Not-So Live MP3 Encoder

18-551

Project Final Paper

GROUP 14:

Thomas McElroy

Charlie Oswald

Domenic Senger-Schenck

5/8/2000

Introduction:

Today, portable MP3 digital audio decoders/players are everywhere. These devices are easy to implement because decoding takes relatively little computation. Encoding, however, is a more computationally expensive operation and, as a result, has not yet been implemented on any portable devices. As of now, the power of a computer is needed to do the encoding and is usually a fairly slow process. There are, however, some algorithms that can encode CD's to MP3 format in real time using very fast, high-end machines (http://www.qdesign.com/MP3sdk.htm). But, a standalone, portable unit to do MP3 encoding in real time does not yet exist.

There is certainly a market for such a device. Sony MiniDisc recorders use a compression scheme called ATRAC (Adaptive Transform Acoustic Coding) so that they can fit the same amount of music as a compact disc onto a much smaller disc. If MP3 encoding could be implemented in the same context, one could improve upon both compression rate and sound quality. But MP3 encoding would be most useful in an application such as web broadcasting. Because of its low bit rate, MP3 encoded music can be transmitted using far less bandwidth than standard PCM encoded audio, or ATRAC encoded music for that matter. A real time encoder

would allow live broadcasts from anywhere without the need for a computer. A DSP could be packaged in a box with an analog input from the concert audio and an Ethernet interface for broadcasting over the internet. Thus, MP3 has uses in both recorded and broadcast music.

As stated above, encoding is relatively complicated compared to decoding. Technology now exists to decode fast enough on a portable device for real time playback of musical tracks. It is only a matter of time before the development of a real time encoder. It was our goal to implement such an encoder on the C67.

Background:

The main point of MP3 encoding is to analyze an audio signal to find out which elements of the signal are unnecessary because they are inaudible to the human ear. Masking effects are accounted for in both the time and frequency domain, though there is usually more to be gained in the frequency domain. In order to perform this analysis and filtering, the MP3 encoding process makes use of a polyphase filterbank, a modified discrete cosine transform (MDCT), and a psycho-acoustic model which uses an FFT in part of its analysis. The results from these steps are

then further compressed by using standard Huffman encoding. The signals-based steps are discussed below, leaving out the details of Huffman coding.

Goal:

The goal of our project was to implement an MPEG-1 Layer-3 encoder on the Texas Instruments C67. We believed that many of the stages of the algorithm could be executed faster, more efficiently, and, most importantly, eventually in a stand-alone fashion, on the C67. By developing code for the PC based on this ISO standard algorithm, these stages could be executed either on the PC or C30 or a combination of the both.

We planned to have dual functionality in our MP3 encoder/recorder. The first step of the project was to implement an MP3 encoder to convert from a digital Wave file (PCM encoded) on the PC with 44.1 kHz sampling rate. The goal here was simply to process the encoding as fast as possible so that the speed needed for real time encoding was available. Our next plan was to then make our DSP code able to convert an analog signal directly from the A/D RCA jack on the EVM board (sampled at 44.1 kHz). Again, this

was to be done in real time, the only useful speed at which to encode a live analog input.

Psycho-acoustic Analysis:

The first step in the encoding process is the psycho-acoustic analysis. It is done to weight the sub-band information based on how a human will hear it. For example, an intense tone at 1000 Hz will mask a 1100 Hz tone that is soft. So, because the ear cannot hear much detail on the 1100 Hz tone, very little detail needs be stored as to its exact shape and magnitude. In MP3 encoding, the psycho-acoustic model takes a 1024 point FFT of the input signal (audio signal to be encoded) which gives it relatively high frequency resolution. The signal is thereby effectively split into 63 frequency bands. These bands are compared against one another to test for masking. Furthermore, the psycho-acoustic model will use the FFT results to split the data into long and short blocks, depending on whether high frequency or high time resolution is needed, respectively. Using pre-calculated tables that tell the psycho-acoustic model which frequency bands are masked and by how much so that it can decide how much quantization error is allowed. Actual testing of human hearing, using controlled experiments, was used to

generate these tables. As it works out, a loss of 1 bit in quantization results in 6dB of error. This means that if a signal is below the masking threshold it can be ignored, but if it is above the masking threshold, it can afford some quantization error so long as that error does not rise above the masking threshold. In this way you can disregard certain frequency bands and inject imperceptible error in others, reducing the resultant bit rate.

The psycho-acoustic model also takes into account the effects of time domain masking. To do this it needs to perform some prediction since it can account for masking both before and after a strong signal. A strong sound will mask other sounds coming immediately after and right before the dominant sound.

Polyphase Filtering:

The polyphase filtering step of the encoding process is called after the psychoacoustic analysis step. As implemented in the code we are using, the polyphase filter takes one channel from one granule at a time. Filtering is done by nested loops multiplying and adding by a constant matrix that represents the necessary filter. This produces

32 separated sub-bands that can now be operated on to further encode before using perceptual information.

Modified Discrete Cosine Transform:

The MDCT, or modified discrete cosine transform, is similar to a fast fourier transform, but the way it is used on the outputs of the filterbank is to finalize pure, non-perceptual lossy encoding by breaking up each sub-band into primary components and send them off to be compressed. The benefit of using an MDCT instead of an FFT is that temporal aliasing is eliminated. The MDCT uses 50% overlapping windows to compute coefficients for each given sub-band.

Process:

Because the encoding process is so complex, we decided to start with existing MP3 code. It was first necessary to remove the user interface from the code, since this would not be used. But the program was still too large for the available memory on the EVM. So we next needed to trim out all unnecessary elements of the code. Because we were only doing stereo, 44.1kHz sampled, encoding to 256 bits/second, we could eliminate much of the code allowing switching between these and other available options. Finally, as we were to find out very late in the semester, it was also

necessary to get rid of as many doubles from the code as possible.

Implementation of Code Modifications:

We started with the source code to a command line multi-platform MP3 encoder and wanted to implement an MP3 encoder that was fast and lean and able to run on the EVM. To move from start to finish took a lot of work.

First of all, we needed to gain a good understanding of what procedures and function calls implemented what high-level filters and operations and figure out how much of the code we really needed. This took quite some time, not only due to the complexity of the algorithm and the enormous volume of code that allowed the source code great cross-platform ability, but also due to the fact that it had been programmed and optimized for very general use on a processor with a large store of RAM.

For example, the interface to the encoding function was very obscured by massive volumes of user interface code, but the biggest obstacle by far was the unrestrained use of global variables. The data flow through the program was made opaque by having data flow through global

variables because we could not trace them through function calls. Contributing to the difficulty of figuring out program flow was the fact that there were almost no comments and the variable names were incredibly arcane, and not grouped by functionality. For instance, there were a few buffers that represented the coefficients for filters that were initialized towards the beginning of the code, and then used later, but they weren't named in such a way that belied their purpose, and there were often local variables that were either named the same thing, or local pointer variables that were set to point to the global variables so that the global data could be accessed more simply.

Once we had the code stripped down to the functions we wanted, we had to begin the process of getting it to work, and work quickly on the EVM. To start off this task we worked on getting it to load into the EVM memory which took a little bit of rudimentary learning on how to set functions to go where and what else needed to occupy memory (.cinit, .stack, .sysmem, using pragma, etc.). Once this was complete, we worked on the necessary communication channel between the PC and the EVM. We spent some time testing it and found that there were some definite problems

going on in just the communications channel. This was a bit unexpected because we were using working code from lab3 to do the communications.

Once we had the communications working (or working better at least) we started at the task of testing the MP3 encoder on the EVM. It didn't work. We were running the same code on the EVM and the PC and it just would not work in ways we couldn't understand. We came to understand from Steve that there might be something "flaky" about the doubles on the EVM, so we asked Pete what the situation was. He concurred that there was definitely something wrong with LDDW when accessing off chip memory, but he wasn't sure. This worried us severely as it was close to the deadline. We got the full story from TI the next day.

At this point our push became to try and get it to work despite this problem. We first tried naively changing all doubles to floats and changing all calls to math functions to be mathf. The conversion of doubles to floats worked for some of the variables, but, for others, it appeared to ruin the functionality of the program. So, we went back to the doubles, but kept the mathf functions. We then tried to figure out if we could get rid of some of the

doubles while keeping the others there, and we had limited success with that, but there were still quite a few global variables that were doubles. At this point we changed tacks and just tried to get all the global doubles to fit in on chip memory so that LDDW would work. We did manage to do this, using both on chip data and program memory spaces. Still, the program would not work on the EVM.

We had moved all the doubles on chip, including all static variables in functions that, as static doubles, would get allocated off chip. However, this still yielded an error despite the fact that the exact same code worked on the PC. We could only guess that this was due to the volume of LDDWs happening from on chip memory that have slim chances at error.

So, after the due date and demo had passed, we worked on trying to remove all global doubles. When we really delved into this task we realized that the reason the code had failed when we did a naïve replace of doubles with floats was due to the amount of global pointer passing. As mentioned previously, there are local pointers that are initialized at the beginning of functions to point to global variables, and then accessed to change the data

pointed to by the global pointer. During this process, the doubles we had set to floats were being coerced back to doubles because a double pointer was pointing at the memory location. This, of course, was resulting in all types of errors.

We spent some time sifting through all the functions that these global variables touched, having to change all the locations at once to keep the data consistent for testing on the PC. In this way of finding all references to a global variable through using some of the advanced browse features in MS Development Studio, and persistence, we were able to eliminate all the global Double variables, leaving only the doubles local to functions. However, despite the fact that it compiled and ran correctly on the PC (with a little distortion due to the lack of precision), it would no longer run to completion on the EVM. The problem would appear at first glance to be a memory problem causing a stack overflow, however, we have set our stack size to be very large, our ".text" section is split up between ONCHIP_PROG and SBSRAM_PROG, and yet, when a couple of our functions return, they jump to garbage addresses. We also checked the stack pointer, register B15, to make sure it was pointing to a valid address. The

pointer was fine, but apparently the return function address it was pointing to had been corrupted.

At this point we gave up yet again. We do not know if it is the slim chance of the onchip LDDWs failing something in TI's memory management or something we are not setting up for correct stack management, but it seems there is little more we can do but comment the code we have been working on so that someone else can come in and modify it with much more ease.

It is at this point where we now find ourselves. We do not know if there are still problems using LDDW, even though all doubles are on chip. The TI errata sheet (published on January 18, 2000 and posted to the class web page May 3, 2000) states that LDDW only fetches the lower 32 bits of a 64 bit word, even on chip in some cases. Please see page 7 of the C6700 Silicon Errata sheet referenced from the lab files section of the course web page. When using doubles, even on chip, an error will be caused when "stepping through" the program using the debugger any time an LDDW occurs, making the debugger useless when any double is loaded. This problem will be fixed with the new Version 1 silicon boards and it is

suggested that these boards be used in the years to come, or to switch to a different EVM.

Conclusion:

Our attempt at the end of the road was to set up things to be carried by another group if this project is ever attempted again. We tried not to reiterate details about the encoding process from last year's paper, as that can be used in conjunction with our paper and code as a starting point for the next group. We believe, that as engineers, commenting our code and describing the problems encountered was the best way to spend our time, so the next group will not have to struggle through the same issues.

One suggestion that all of our group members strongly advise following is that this project should not be attempted again until new version EVM's are installed in the lab. Although it may be possible to implement on the current EVM, the semester will probably just be spent chasing doubles instead of implementing signals algorithms.

References:

http://www.ece.cmu.edu/~ee551/Old projects/s99 projects.html

Previous 551 project on MP3 encoding.

http://www.qdesign.com/MP3sdk.htm

Web site demonstrating current capabilities of software encoders to encode in real time.

http://bladeenc.mp3.no/

Web site containing the source code for our encoder.