Audio Compression

14-bit to 8-bit

Duncan MacDonald
duncmac.16@gmail.com
V00842740

SENG 440 - Embedded Systems
University Of Victoria
Department of Engineering

August 1, 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 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 to compress an 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 assists 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 `.wav` 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 codecs 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 be 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, the software is going to use a piecewise logarithmic approximation of this equation to compress a signal. Inversely, the software is 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 step needed to be done was to model the program with a UML diagram to meet the requirements of the system. Doing 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 from UML to C.

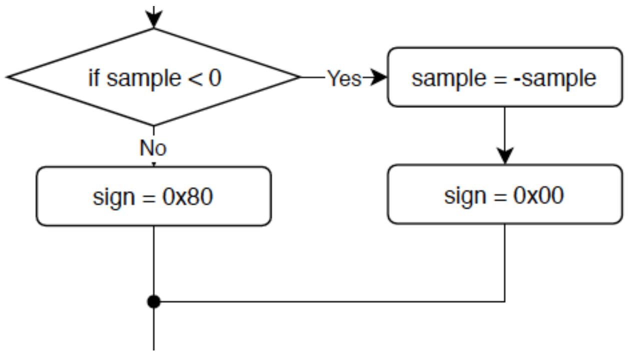


Figure 1: UML diagram of procedure that takes place when finding magnitude.

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

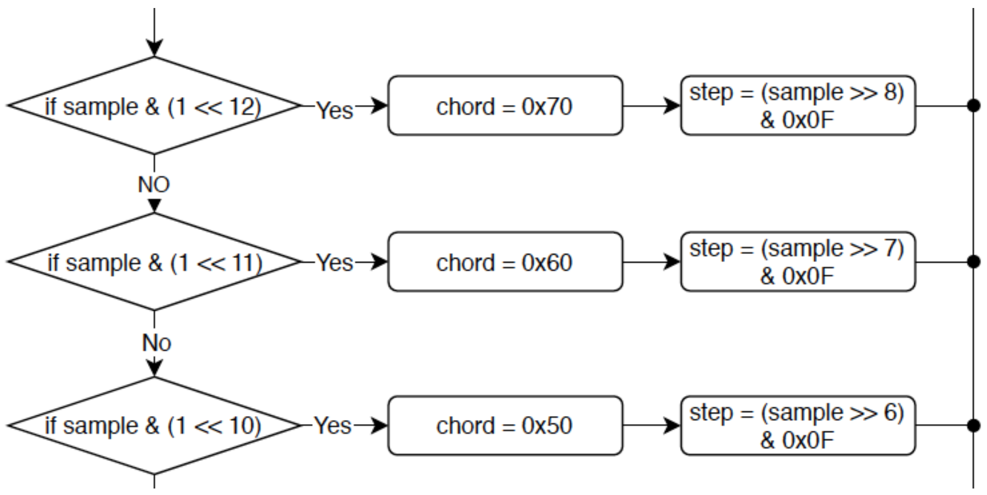
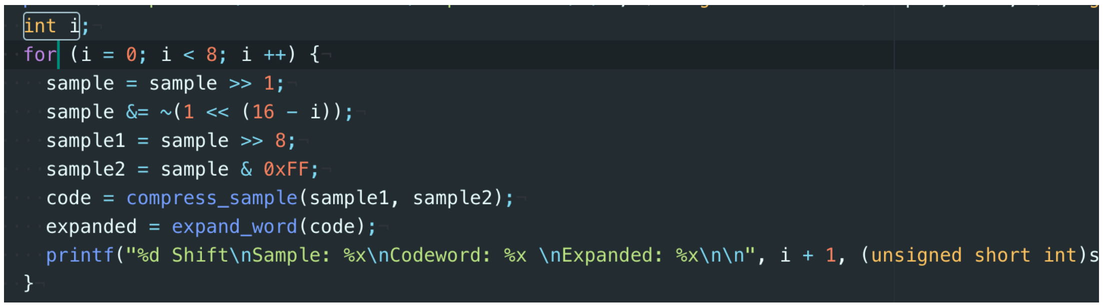


Figure 2: UML diagram of procedure that takes place when finding the piecewise log of the sample.

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 of the most significant figures and dropping the rest. The pseudo-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 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

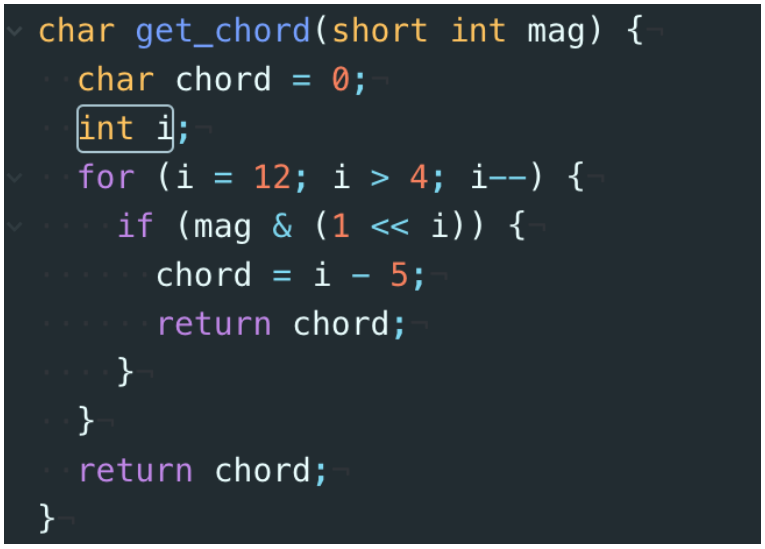
3.3 Original Design

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 that harmed the speed performance. An example of this can be seen in figure 4, 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



```c
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 4. Using a for loop creates a significant amount of branching and 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.

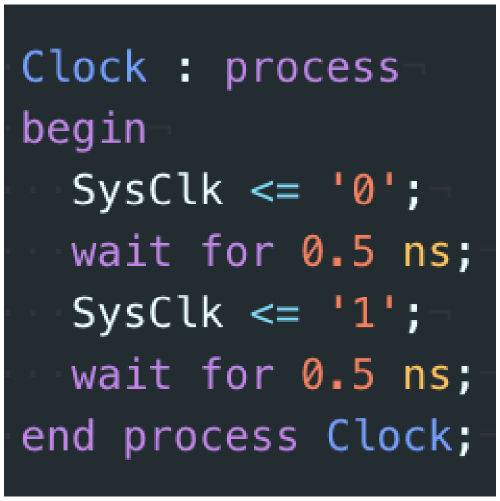
Another aspect of the code that was changed to reduce time is using the **`register`** key for the **`sample`** variable. This being a highly modified value, it is important to remove the overhead of having to look in memory for this variable.

3.5 Hardware Design

To design hardware for the system, the bottle neck of the system had to be identified. This was definitely the actual compression or decompression part of the algorithm as it would be called for each sample in the file or the stream going into or out of the system. In the program that compresses a whole `.wav` file. The algorithm first removes the header of the file, because that information in that cannot be lossy, then it goes through all of the samples in the file and compresses them into code words. Because the head is under 100 bytes of information and the body contains a significant more than that, the bottle neck of the system would benefit greatly if there the samples were compressed through hardware.

VHDL

To model the compression of the algorithm, the algorithm was redesigned in VHDL, a hardware description language. This was useful to use as it simulated the logic of designing a circuit without having to actually design it. In the hardware model, there would be an input for the mode of either compression or decompression, and an input of either the codeword or sample sample. The output would be either the compressed sample or decompressed codeword respective to the mode that the hardware was in.

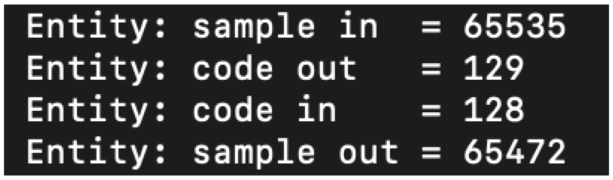


```
Clock : process
begin
  SysClk <= '0';
  wait for 0.5 ns;
  SysClk <= '1';
  wait for 0.5 ns;
end process Clock;
```

Figure 5: Clock component used in VHDL simulation.

To implement this in VHDL, there needed to be an architecture to have a test bench and a flow throughout the system. The main component of the system is the `Clock` process as seen in figure 5. This component is responsible for creating all of the values and generating the signals for the compression and decompression methods. The `SysClk` signal is a value of one or zero that is updated every half nanosecond. On every rising edge of the `SysClk` the `input1` signal is incremented by one and sent in the ulaw. The `input1` is used by both the compression and decompression processes for the sake of being able to see the samples going into the system and the samples going out of the system at the same time. The other part of the simulation is the `simulation` process; since this process is running concurrently with the rest of the program, it will be able to stop the simulation once it has reach a time when all of the values have entered into the system.

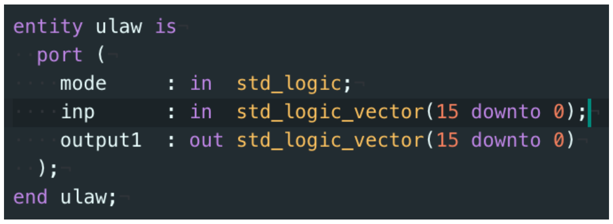
Together, the clock signal drives the incrementation of the inputs going into the compression and decompression until the simulation process finishes. Since these processes run concurrently, both the code word going out of the compression and the code word going into the decompression process were recorded because the decompression method would happen faster than the compression method and not update the signal until after the compression process was done. This phenomenon can be seen in figure 6.



```
Entity: sample in  = 65535
Entity: code out   = 129
Entity: code in    = 128
Entity: sample out = 65472
```

Figure 6: Concurrency in action in during simulation.

The mode component was added to the entity, but not actually implemented, since it would be trivial to add and made it a lot harder to test. This was also true for the output in the `ulaw` entity. The resulting assembly operation would be `ulaw Rs Rm Rt` as suggested by Dr.Sima's slides, where Rs is the source operand, Rm would be mode and Rt is the target operand where the results would be stored[1]. Since ARM is a 32-bit processor, all inputs and outputs would be 32-bit words, but not all of the bits would be used. A VHDL representation of the operator can be seen in figure 7.



```
entity ulaw is
  port (
    mode      : in  std_logic;
    inp       : in  std_logic_vector(15 downto 0);
    output1   : out std_logic_vector(15 downto 0)
  );
end ulaw;
```

Figure 7: Concurrency in action in during simulation.

4 Performance

To determine the performance of the system, I decided to time the functions in action. In this evaluation, my main machine was used for to give an estimation of how the performance would increase in an embedded environment. To reduce the affect of overhead on the timing of the function, it was ran two-hundred-million times to get the average time it takes to run the function. Figure 8 shows the amount of cycles it took to complete each task.

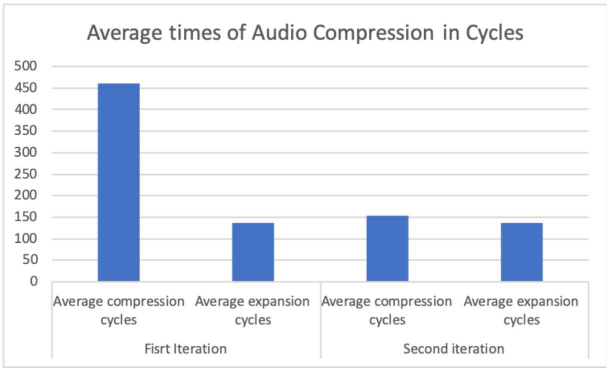


Figure 8: Cycles it takes for each process on average.

The amount of cycles could be directly related to the amount of instructions were produced from the C code along with the little amount of branching. Figure ?? shows the amount of instructions produced in the functions `compress_sample` with and without optimization.

5 Conclusion

Sending audio signals in there fullest, uniformly pulsated state is very costly in the amount of data needed to be sent. The characteristics of the human voice and auditory system as such that people do not need to hear lower amplitude sound.

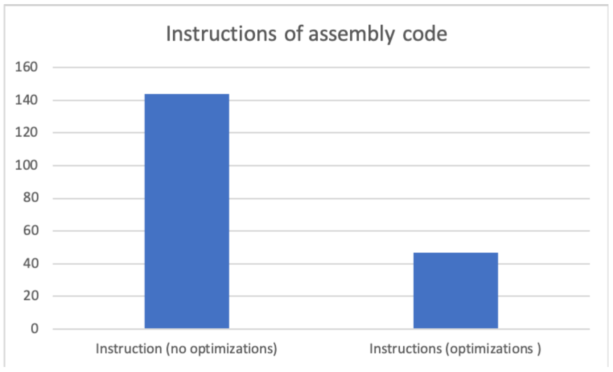


Figure 9: Comparing the amount of instructions in an optimized and non-optimized process.

This gives reason to use such codecs as A-law and μ-law to compress these signals into something that is both smaller than the original signal and recognizable to the human ear even if there is some data lost in compression.

The piecewise logarithm method, implementable in software, is a good way to approximate μ-law to preform a lossy compression of audio samples. The algorithm was first implemented as a UML diagram to complete the requirements of the system and then converted into C code. After the C code was optimized to preform the same task in fewer instructions and less looking in memory. The assembly code was then generate through the `gcc` and more unnecessary instructions were taken out. After this was done, a simulation of the hardware that would replace this compression method was designed and tested. Finally, the hypothetical instruction described by the simulation replaced the μ-law compression and expansion methods and the instruction count with the estimated latency of new instruction was used to analyse the performance of system with no optimizations, the system with optimizations, and the system with simulated hardware involved.

Audio compression, software optimization and hardware design are what make such systems as telecommunications possible. These techniques and practices are permeated throughout the design world and are important to having high performing systems.

6 Bibliography

[1] M.Sima, “SENG 440 Lesson 101 Project Audio Compression” [Online Document]
http:www.ece.uvic.ca msimaTEACHINGCOURSESSENG_440 PROTECTEDSLIDES
SENG_440_slides_Lesson_project_101.pdf
[Last accessed July 31 2019]