18-551 Spring 2003
Group 4, Final Report
To Ogg, or Not to Ogg!

César Naranjo (can@andrew.cmu.edu)
Jason France (jof@andrew.cmu.edu)
Nathaniel Watterson (npw@andrew.cmu.edu)

Index

Introduction

Motivation

Portable digital audio is becoming ever popular in today's society. As a result, a growing number of people have purchased portable audio players to take advantage of its size, sound quality during playback, and ability to playback various audio files of the users choosing. Multiple forms of technology now offer support of such files such as portable music players, PDA's and even cell phones. The need for new forms of technological support remains constant; however the type of digital audio to be supported is in debate. Currently there are multiple forms of digital audio that are available to the public. Among those are MP3, Ogg Vorbis, Real Media files and Windows Media Audio files.

Today's technology industry drives for products that are more efficient in making use of given resources. Most arguments against earlier portable devices were that the given onboard memory was not enough to store a sufficient amount of digital audio. Later versions increased the amount of onboard memory to appease customers. However, a more efficient approach would have been to decrease the size that each digital audio file consumes.

Ogg Vorbis, though not as popular as MP3 audio, is to say the least comparable in quality. Ogg files can be compressed to smaller sized files and still have the sound quality that portable device owners desire.

Our motivation behind this project was to show that it is possible to port implementations of the Ogg Vorbis decoding algorithm from software to hardware. In doing so we hope to show a possibility that future portable devices can not only support Ogg Vorbis files but also convert MP3 audio files to Ogg Vorbis files on the fly. In doing so portable device owners will be able to store more audio files while maintaining the sound quality the desire. Once decoding on portable devices becomes a reality, the next step would be encoding for the transition to happen.

Background

Ogg Vorbis is a fully open, non-proprietary, patent-and-royalty-free, general-purpose compressed audio format for mid to high quality audio and music at fixed and variable bitrates from 16 to 128 kbps per channel. Vorbis is in the same competitive class as audio representations such as MPEG-4, and similar to but higher performance than MPEG-1/2 audio layer 3, MPEG-4 audio, WMA and PAC. Ogg Vorbis is different from these other formats because it is completely free, open, and unpatented. For a given file size, Vorbis sounds better than MP3. By converting audio files from MP3 to Ogg you can keep your music collection at about the same quality level and it will take up less space, or you can have your music collection take up about the same amount of space, but have it sound better.

Prior Work

In the scope of digital audio, the founders of the Ogg Vorbis technology, XIPH.ORG, have made progress. This organization created Ogg Vorbis in an attempt to give

musicians, music fans, and developers a form of source free digital audio that requires no patent for use. Currently portable devices such as the Sharp SL-5500 Linux PDA, Neuros Digital Audio Computer, and the Mac iPod support Ogg Vorbis audio files. Software such as AeroPlayer 2.1 and Ahead's Nero CD Creator also support Ogg Vorbis files through third party plug-ins.

As for previous 18-551 projects, two groups in recent years have proposed related projects. In the spring of 2000 group 14 proposed to implement an encoding algorithm for MP3 onto the evm. Differences in our project and theirs are the use of different audio formats as well as decoding versus encoding. The noticeable similarity is the use of modified discrete cosine transform, or in our case the inverse MDCT. There use of the MDCT was on the outputs of the filterbanks to finalize pure, non-perceptual lossy encoding by breaking up each sub-band into primary components and send them off to be compressed. Our use of the IMDCT was to convert the audio spectrum vector of each channel back into time domain PCM audio. Group 14 was unable to get their implementation fully functional on the evm, though they stated that in their view encoding was more computationally intense than decoding. Group 16 in the spring of 1999 proposed a similar project to group 14. Their encoding algorithm failed to function on the evm as well.

Algorithm

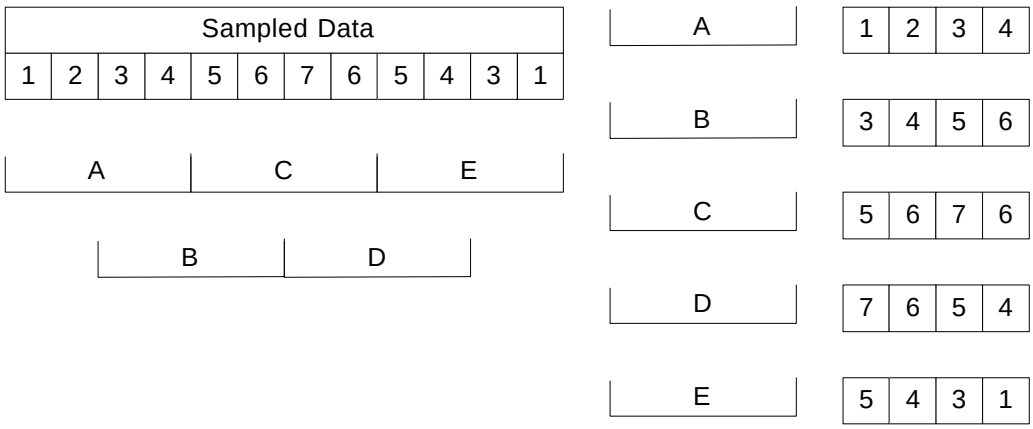
Decoding Algorithm Overview
To better understand decoding we will go through an overview of the encoding procedure knowing that these steps are reversible and that certain concepts can be expanded on from the this simple explanation. Below is the actual procedure used in decoding.

1. Decode window shape (long windows only)
2. Decode Floor
3. Decode residue into residues vectors
4. Inverse channel coupling of residue vectors
5. Generate Floor curve from decoded floor data
6. Computer dot product of floor and residue, producing audio spectrum vector
7. MDCT of audio spectrum vector
8. Reverse Windowing using previous frame for overlapping

Audio Frames
Audio data is split into several frames. Each frame contains a set number of PCM audio samples. The size of each frame is determined by the encoder. The decoder simply is told what each frame size is. Frame sizes are limited to powers of two and typically are 256 and 2048 samples long. This constraint is set because transforms and their inverses used in signal processing are usually optimized for such sizes (eg: FFT and FDFT).

Frame Overlapping
Each frame overlaps exactly 50% into the next frame as well as 50% into the previous frame. Below a simple example is shown. The length of the signal is 12 long and for simplicity each frame is 4 long. We split our data into frames A, B, C, D and E as shown.

In our example we went from 12 samples to 20 samples (number of frames x frame size) and obliviously we have a lot redundancy. We will only give a brief explanation for why this is done. If we were to continue from here and use only A, B and C as our frames then perform some transform and quantization in reconstruction quantization will cause inconsistency between frames. Framing in this fashion will smooth our error. In audio this is often manifested in clicks created at frame boundaries.



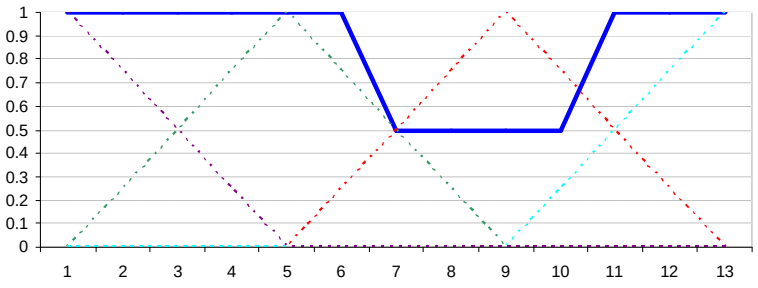
Windowing

To prevent this, we take frames as previously specified and at the same time perform windowing. You can visualize this as slowing passing information into the next frame. For example at samples 3 and 4 where A and B share samples; sample 3 would be weighed more in frame A and sample 4 more in frame B.

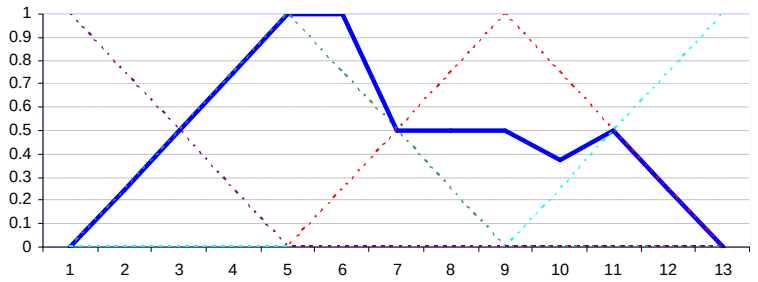
To better explain this we will use a simple example and continue through decoding. Take [I] as our input vector and we will window that signal using a simple triangle wave. Columns 4 and 5 show our sampled windowed into frames [A] and [B] respectively. Windows are of size 8.

Input [I]	Win A [Aw]	Win B [Bw]	[A] = [I] * [Aw]	[B] =[i] * [Bw]
1	0	0	0	0
1	0.25	0	0.25	0
1	0.5	0	0.5	0
1	0.75	0	0.75	0
1	1	0	1	0
1	0.75	0.25	0.75	0.25
0.5	0.5	0.5	0.25	0.25
0.5	0.25	0.75	0.125	0.375
0.5	0	1	0	0.5
0.5	0	0.75	0	0.375
1	0	0.5	0	0.5
1	0	0.25	0	0.25
1	0	0	0	0

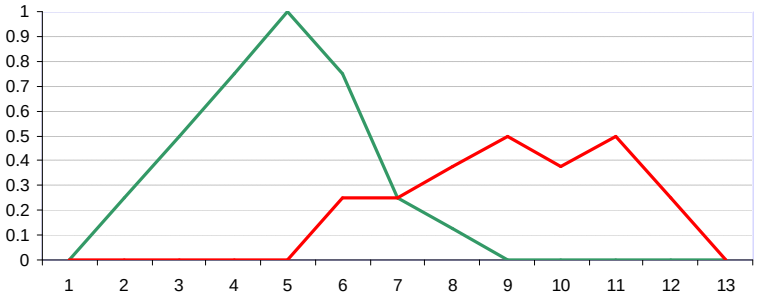
Blue is [I]



Window [A] combined with window [B] shown in blue; this is the data we will encode. Windowing function A [Aw] in light green. Windowing function B [Bw] shown in red.



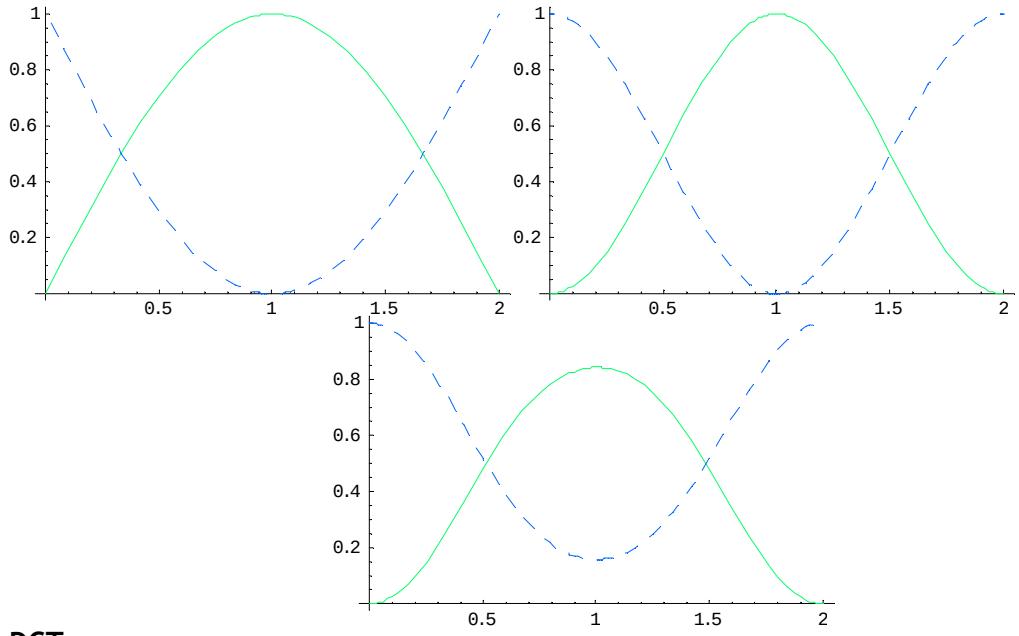
Window [A] in green. Window [B] in red.



Below are the graphs to the actual windowing functions used in Ogg Vorbis [1]. We would take frame A using sampled data from -1 to 1 using the blue dash window. Frame B would be taken from 0 to 2 using the green solid window. Frame C from 1 to 3 using the blue dashed window. It can be easily proven that the second windowing function will produce equivalent A, B and C windows.

[Mathematica]

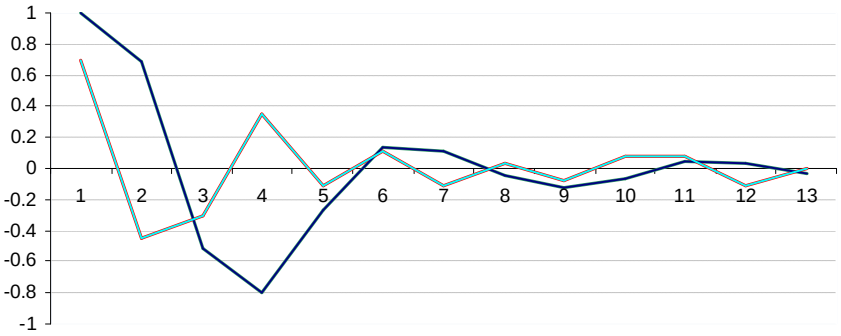
```
Plot[{Sin[x * π / 2], 1 - Sin[x * π / 2]}, {x, 0, 2},
    PlotStyle → {{Hue[.4]}, {Hue[.6], Dashing[{.03}]}}];
Plot[{Sin[x * π / 2]^2 , 1 - Sin[x * π / 2]^2}, {x, 0, 2},
    PlotStyle → {{Hue[.4]}, {Hue[.6], Dashing[{.03}]}}];
Plot[{Sin[Sin[x * π / 2]^2], 1 - Sin[Sin[x * π / 2]^2]}, {x, 0, 2},
    PlotStyle → {{Hue[.4]}, {Hue[.6], Dashing[{.03}]}}];
```



DCT

We now have our two windows. What we do now is perform the DCT on both these. For audio as in imaging DCT is highly energy compact; this is the reason why this transform is often used in audio compression. Below are the results of a DCT on [A] and [B]. Note for our explanation we used the zeros in our windows. Obviously, these zeros can be simply truncated off before performing transform in practice. DCTs were calculated using Matlab.

[Awin]	[Bwin]
1.0054	0.6934
0.6891	-0.4535
-0.5173	-0.3013
-0.7954	0.3496
-0.2667	-0.11
0.1345	0.1106
0.1164	-0.1144
-0.0463	0.036
-0.1283	-0.077
-0.0619	0.0751
0.048	0.0818
0.0352	-0.1109
-0.0314	0.0002



MDCT = windowing + DCT
The procedure we have just described is what is most commonly know as the Modified Discrete Cosine Transform:

$$y_t(p) = f(p)\frac{n}{4}\sum_{m=0}^{\frac{n}{2}} X_t(m)\cos(\frac{p}{2n}(2p+1+\frac{n}{2})(2m+1))$$

$$\text{for } p = 0...n-1$$

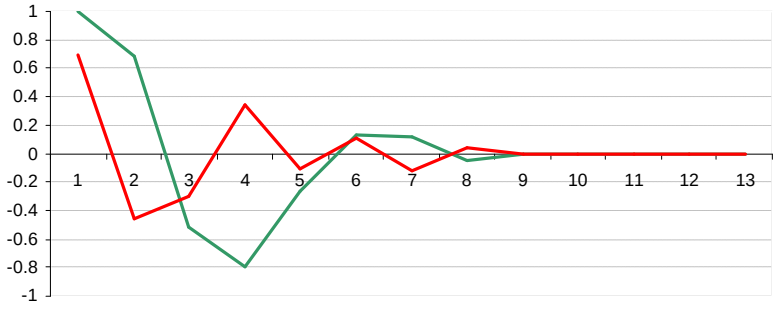
Where f(p) is simply our windowing function and to the right of it a regular DCT (almost; overlapping has been taken into consideration). This is the main concept in Ogg Vorbis.

Motivation for using MDCT

- Fast algorithms offer a low computational complexity; computational efficiency similar to an FFT. Low numbers of operations(+,*,storage operations).
- MDCT combines critical sampling with the good frequency provided by a sine window.
- MDCT offers the possibility to change the block length of the transform dynamically. Same idea as progressive decoding in JPEG.
- Minimum size of storage needed. Inplace routines.
- High robustness against rounding errors.

Quantization (Not reversible in decoding)
Though in reality the encoder uses psychoacoustic models to analysis and then very selectively quantize the spectral information held within [Awin] and [Bwin] the decoder is naïve to this. Also this would have been done to DCT-ed vector of size 8 in this case. In our example we will simply zero out the end of our DCT-ed windows.

[Awinq]	[Bwinq]
1.0054	0.6934
0.6891	-0.4535
-0.5173	-0.3013
-0.7954	0.3496
-0.2667	-0.11
0.1345	0.1106
0.1164	-0.1144
-0.0463	0.036
0	0
0	0
0	0
0	0
0	0



Storage
The quantized vectors generated through this process then encoded and placed into an Ogg Stream, as described earlier. This is done be generating two vectors; a floor vector this can be seen as a vector with just enough information to describe the general idea of what [Xwinq] should be then details are held within another residual vectors. Mathematically [Xwinq] can be regenerated by performing a dot product between [X residual] and [X floor]. Stereo coupling can also be performed. For example if a left channel is very similar to a right channel, it might be possible to generate a floor for both channels and keep the necessary details in two separate residual vectors.

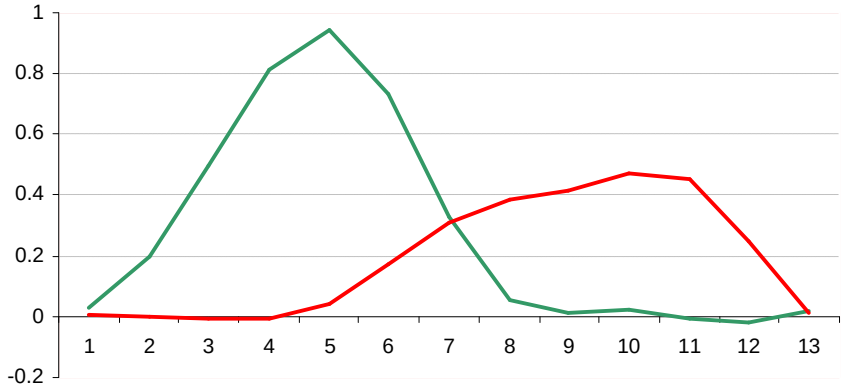
What should also be noted is that during decoder initialization frequency vectors are generated which detail what frequency values on the x axis are being encoded for each channel. For example if we had a bass channel that only ranged from 0Hz to 1Mz the frequency vector would scale our [X residual], [X floor] dot product into that range; this would give us higher resolution where we need most. Clearly, this can also be done for a tweeter channel. Another technique used is to make the frequency vectors non-linear; this makes sense because humans can perceive differences between certain frequencies. As you can see the encoder can be highly complex while the decoder simply follows a very simple routine oblivious to complex techniques used.

Compression is achieved because quantization produces vectors which have less information and when passed through an entropy encoder will be compressed.

Regeneration
To regenerate our signal all the previous steps excluding quantization are reversed. The following tables and figures will take you through process to show how similar our compressed/decompressed signal will look like the original.

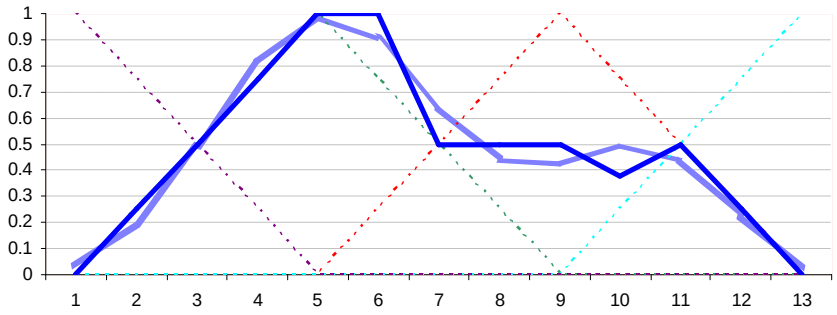
Shown in are IDCT of our earlier quantized DCT vectors.

[Aq]	[Bq]
0.0314	0.0025
0.1985	-0.0005
0.498	-0.0064
0.8156	-0.0052
0.9415	0.0445
0.7325	0.1709
0.3315	0.3122
0.0547	0.3848
0.0096	0.4178
0.0265	0.4703
-0.0084	0.4513
-0.0216	0.2487
0.0154	0.009



Shown is our regenerated signal (light blue) with our original signal (dark blue). They are very similar.

[lq]	[lw]
0.0339	0
0.198	0.25
0.4916	0.5
0.8104	0.75
0.986	1
0.9034	1
0.6437	0.5
0.4395	0.5
0.4274	0.5
0.4968	0.375
0.4429	0.5
0.2271	0.25
0.0244	0



Implementation

PC<->EVM Communication

We decided to use two groups of two buffers along with five flags/mailboxes to design our pc/evm communication. These four buffers and five flags were located in onchip memory. The buffers were setup in a way such that data could be read or written to one buffer while not interrupting data stored in other buffers. The first two buffers were designated for the pc to write to and the evm to read from. The last two buffers are for the evm to write to and the pc to read from. Each mailbox tells the pc and evm to perform a specific function based on the value stored at its memory address. These functions consist of write data to a certain buffer, read data from a certain buffer, wait for the evm/pc, tell the evm/pc your done, and tell the pc decoding is done. The evm drives the communication, as it determines what needs to be done next whether is be decode data or to receive more data. The pc is idle until it receives commands from the evm.

Note: Further mention of flags/mailboxes will be addressed as flag. Buffers 1 and 2 are evm reads, pc writes. Likewise, buffers 3 and 4 are evm write, pc reads.

Step 1: Get some data!

Initially the pc is waiting for the evm to set a flag so it knows what function to perform. The evm sets a flag telling the pc to send data to buffer 1 and to set a flag once it is done for the evm. Once the pc has finished writing data to buffer 1, it sets a flag letting the evm know it is done. The evm then sets a flag telling the pc to set data to buffer 2. As before, the evm will wait until the pc sets a flag confirming that data has been sent to buffer 2. When the pc is done sending data to buffer 2 it sets a flag and then waits for the next command.

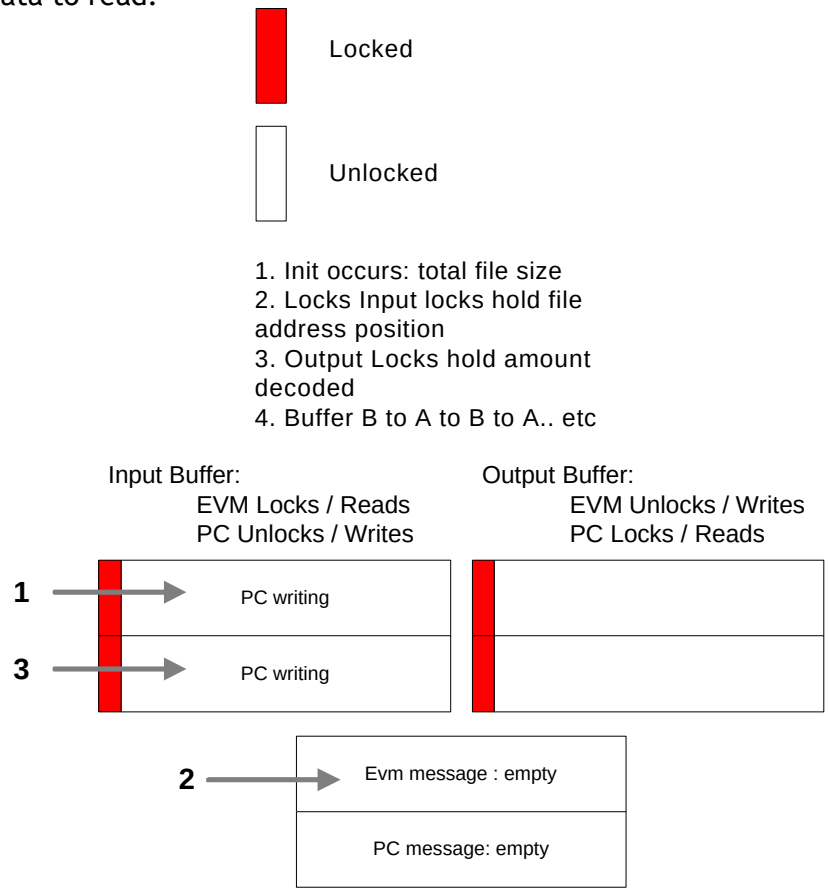
Step 2: Multitask!

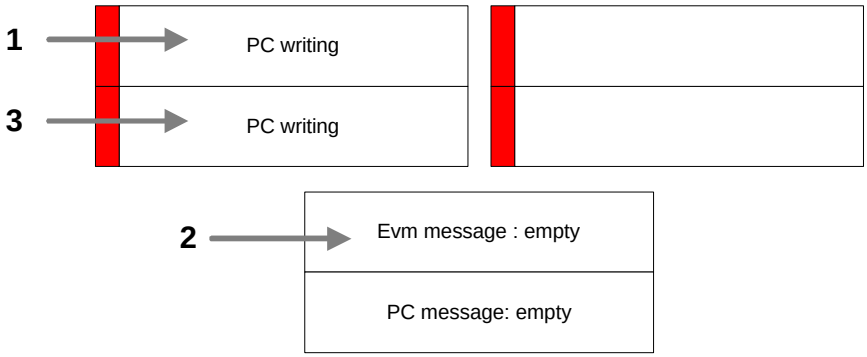
When the evm tells the pc to send data to buffer 2, it begins decoding the data in buffer 1. Initially this data consists of the headers of the ogg file. Once the evm

decodes this buffer it stores the RAW data into buffer 3. The evm produces and integer whenever it writes data to either buffer 3 or 4. This value represents the number of bytes written to a respective buffer. If this value equals the length of the output buffer then the evm will set a flag telling the pc to read from this output buffer. However this case is unlikely to happen. It most situations the evm must evaluate the situation. If the value of bytes is less then the buffer length or if the current number of bytes written to the output buffer is less than the buffer length, the evm will continue to write decoded data to the same output buffer. Once the output buffer is filled, the evm sets a flag telling the pc to read from that buffer while it sends new decoded data to the other output buffer.

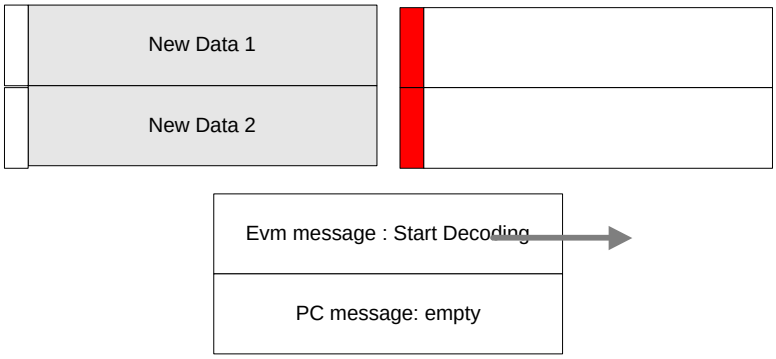
Step 3: Keep decoding until done!

So far the pc has written data to buffers 1 and 2. The evm has begun decoding data from buffer 1. The decoded information will be sent to buffer 3. Once the evm is finished, it tells the pc to read from buffer 3 as well as write more data to buffer 1. The data in buffer 1 is no longer needed so it can be overwritten. While the pc is reading decoded data from buffer 3 and writing data to buffer 1, the evm is decoding the data in buffer 2 and storing the decoded data into buffer 4. This process of reading, decoding and writing will continue until the evm runs out of data to decode. Once this happens the evm sets its final flag to the pc signaling that decoding has finished. The pc will continue to read decoded data from buffers 3 and 4 until there is no more data to read.

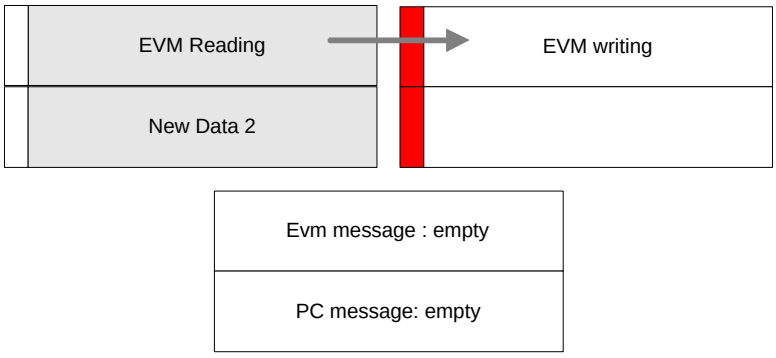




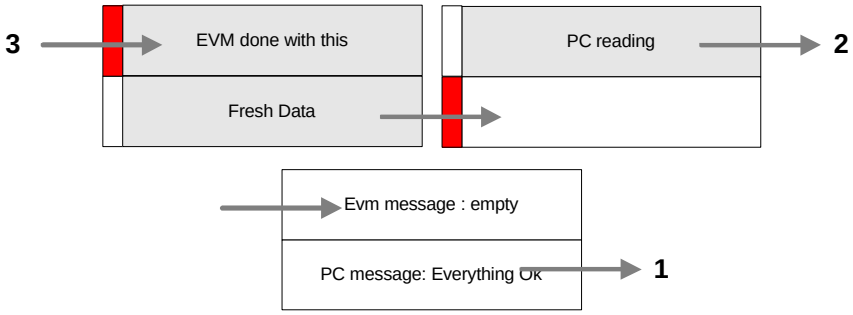
PC writes into input buffer A then unlocks it
Tells EVM to run with it
PC then writes into input buffer B and then unlocks



EVM starts decoding from input A when done locks input A,
writes to output buffer A then unlocks it; resets its box
EVM tells the PC it decoded a to a certain amount



EVM starts decoding from input B when done locks input B,
writes to output buffer B then unlocks it
while PC reads from buffer A or maybe Writes to input buffer A
which ever one comes first. Tells EVM to continue Etc...



Technical Difficulties

PC types different from EVM types / 40 bit longs
We had several problems with types on the board. Since we were using the same code on the EVM that we were using on the PC we simply added compiler preprocessors to achieve type compatibility.

```
//WIN
#ifdef _WIN32
 typedef __int64 ogg_int64_t;
 typedef __int32 ogg_int32_t;
 typedef unsigned __int32 ogg_uint32_t;
typedef __int16 ogg_int16_t;
#else
//EVM
typedef int ogg_int32_t;
typedef unsigned int ogg_uint32_t;
typedef short ogg_int16_t;
typedef double ogg_int64_t;
#endif
```

The PC will compile with the EVM's types but audio will be produced with noise. The code will not compile on the EVM using PC types. Since EVM uses 40bit longs instead of 64longs. This also lead to using rapper functions on the PC side to emulate the EVM's special handling for such things as 32 x 32 bit multiplies with 64 bit results as well as special function on the EVM side for handling 64 bit integers (these also ran on the PC, to evaluate correctness). Below are some examples.

```
#ifdef _WIN32  //these are base functions on the EVM
ogg_int64_t _mpyid(ogg_int32_t x, ogg_int32_t y) {
        union  {
          struct {
                  ogg_int32_t lo;
                  ogg_int32_t hi;
          } halves;
          ogg_int64_t whole;
          __int64 whole2;
        } magic;

         magic.whole2 = (__int64) x * y;
         return magic.whole; }
#endif

ogg_int64_t lshift64(ogg_int64_t num, char n) {
        union  {
          struct {
```

```
        ogg_int32_t lo;
        ogg_int32_t hi;
    } halves;
    ogg_int64_t whole;
} num2, ans;

num2.whole = num;
ans.halves.hi = ((unsigned int)num2.halves.hi << n) |
        ( (unsigned int)(num2.halves.lo & mask32[n]) >> (32 - n) );
ans.halves.lo = num2.halves.lo << n;
return ans.whole; }
```

Compiler Problems

We experienced several compiler problems.

1. When declaring more than 6 integers (or unknown strange combination of doubles, ints and/or longs) in a certain scope, excluding global, code can possibly become unstable. The exact reason for this is unknown. To solve this we made sure to verify code on the PC and to inline making sure not to consider the stated constraint. Example shown.

   ```
   void pizza(){
   int i1, i2, i3, i4, i5, i6;
   …
   { // need more ints, lets be safe
        int i7;
        …
   }
   ```

2. Called functions would under certain circumstances would not return to callie. Other groups experienced this problem. We looked at the asm files and were not able to notice anything strange, because the functions that were being called were small we simple inlined them into the callie function.

3. If our code was not completely explicit (see example below for the most annoying occurrence of this issue; the call back function based into qsort()) we would encounter strange bugs.

   ```
   static int sort32a(const void *a,const void *b){
            ogg_uint32_t a_temp = **(ogg_uint32_t **)a;
        ogg_uint32_t b_temp = **(ogg_uint32_t **)b;

        if(a_temp == b_temp)
            return 0;
        else if(a_temp < b_temp)
            return -1;
        else
            return 1;
        //The following two lines fail! Second is fairly standard.
        //return ( (**(ogg_uint32_t **)a>**(ogg_uint32_t **)b)<<1)-1;
   ```

```
        //return (ogg_int32_t)(a_temp > b_temp) - (ogg_int32_t)( a_temp <
b_temp );
}
```

4. We had many issues with Malloc. Even though sysmem was assigned into SDRAM the stdlib malloc functions would not allocate more than a total 64k bytes. We solved this by writing our own set of malloc functions (calloc, malloc, free, realloc) which used the stdlib malloc functions for small request which went on SBRAM and everything less, or when malloc was at its 64k limit, onto another portion of SBRAM then to SDRAM. We used a simple bucket system to keep track of allocated blocks. Below is our header file. We have explicitly defined memory regions to correspond with code composer's cmd file. We choose chunk sizes by taking statistics on the PC side. Because we used this method we were also able to 'simulate' the memory partitions on the EVM on the PC by mallocing equivalent memory sections on the PC using stdlib then using our own malloc to malloc within those sections.

```
#define HACK_MEM_START_SBRAM 0x00410004
#define HACK_MEM_SIZE_SBRAM  16380
#define HACK_MEM_CHUNK_SBRAM 4096
#define HACK_MEM_COUNT_SBRAM HACK_MEM_SIZE_SBRAM /
HACK_MEM_CHUNK_SBRAM
#define HACK_MEM_START_SDRAM0a 0x02000000
#define HACK_MEM_SIZE_SDRAM0a  0x200000
#define HACK_MEM_CHUNK_SDRAM0a 4096
#define HACK_MEM_COUNT_SDRAM0a HACK_MEM_SIZE_SDRAM0a /
HACK_MEM_CHUNK_SDRAM0a

#define HACK_MEM_START_SDRAM0b 0x02200000
#define HACK_MEM_SIZE_SDRAM0b  0x200000
#define HACK_MEM_CHUNK_SDRAM0b 16900
#define HACK_MEM_COUNT_SDRAM0b HACK_MEM_SIZE_SDRAM0b /
HACK_MEM_CHUNK_SDRAM0b

static struct
{
        char mem_set_sbram[(HACK_MEM_COUNT_SBRAM / 8) + 1];
        char mem_set_sdram0a[(HACK_MEM_COUNT_SDRAM0a / 8) + 1];
        char mem_set_sdram0b[(HACK_MEM_COUNT_SDRAM0b / 8) + 1];
#ifdef _WIN32
        int sdram0_a;
        int sdram0_b;
        int sdram1;
        int sbram;
#endif
} mem_hack ;

void hack_mem_init();
void * hack_malloc(int, int); // second parameter is request memory section
```

```
(eg: SDRAM)
void * hack_calloc(int, int, int);
void * hack_realloc(void *, int, int);
void hack_free(void *);
void hack_mem_close();

#endif

//our mallocs initization function
void hack_mem_init()
{
        /* init states */
        memset(&mem_hack.mem_set_sbram, 0,
sizeof(mem_hack.mem_set_sbram));
memset(&mem_hack.mem_set_sdram0a, 0,
sizeof(mem_hack.mem_set_sdram0a);
        memset(&mem_hack.mem_set_sdram0b, 0,
sizeof(mem_hack.mem_set_sdram0b));
#ifdef _WIN32
        mem_hack.sbram = malloc(HACK_MEM_SIZE_SBRAM );
        mem_hack.sdram0_b = malloc(HACK_MEM_SIZE_SDRAM0);
#endif
}
```

Performance

All timing was done using a 10 second ogg vorbis file. Because profiling requires compiling with the -g flag (symbolic debug), we used the timers.h library and wrote our own timing code on the PC side. This was acceptable because we were only looking to improve the decode time, therefore provided everything else remains constant we lose nothing by timing from the PC. Following is a chart showing the major speed improvements and some of the changes to the code that were responsible for these improvements.

~Time	Action	~Size
50sec	removed –g	440k
48sec	Removed printfs; moved to SBRAM	280k
30sec	Inline madness	245k
28sec	Deleted lines in code; c optimized	243k
26sec	Played with –o compiler flags	219k
24sec	Edited ASM files; changed .sect placed stuff onChip	219k
22sec	Double buffering on to board onChip(no more pci.h)	219k
20sec	Double buffering off from board	219k

The first action was to remove the –g compiler flag. Although this did not yield tremendous increases in speed, it shrunk the code from 440k to 280k. Removing the –g

flag in combination with removing all print statements shrunk the code to a size where we could put it on the SBRAM. Moving the code to SBRAM yielded a 37.5% increase in decode speed. At this point the code was much faster than it was to start with, yet the code was very messy as it contained function calls and variable declarations that we were no longer using. Essentially the front end of the code on the EVM side was rewritten. Although the code size reduction from this step was minimal, it brought our time down another 2 seconds. Next we played with the compiler flags for individual files, as for some reason not all of the files could be –o3 optimized. Once we found the fastest combination that still yielded correct results, out code was 2 seconds faster and 24k smaller. In the decode process there are some functions that get called very frequently. The next step was to put these functions onto the onchip memory. This reduced the running time by another two seconds. The final 4 second reduction in time came from removing the pci.h library altogether and using a messaging system we developed for all the data transfers.

Testing

To ensure the EVM code would work correctly, we developed code that would run identically on the PC and EVM. For instance, even though the PC can handle 64 bit logical shifts, we had the PC run the same shift code that the EVM was running. This allowed us to trace through the code from each and observe where differences among the output data began to occur. We would correct the function that was in error and then continue the process until eventually the evm produced valid pcm data.

The test sets of data that we used to test the code were ogg vorbis media files that we either ripped from .mp3 files using Sonic Foundry or downloaded from the internet. The primary file we used for testing was a 10 second clip of audio from the song “After Party” by Koffee Brown. The algorithm worked perfectly for this song, however when we tested the code on other songs there were some cases where the pcm data yielded by the evm had skips or clicks in it. We noticed this rather late into project (hours before the demo) thus we can only speculate on what was causing these errors. One idea is that when we turned off the “waiting flags” which make it so that the PC and EVM are perfectly in sync, the pcm data got written to the PC prematurely. The reason we did not notice this is because no errors appeared when we ran the short file through the evm. This was the fault of us spending too much thought on optimizing the code for speed and not considering what would happen in longer or otherwise different files

Demo

Our lab demo consisted of decoding Ogg Vorbis audio files of various sizes. The audio file that was used during all of the initial testing phases as well as for our timing results mentioned earlier was “k1.ogg”, a 162kB ten second audio file. This also happened to be the first audio file used in our demonstration. The demonstration of each audio file followed the same procedure. The evm was started first, followed by the pc. When the pc is started, the Ogg Vorbis file to decode as well as the RAW output file name is selected. Once decoding has finished, the RAW output file is loaded into a digital audio program named Goldwave for playback. Each RAW file must be played at 16 bit signed stereo and 44100Hz. This is the given specification by XIPH.ORG for RAW file decoded by the Tremor codec.

File 1:
The first Ogg Vorbis file decoded was a 162kB audio file. The evm as well as pc did not experience any problems while decoding this file. Playback of the RAW file showed audio quality was maintained during the decoding process.

File 2:
The next Ogg Vorbis file decoded was ~3.63MB. Although this was the first test using an Ogg file other than the 162kB k1.ogg, our group was confident that our algorithm would successfully decode the Ogg file. We assumed that decoding this file would longer to decode due to its size but not a significantly longer. After decoding for two minutes the evm had yet to signal it was finished. However the evm was decoding data as shown from the increasing size of the RAW file. At this time we concluded that either the pc or evm were stuck waiting for a flag to be set. The RAW file produced was ~4MB and was playable using Goldwave. Audio quality was maintained for this Ogg Vorbis file, however an occasional clip in audio was noticed near the end of the file.

File 3:
The final Ogg Vorbis file decoded was ~4.24MB. At this point our group expected this file to have the same halting experienced with the previous file. Once decoding started we gave the evm two minutes to decode the Ogg file. After timed had expired we manually killed the evm and pc. The RAW file produced this time was again ~4MB. It was playable using Goldwave and maintained audio quality, however with subtle clips that were not as noticeable as in the previous file.

Results:
We concluded that the decoding algorithm used on the evm was successful for our purposes. With more time and resources we believe Ogg Vorbis files can be decoded in a small amount of time, if not real time. The clips heard in the last two ogg files were unexpected by us but were reasoned to occur when the evm is writing RAW data to the output buffers and checks if the buffer is full or not.

For future advancements on our project would start with memory allocation. Malloc is called at least 100 times, which was an initial concern for us. Currently our “ghetto malloc” uses onchip memory and parts of SDRAM0. An efficient version of malloc that can handle block sizes of various lengths and can adjust to them would be necessary to minimize the amount of memory needed for malloc calls.

Each input and output buffer we use are 4096 bytes each. It made sense for us to use buffers of this size because we could fit each on internal memory and by storing the entire Ogg Vorbis file and/or the RAW output file onto memory would use external memory and would result in slower times. By increasing the size of internal memory, our buffers sizes would also be able to increase, reducing the number of reads/writes between the evm and pc.

Purchases
none.

References

[1] http://www.mathdogs.com/vorbis-illuminated/

[2] http://www.xiph.org/ - Ogg Vorbis Tremor source code as well as documentation

[3] The Use of Multirate Filter Banks for Coding of High Quality Digital Audio, Sporer, Brandenburg, Elder – Documentation of the MDCT algorithm, found at www.xiph.org

[4] Ogg Vorbis software developers (through IRC chat room irc.xiph.org #vorbis) – Answered questions to Tremor code specifics

[5] Not-So Live MP3 Encoder, 18-551 Project Spring 2000, Group 14 – Information concerning the MDCT as well as tips for evm implementations

Software Used

1. Mathematica
2. Matlab
3. Excel
4. Visual Studio C/C++
5. Code Composer C
6. Goldwave