Audio Compression
14-bit to 8-bit

Duncan MacDonald
duncmac.16@gmail.com
V00842740

SENG 440 - Embedded Systems
University Of Victoria
Department of Engineering

July 24, 2019

Contents

List of Figures

1 Introduction

Recording audio and storing it digitally, in its fullest form is a very costly process; the same goes for transmitting it. For that reason, audio compression came to be. There are two type of audio compression methods, lossless and lossy, this report will be going over the latter, specifically μ-law and A-law. Even though μ-law does loss some data in compression, it's usage is still justified in telecommunication as it makes data transfer a lot faster with little noticeable to the human ear[1].

μ-law takes in a 14-bit sample (14 of which are used) of the recorded signal and creates a logarithmic approximation of the signal recording the position of the most significant bits and the step of the preceding bits. This project in particular is an implementation of μ-law used on to compress a audio file and uses the inverse to expand it back to a usable file. After the initial implementation was create, multiple different optimization techniques were preformed to the algorithm to take the highest traffic parts and reduce the instructions needed to complete them. After the software optimizations were completed, hypothetical hardware assisted were added to the algorithm to increase the speed of the algorithm even further.

The resulting algorithm would be the base to a highly optimized embedded system could be further implemented with hardware to be used in a production setting. The following report highlights the background knowledge the algorithm is based on, the optimization techniques used in the software and performance analysis on all stages of the design.

2 Background

The basis of why the data lost in compression is justifiable is the characteristics of human speech[1]. Most of the things people say vocally are of a much higher amplitude than things such as background noise or sounds of friction of things around the recording device[1]. This is why compressing large signals of vocal sounds is affordable, since the lower amplitude signals of non-vocal sounds aren't really relevant to the transfer of human voice.

The way that wave files are recorded is in uniformly spaced quantized values of a signal, with lower levels of noise recorded with fewer bits and larger levels of noise recorded with more bits. This is referred to as uniform pulse code modulation[1]. The idea behind such codec as μ-law and A-law is to take a logarithmic

representation of the uniformly spaced signals so that an 8-bit code can represent the potentially larger signal.

$$y = sgn(x)\frac{ln(1 + \mu|x|)}{ln(1 + \mu)} \tag{1}$$

$$y = \begin{cases} sgn(x)\frac{A|x|}{1+lnA} \\ sgn(x)\frac{1+ln(A|x|)}{1+lnA} \end{cases} \tag{2}$$

μ-law, represented by equation 1 is the function that is to implemented in software, where μ is a parameter that can range from 0 (no compression) to 255 (maximum compression)[1]. To implement this in software, I am going to use a piecewise logarithmic approximation of this equation to compress a signal. Inversely, I am going to use exponentials to expand the compressed codec into something useful again.

3 Design

3.1 Modelling

To create a basis of how the program was going to work, the first needed to be done was to model the program with a UML diagram. Using this would create a generic overview of how the program was going to flow and what components need to created in high-level terms. There was some deviation from this diagram in the first iteration of creating the code for the sake of readability, but the algorithm remained somewhat the same going to UML to C.
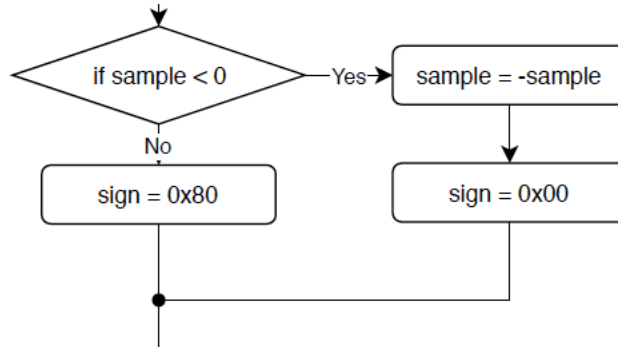

Figure 1: UML diagram of procedure that takes place when finding magnitude.

The first thing that is done after the initialization of the variables, is the magnitude of the sample is found. When this happens, the flip of the last bit is store and the magnitude is kept for the rest of the processing.
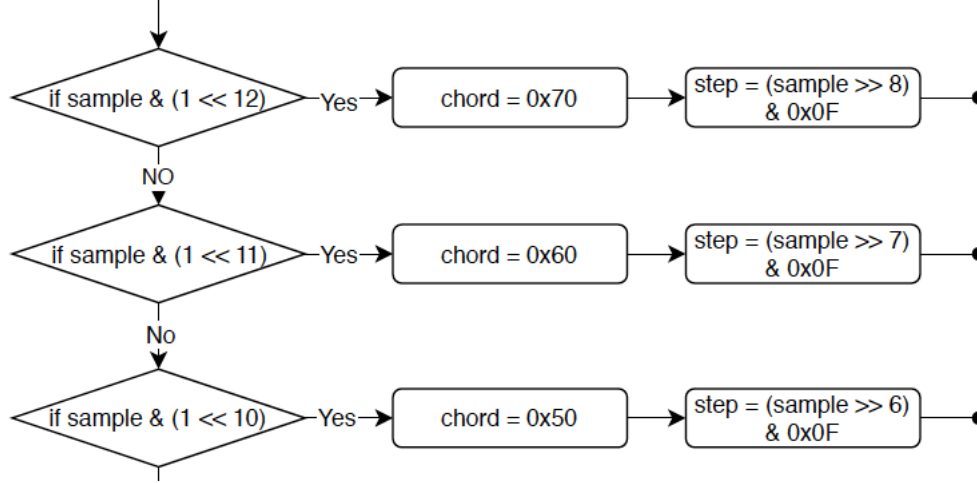

Figure 2: UML diagram of procedure that takes place when finding the piecewise log of the sample.

After the magnitude is found, the piecewise log is found and stored and then the 4 bits after wards are stored as the step of the value. Out of the scope of the figure is all of these components together in a loop, storing the results in an area. This deviates from what was actually code for the arm controller, as there was only a test suite set up for a select few values.

3.2 Design Methodology

For a design methodology, I used a low cost version of test driven development. To produce some samples I made a 16-bit integer as a pseudo-sample and compressed it into a code word and expanded it again. In the end of the first iteration, some of the samples produced by the testing did not expand back in the same format, but it was expected since the samples produced were not in the format of a 14-bit quantized value. The rest of the values expanded in a similar fashion, only keeping the five or six most significant figures and dropping the rest. The samples produced were dispersed throughout a range of different chords and steps so that all aspects were tested. I used a for loop as seen in figure 1 that takes the initial sample and shifts it over one space, clearing the most significant bit and compresses and decompresses sample and displays the results. This was then repeat with a different step to ensure that the steps were correctly being

4

compressed. As I mentions before this was a low cost testing method, so no actual testing software was used. Using a form of test driven development was very helpful though, as bugs that occurred were quickly seen and corrected.

```
int i;
for (i = 0; i < 8; i ++) {
    sample = sample >> 1;
    sample &= ~(1 << (16 - i));
    sample1 = sample >> 8;
    sample2 = sample & 0xFF;
    code = compress_sample(sample1, sample2);
    expanded = expand_word(code);
    printf("%d Shift\nSample: %x\nCodeword: %x \nExpanded: %x\n\n", i + 1, (unsigned short int)s
}
```

Figure 3: Testing for loop to produce samples to compress

3.3 Original Design

In the actual implementation of the algorithm, the program would take in a file and write the data into a struct with a place for a header, data and the size of the data. The first thing that was done was removing the head from the original file and placing it in the compressed file. After this was done the data, in the form of 8 bit character data type were compressed, this was done by first getting the sign of the character, the magnitude of the character, the chord and step. After the values were found, they were connected together into one code word and written into the `Compressed` struct. The header and the compressed data was then written into a file.

The sign, step and chord were found through functions calls in the first iteration for the sake of convenience and readability. This was done knowing that it would later have to be optimized, but I thought it would be better to get the algorithm complete and readable before I started optimizing the code. There were multiple different things I did to get more out of readability and harm on speed performance. An example of this can be seen in figure 2, where I used a for loop to cycle through cases to find the leading bit.

3.4 Optimizations

To optimized the code, the parts of the code looked at were the parts that were called the most. All of the parts of code having to do with the compression and

```
char get_chord(short int mag) {
  char chord = 0;
  int i;
  for (i = 12; i > 4; i--) {
    if (mag & (1 << i)) {
      chord = i - 5;
      return chord;
    }
  }
  return chord;
}
```

Figure 4: For loop used to find the leading one bit.

expansion could be improved by quite a bit. The first thing that was done to optimize the code is to remove the for loop shown in figure 2. Using a for loop creates a significant amount of branching added instructions to increment the loop. There is also a lot of extra math that could be avoided. So the result of optimization produced a long string of if else statements.

To optimize the compression further, all of the functions in the code to find the different sections of the code words were put into the same function. Doing this takes away the overhead of stack pushing and popping. After all of the function were brought together, the values that were previously returned using math were hard coded save on operations. The bit values were written in the way that they would be in the code word (ex. finding a chord of `0x4` would result in `0x40`). This helps save operations on shifting bits over when combining the values of step, chord and sign together.

Another aspect of the code that was changed to reduce time is using the `register` key for the `sample` variable. This being a highly modified value, it is important to remove the overhead of having to look in memory for this variable. The use of of `static inline` for the function makes the code in the function be placed in where the function call is, reducing the amount of stack used when running.

3.5 Hardware Design