :  unsigned(6 downto 0);
signal r_counter         :  unsigned(6 downto 0);
signal g_counter         :  unsigned(2 downto 0);
signal w_counter         :  unsigned(6 downto 0);
signal c_counter         :  unsigned(6 downto 0);
signal counter           :  unsigned(4 downto 0);
signal out_counter      :  unsigned(4 downto 0);
signal proc_in_counter  :  unsigned(4 downto 0);
signal switch_counter   :  unsigned(1 downto 0);
signal address_a_0        :  unsigned(4 DOWNTO 0);
signal address_b_0        :  unsigned (4 DOWNTO 0);
signal data_a_0            :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal data_b_0            :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal wren_a_0            :  STD_LOGIC;
signal wren_b_0            :  STD_LOGIC;
signal q_a_0            :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal q_b_0            :  STD_LOGIC_VECTOR (31 DOWNTO 0);

signal address_a_1        :  unsigned (4 DOWNTO 0);
signal address_b_1        :  unsigned (4 DOWNTO 0);
signal data_a_1            :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal data_b_1            :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal wren_a_1            :  STD_LOGIC ;
signal wren_b_1            :  STD_LOGIC;
signal q_a_1            :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal q_b_1            :  STD_LOGIC_VECTOR (31 DOWNTO 0);

signal data_a_add        :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal data_b_add        :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal result_add        :  STD_LOGIC_VECTOR (31 DOWNTO 0);

signal data_a_sub        :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal data_b_sub        :  STD_LOGIC_VECTOR (31 DOWNTO 0);
signal result_sub        :  STD_LOGIC_VECTOR (31 DOWNTO 0);

signal data_a_mul        : STD_LOGIC_VECTOR (31 DOWNTO 0);
signal data_b_mul        : STD_LOGIC_VECTOR (31 DOWNTO 0);
signal result_mul        : STD_LOGIC_VECTOR (63 DOWNTO 0);

signal data_a_delay        : STD_LOGIC_VECTOR (31 DOWNTO 0);
signal result_a_delay    : STD_LOGIC_VECTOR (31 DOWNTO 0);
--taps_sreg        : STD_LOGIC_VECTOR (31 DOWNTO 0);

signal address_costab    : unsigned (6 DOWNTO 0);
signal q_costab          : STD_LOGIC_VECTOR (31 DOWNTO 0);

signal address_add_1    : unsigned (6 DOWNTO 0);
signal q_add_1             : STD_LOGIC_VECTOR (4 DOWNTO 0);

signal address_add_2    : unsigned (6 DOWNTO 0);
signal q_add_2             : STD_LOGIC_VECTOR (4 DOWNTO 0);

signal address_add_1_o    : unsigned (6 DOWNTO 0);
signal q_add_1_o         : STD_LOGIC_VECTOR (4 DOWNTO 0);

signal address_add_2_o    : unsigned (6 DOWNTO 0);
signal q_add_2_o         : STD_LOGIC_VECTOR (4 DOWNTO 0);

signal address_one_zero    : UNSIGNED(6 DOWNTO 0);
SIGNAL q_ONE_ZERO        : STD_LOGIC_VECTOR (0 DOWNTO 0);

signal reg                  : STD_LOGIC_VECTOR (31 DOWNTO 0):=x"00000000";
begin



ram_2_port_inst0 : ram_2_port PORT MAP (
address_a=> STD_LOGIC_VECTOR(address_a_0),
address_b=> STD_LOGIC_VECTOR(address_b_0),
clock      => clock_sig,
data_a      => data_a_0,
data_b      => data_b_0,
wren_a      => wren_a_0,
wren_b      => wren_b_0,
q_a            => q_a_0,
q_b           => q_b_0
);
```

```
ram_2_port_inst1 : ram_2_port PORT MAP (
address_a=> STD_LOGIC_VECTOR(address_a_1),
address_b=> STD_LOGIC_VECTOR(address_b_1),
clock      => clock_sig,
data_a      => data_a_1,
data_b      => data_b_1,
wren_a      => wren_a_1,
wren_b      => wren_b_1,
q_a           => q_a_1,
q_b           => q_b_1
);


add_inst : add PORT MAP (
clock      => clock_sig,
dataa      => data_a_add,
datab      => data_b_add,
result      => result_add
);

sub_inst : sub PORT MAP (
clock      => clock_sig,
dataa      => data_a_sub,
datab      => data_b_sub,
result      => result_sub
);



mul_inst : mul PORT MAP (
clock      => clock_sig,
dataa      => data_a_mul,
datab      => data_b_mul,
result      => result_mul
);

--shift_reg_inst : shift_reg PORT MAP (
--          clock      => clock_sig,
--          shiftin      => shiftin_sreg,
--          shiftout => shiftout_sreg,
--          taps      => taps_sreg
--     );

delay_add_inst : delay_add PORT MAP (
clock      => clock_sig,
dataa      => data_a_delay,
result      => result_a_delay
);


cos_tab_inst : cos_tab PORT MAP (
address      => STD_LOGIC_VECTOR(address_costab),
clock      => clock_sig,
q          => (q_costab)
);


add_1_inst : add_1 PORT MAP (
address      => STD_LOGIC_VECTOR(address_add_1),
clock      => clock_sig,
q      => (q_add_1)
);

add_2_inst : add_2 PORT MAP (
address      => STD_LOGIC_VECTOR(address_add_2),
clock      => clock_sig,
q      => (q_add_2)
);

add_1_inst_o : add_1 PORT MAP (
address      => STD_LOGIC_VECTOR(address_add_1_o),
clock      => clock_sig,
q      => (q_add_1_o)
);

add_2_inst_o : add_2 PORT MAP (
address      => STD_LOGIC_VECTOR(address_add_2_o),
clock      => clock_sig,
q      => (q_add_2_o)
);

one_zero_inst : one_zero PORT MAP (
address      => STD_LOGIC_VECTOR(address_one_zero),
clock      => clock_sig,
q      => q_one_zero
);

clock_sig <= avs_dct_clk;
process(avs_dct_clk)
```

```
begin
if rising_edge(avs_dct_clk) then


    if(avs_reg_chipselect ='1')  then
      if avs_reg_read ='1' then
        avs_reg_readdata <= reg;
      elsif avs_reg_write ='1' then
        reg<=avs_reg_writedata;
      end if;
    end if;

  if start = '0' then
    if(avs_dct_chipselect = '1') then
       if(avs_dct_write = '1') then
            if counter < "11111" then
                counter <= counter +  1;
                address_a_0 <= counter;
                wren_a_0 <='1';
                data_a_0 <= avs_dct_writedata;
            else
                counter <= counter +  1;
                address_a_0 <= counter;
                wren_a_0 <='1';
                data_a_0 <= avs_dct_writedata;
                --wren_a_0 <='0';
                start <= '1';
                switch <= '0';
                reg <= x"00000000";
            end if;



       elsif(avs_dct_read = '1') then
             if proc_in_counter < "11111" then
                proc_in_counter <= proc_in_counter +  1;
                address_a_1 <= proc_in_counter;
                wren_a_1 <='0';
                avs_dct_readdata <= q_a_1;
            else
                proc_in_counter <= proc_in_counter +  1;
                address_a_1 <= proc_in_counter;
                wren_a_1 <='0';
                avs_dct_readdata <= q_a_1;
                --wren_a_1 <='1';
                reg <= x"00000000";
                --start <= '1';
            end if;

      end if;
    end if;
  end if;

if(start = '1') then
  if(switch = '0') then
    address_a_0<=unsigned(q_add_1);
    address_b_0<=unsigned(q_add_2);

    wren_a_0<='0';
    wren_b_0<='0';

--add
    data_a_add<=q_a_0;
    data_b_add<=q_b_0;

    strt1_inv0 <= q_one_zero(0 downto 0);

--sub
    if(strt1_inv0(0) = '1') then

        data_a_sub<=q_a_0;
        data_b_sub<=q_b_0; -- if (1-2)
    else
        data_a_sub<=q_b_0;
        data_b_sub<=q_a_0; -- if (2-1)
    end if;

--mul
    data_a_mul<=result_sub;
    data_b_mul<=q_costab;

--shift_reg
    data_a_delay<=result_add;

--storage

    address_a_1<=unsigned(q_add_1_o);
    address_b_1<=unsigned(q_add_2_o);

    wren_a_1<='1';
    wren_b_1<='1';

    data_a_1<=result_a_delay;
    data_b_1<=result_mul(46 downto 15);

  else

    address_a_1<=unsigned(q_add_1_o);
    address_b_1<=unsigned(q_add_2_o);

    wren_a_1<='0';
    wren_b_1<='0';

--add
    data_a_add<=q_a_1;
    data_b_add<=q_b_1;

--sub
    if(strt1_inv0(0) = '1') then

        data_a_sub<=q_a_1;
        data_b_sub<=q_b_1; -- if (1-2)
    else
        data_a_sub<=q_b_1;
        data_b_sub<=q_a_1; -- if (2-1)
    end if;

--mul
    data_a_mul<=result_sub;
    data_b_mul<=q_costab;

--shift_reg
    data_a_delay<=result_add;

--storage
    address_a_0<=unsigned(q_add_1);
    address_b_0<=unsigned(q_add_2);

    wren_a_0<='1';
    wren_b_0<='1';

    data_a_0<=result_a_delay;
    data_b_0<=result_mul(46 downto 15);
  end if;
--end if;
```

```
--if start = '1' then

    if g_counter = "100" then
      g_counter <= "000";
      reg <= x"00000001";
      over_sig <= '1';
      start <= '0';
     else
       over_sig <= '0';
       reg <= x"00000000";
    end if;

   if(stop_global_count = '0') then
    if(global_counter = "1010001") then
       global_counter <= "0000000";
       stop_global_count <= '1';
    else
       global_counter <= global_counter + 1;
    end if;
   end if;

    if switch = '1' then
      address_add_1_o<=r_counter;
      address_add_2_o<=r_counter;
      address_one_zero <=r_counter;
    elsif switch = '0' then
     address_add_1<=r_counter;
     address_add_2<=r_counter;
     address_one_zero <=r_counter;
   end if;

 if(stop_count_w = '0') then
  if(global_counter > "0000001" and w_counter < "1001111") then
      w_counter <= w_counter + 1;
      if switch = '0' then
         address_add_1_o<=w_counter;
         address_add_2_o<=w_counter;
     elsif switch = '1' then
        address_add_1<=w_counter;
        address_add_2<=w_counter;
     end if;
   end if;
  end if;
```

```
if(stop_count_c = '0') then
  if(global_counter > "0000000" and c_counter < "1001111") then
     c_counter <= c_counter +  1;
     address_costab <= c_counter;

  end if;
end if;

  if c_counter = "1001111" then
     c_counter <= "0000000";
     stop_count_c <= '1';
  end if;

  if w_counter = "1001111" then
     --w_counter <="0000000";
     stop_count_w <= '1';
     if switch_counter < "11" then
        switch_counter <= switch_counter +  1;
      elsif switch_counter = "11" then

          case g_counter is
              when "000" => switch <= '1';
              when "001" => switch <= '0';
              when "010" => switch <= '1';
              when "011" => switch <= '0';
              when others => switch <= '0';
              end case;
         switch_counter <= "00";
         g_counter <= g_counter +  1;
         stop_count_c <= '0';
         stop_global_count <= '0';
         stop_count_r <= '0';
         stop_count_c <= '0';
         stop_count_w <= '0';
         w_counter <= "0000000";
      end if;
   end if;


 if stop_count_r = '0' then
  if(r_counter = "1001111") then
      r_counter <= "0000000";
      stop_count_r <= '1';
   else
```

```
        r_counter <= r_counter +  1;
     end if;
    end if;

  end if;     --start
--end if;
end if;
end process;
end architecture;
```