University of Victoria
Department of Software Engineering

SENG 440 Project
14-bit to 8-bit Audio Compression

By:

Hamzah Mansour	V00812108	(hamzahbmansour@gmail.com)
Rabjot Aujla	V00724640	(aujla.rj@gmail.com)

Submitted: August 2, 2018

Table of Contents

1 List of Figures and Tables

2 Introduction

Audio compression is very commonly used to reduce storage and easy transmission of audio data. This is generally achieved by using *A law* or μ *law* method. Both method used a logarithmic function to scale and map the given input to desired output. This is based on the idea that if we increase the quantization step for large signal levels, while we keep it as it is for small signals levels, the maximum relative quantization error remains the same[1]. This allows us to use fewer bits to represent the audio signal.

The project explains the basis of audio compression using μ *law* by demonstrating an example which compresses a 14-bit value to 8-bit value giving us the compression ratio of 1.75:1. The implementation of the piecewise algorithmic function is performed in C. The algorithm is created by dividing the possible input values into 8 segments and calculating the output based on those segments. The lower bits goes through a much lower compression since that data is more sensitive to human ear. The signals with higher dynamic ranges, which are less sensitive to human ear, goes through a higher compression ratio.

This report illustrate the scaling of algorithm to our desired input & output, detailed implementation of code, optimization methods performed for efficiency and results of our experiment using the given methods.

3 Theoretical Background

This section of the report details the theoretical calculation and equations used to develop our implementation in C.

3.1 Initial Calculations:

North american μ *law* quantizer equation:

$$y = \frac{ln(1+\mu x)}{ln(1+\mu)} \qquad (1)$$

where $0 \leq x \leq 1$ and $0 \leq \mu \leq 255$. In this project we using $\mu = 255$ which gives us the following equation:

$$y = \frac{1}{8}\ log_2(1 + 225x) \qquad (2)$$

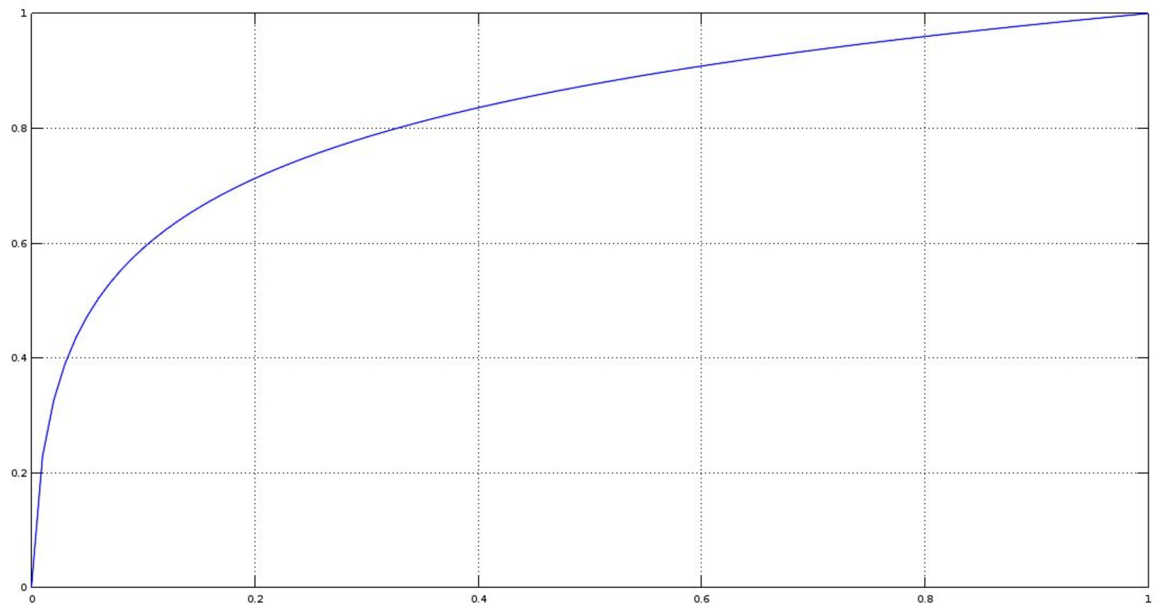

Figure 1: log_2 graph with normalised input/output

To scale the function to take 0 to 2^{14} input(represented as X) following steps were performed:

$$y = \frac{1}{8}\ log_2((1+225x)\frac{2^{14}}{2^{14}}) => \frac{1}{8}log_2(2^{14}+(256-1)X) - \frac{1}{8}log_2(2^{14})$$

$$y = \frac{1}{8}\ log_2(2^{14}+256X-X) - \frac{7}{4} \qquad (3)$$

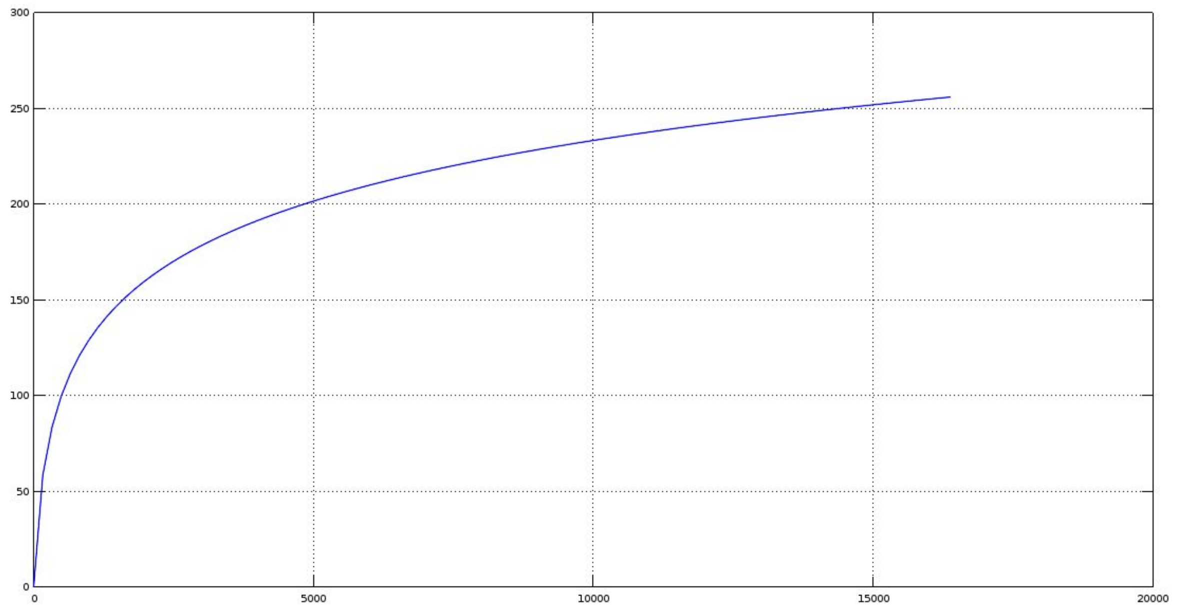

Figure 2: log_2 graph with integer input/output

3.2 Segment Calculation

This section explain how the segments on x axis and y axis were calculated for 14-bit(2^{14}) and 8-bit(2^8) respectively.

3.2.1 X-axis

Since our input is 14-bit, our x axis values ranges from 0 to 16834. As explained in the above section, we are using $log_2(1+255x)$ arithmetic function to compute the output. Our goal is to obtain appropriate x-values in powers of 2 to get our 8 segments. The table below shows our results:

Table 1: Segment Calculation for x axis

Segment #	$log_2(1+255x)$	Normalized x	Scaled x for 14-bit
8	$log_2(256)$	1	16384
7	$log_2(128)$	127/255	8159
6	$log_2(64)$	63/255	4047
5	$log_2(32)$	31/255	1999
4	$log_2(16)$	15/255	963
3	$log_2(8)$	7/255	451
2	$log_2(4)$	3/255	193
1	$log_2(2)$	1/255	64

3.2.2 Y-axis

Computing the y-axis was much similar as we only have to divided it into 8 equivalent segment. The table belows our results:

Table 2: Segment calculation for y-axis

Segment #	Normalised y	Scaled y for 8-bit
8	1	256
7	7/8	224
6	6/8	192
5	5/8	160
4	4/8	128
3	3/8	96
2	2/8	64
1	1/8	32

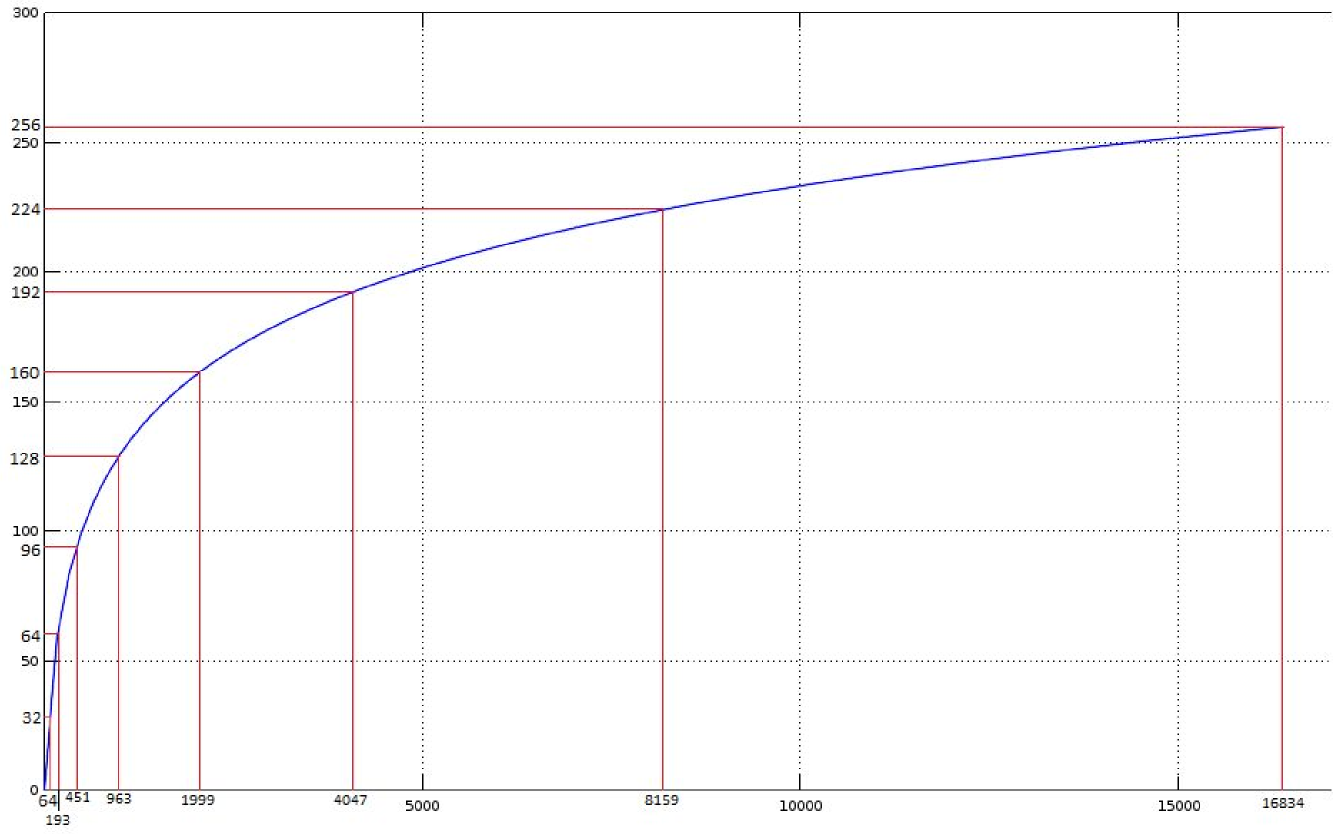

Figure 3: Calculated Segments using log_2

3.2.3 Slope Calculation

The slope is calculated using the basic slope formula of rise over run. The slope is scaled by 2^{16} to match the 14-bit to 8-bit design

Table 3: Slope Calculation

Segment #	Slope = $(y_2 - y_1)/(x_2 - x_1)$	Scaled slope
8	0.5	32768
7	0.248	16257
6	0.124	8128
5	0.0625	4096
4	0.03088	2024
3	0.0156	1024
2	0.0077	510
1	0.00389	255

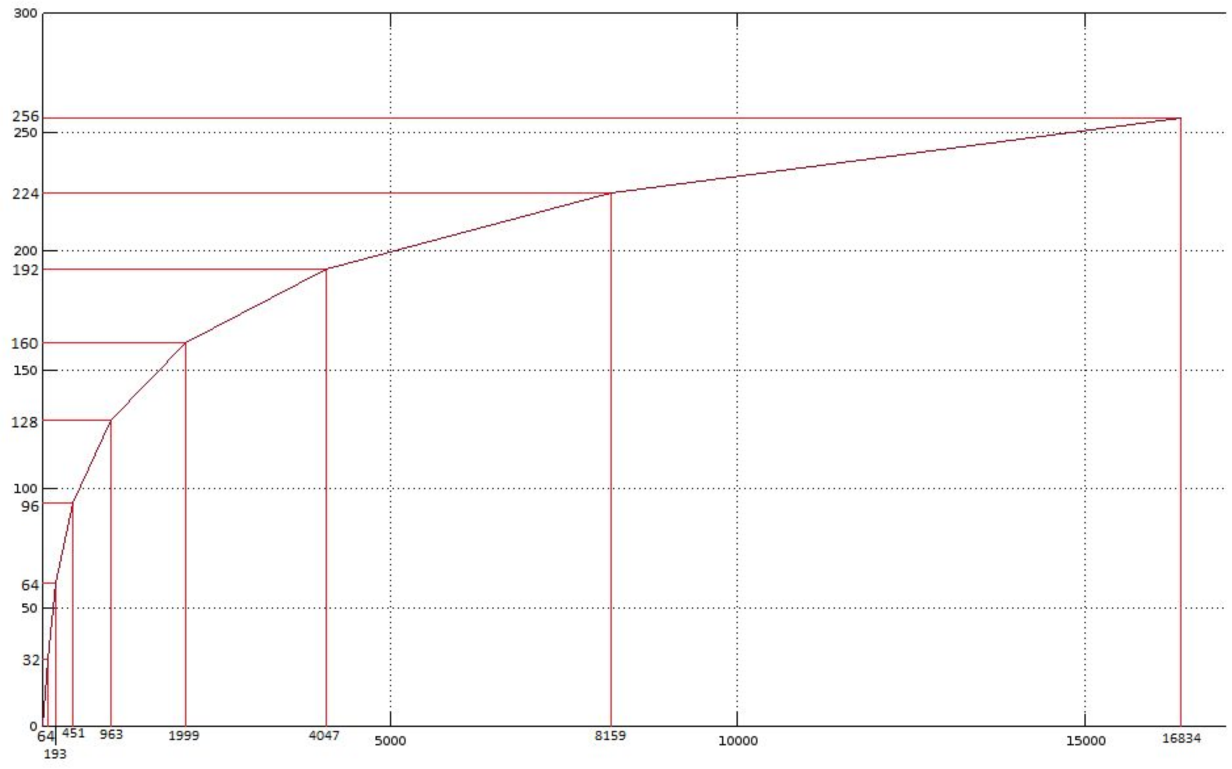

Figure 4: Calculated slope

3.3 Theoretical Example

The example below describe in details on how 14-bit(2^{14}) integer value is converted to 8-bit(2^{8}) integer value. For the demonstration purpose we choose the integer input of '10000'.

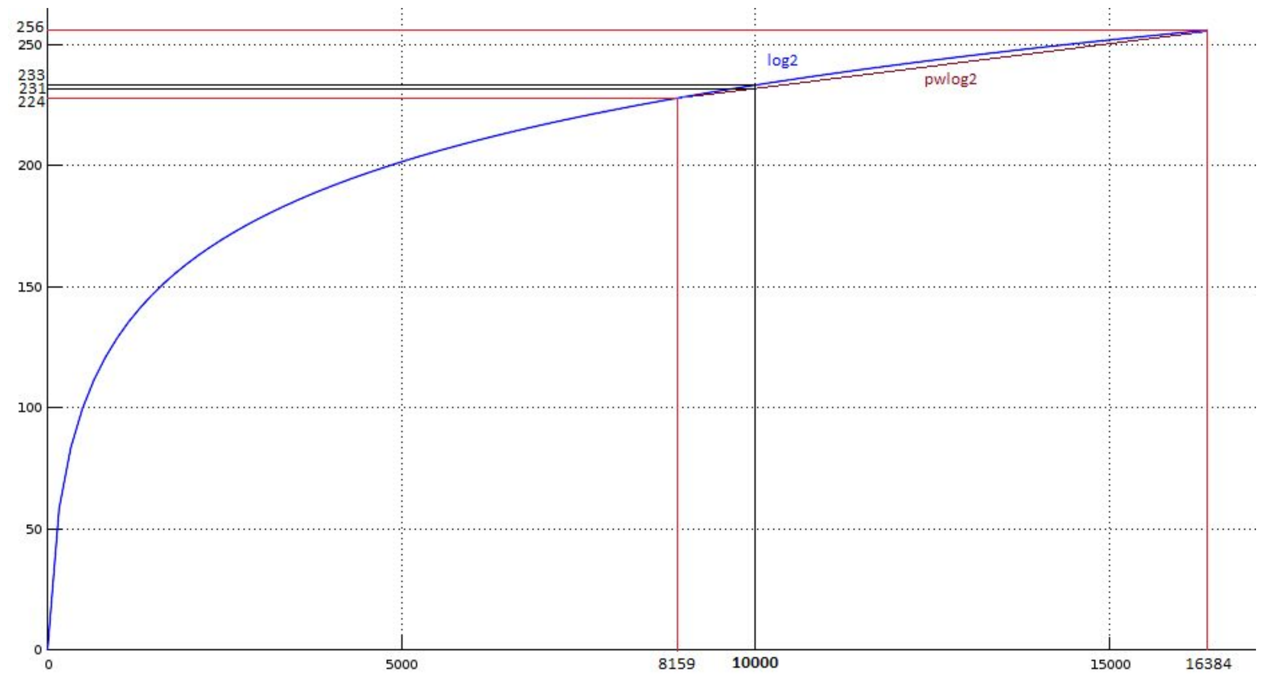

Figure 5:Example with input 10000

The input 10000 belongs to the 8th segment with slope 255 and ranges from 8159 to 16384 on x-axis and 224 to 256 on y-axis. For this particular segment following steps were performed

- 10000 must be subtracted from 8159 to normalise for x-axis 8th segment
- That gets multiplied by the slope of the segment which is 255
- Divided by 2^{16} since we have 32 bit register
- Add 224 to scale for y-axis

$$\frac{(10000-8159)*255}{2^{16}} + 224 = 231 \qquad (4)$$

This is the method which is implemented in our pwlog2 function implemented in C. We can further verify our results by using equation (2) and scaling it to 8-bit by multiplying by 256 (2^8).

$$\frac{1}{8}\ log_2(1 + 225 * \frac{10000}{2^{14}}) * 2^8 = 233 \qquad (5)$$

The values above verify our results and the small difference in these values are because of our approximation of the log function. Since we divided the input into 8 segments and each segment has a particular slope rather than a curve to calculate from. This results in an error of (2/233)%; the error between the log function and piecewise log function is further discussed in evaluation section of the report.

4 Design

4.1 Implementation in C

This section explains the implementation of the $\mu\ law$ in C. Below are some code snippets of our main.c with initial implementation before optimization methods were performed. The UML diagram explain the structure of the code and how the information flows. We timed the part when the function calls the pwlog2 function as rest of the implementation remains for the optimization. Our designed system creates an integer array with hundred unit interval for testing the pwlog2 function. For each value in array the pwlog2 function outputs a compressed value which is stored in a result array.

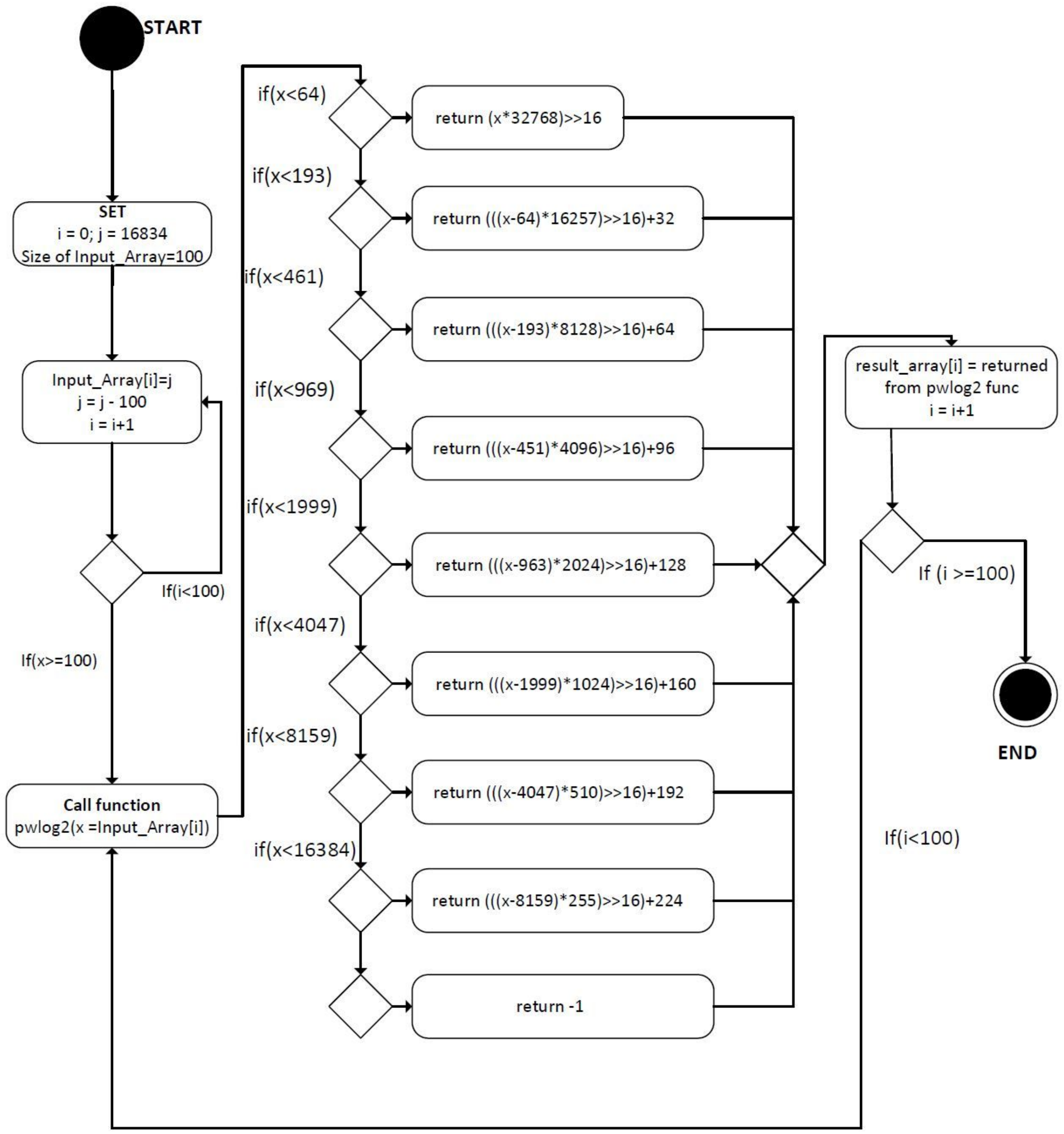
START
SET
i = 0; j = 16834
Size of Input_Array=100
Input_Array[i]=j
j = j - 100
i = i+1
If(i<100)
If(x>=100)
Call function
pwlog2(x =Input_Array[i])
if(x<64)
return (x*32768)>>16
if(x<193)
return (((x-64)*16257)>>16)+32
if(x<461)
return (((x-193)*8128)>>16)+64
if(x<969)
return (((x-451)*4096)>>16)+96
if(x<1999)
return (((x-963)*2024)>>16)+128
if(x<4047)
return (((x-1999)*1024)>>16)+160
if(x<8159)
return (((x-4047)*510)>>16)+192
if(x<16384)
return (((x-8159)*255)>>16)+224
return -1
result_array[i] = returned from pwlog2 func
i = i+1
If (i >=100)
END
If(i<100)

Figure 6: UML of the Design

Code: input array to represent 14-bit input samples

```
unsigned int sample_array[1<<14];
for (int i = 0; i < 1<<14; ++i){
    sample_array[i] = i;
}
```

The above code creates an integer sample array with size 2^{14} and input numbers 0,1,2,3… 2^{14} -1 into the array. Piecewise logarithmic function is performed on this sample array to get the final output.

Code: pwlog2 function

```c
unsigned int pwlog2(unsigned int x){
    if(x<64)
        return (x*32768) >> 16;
    if(x<193)
        return (((x-64)*16257)>>16)+32;
    if(x<461)
        return (((x-193)*8128)>>16)+64;
    if(x<963)
        return (((x-451)*4096)>>16)+96;
    if(x<1999)
        return (((x-963)*2024)>>16)+128;
    if(x<4047)
        return (((x-1999)*1024)>>16)+160;
    if(x<8159)
        return (((x-4047)*510)>>16)+192;
    if(x<16384)
        return (((x-8159)*255)>>16)+224;
    return -1;
}
```

The above code divides the input into 8 different segments. These segments are derived from the equations and calculation described in section 2.

Code: Error calculation

```c
for (int i = 0; i < samples; ++i){
    result_array[i] = pwlog2(sample_array[i]);
    compare_value = round(256*(log2(1+(255*(float)sample_array[i]/16384)))/8);
    compare_array[i] = (int)compare_value;
    diff_array[i] = compare_array[i] - result_array[i];
    if(diff_array[i] > max){
        max_index = i;
        max = diff_array[i];
    }
}
```

As described in section 2.3, there is a small difference between the pwlog2 function and actual log2 function compression values. The pwlog2 function relies on a slope of the segment which is a straight line while log2 function is curved line. The above function calculate the error and difference of the slope line vs the actual curved line.This helped us calculate the max error, that resulted in the middle segments i.e. segment numbers 4 and 5.

4.2 Optimization of C Code

4.2.1 Interval Reversal optimization

Given an evenly-distributed 14-bit input sample that ranges between the values of 0 to 16383, approximately half of the samples will fall in the last interval: [8159,16384). By reversing the order in which the interval that the input value exists in is found, the log2 approximation function `pwlog2()` is made faster.

Code:Optimised version 1

```
unsigned int pwlog2(unsigned int x) // pwlog2 = piecewise log2
{
    if(x>16384)
        return -1;
    if(x>8159)
        return (((x-8159)*255)>>16)+224;
    if(x>4047)
        return (((x-4047)*510)>>16)+192;
    if(x>1999)
        return (((x-1999)*1024)>>16)+160;
    if(x>963)
        return (((x-963)*2024)>>16)+128;
    if(x>451)
        return (((x-451)*4096)>>16)+96;
    if(x>193)
        return (((x-193)*8128)>>16)+64;
    if(x>64)
        return (((x-64)*16257)>>16)+32;
    return (x*32768)>>16;
}
```

4.2.2 Lookup Table Optimization

In order to further optimize the piecewise approximation of `log2()`, rather than calculating approximate values, a lookup table was used to speed up the function. The input values were chosen by finding the midpoint of each of the 8 intervals (32, 128, 322, 707, 1481, 3023, 6103, 12271), and values for the corresponding outputs were found using the previous approximation of `log2()`. These values were then stored in a lookup table and the new pw`log2()` function only returned the approximate value corresponding the that interval.

Code: Optimization version 2

```
unsigned int lookup_table[] = {19, 51, 83, 115, 147, 179, 211, 243};
unsigned int pwlog2(unsigned int x) // pwlog2 = piecewise log2
{
    if(x>=16384)
        return -1;
    if(x>=8159)
        return lookup_table[7];
    if(x>=4047)
        return lookup_table[6];
    if(x>=1999)
        return lookup_table[5];
    if(x>=963)
        return lookup_table[4];
    if(x>=451)
        return lookup_table[3];
    if(x>=193)
```

```
        return lookup_table[2];
    if(x>=64)
        return lookup_table[1];
    return lookup_table[0];
}
```

4.2.3 Lookup Table Elimination

In order to further optimize the `pwlog2()` function, rather than using a lookup table, the values for interval midpoint approximation were directly added to the function as constants to be returned.

Code: Optimization version 3

```
unsigned int pwlog2(unsigned int x) // pwlog2 = piecewise log2
{
    if(x>=16384)
        return 0;
    if(x>=8159)
        return 243;
    if(x>=4047)
        return 211;
    if(x>=1999)
        return 179;
    if(x>=963)
        return 147;
    if(x>=451)
        return 115;
    if(x>=193)
        return 83;
    if(x>=64)
        return 51;
    return 19;
}
```

4.2.4 Interval Detection Approximation

The final optimization method used took advantage of the closeness of the interval boundaries to powers of 2. This allowed the interval detection to be approximated by comparing it to the nearest power of 2, by right-shifting the input value. For example, if the desired interval to check was (2^{13},2^{14}), then the input would be shifted 13 bits `(x>>13)`. The output used in this case was still the interval midpoint approximation seen in the last two optimizations.

Code: Optimization version 4

```c
unsigned int pwlog2(unsigned int x) // pwlog2 = piecewise log2
{
    if(x>>14)
        return 0;
    if(x>>13)
        return 243;
    if(x>>12)
        return 211;
    if(x>>11)
        return 179;
    if(x>>10)
        return 147;
    if(x>>9)
        return 115;
    if(x>>8)
        return 83;
    if(x>>6)
        return 51;
    return 32;
}
```

5 Evaluation

To evaluate the effectiveness of the software optimization that was implemented, the time taken to compress a 100 element array of evenly-distributed 14-bit input values to an 8-bit output array was measured. The results of these findings can be seen in the table below. These results were found by running the programs using remote ssh-access to the linux machines found at seng440.ece.uvic.ca.

Table 4: Performance evaluation on Linux machine with Intel processor

Iteration	Time taken to complete (s)
mu.c	0.859375
mu1_opt.c	0.468750
mu2_opt.c	0.453125
mu3_opt.c	0.437500
mu4_opt.c	0.421875

As we can see above, the overall time taken to complete this task was reduced by 0.4375 seconds, or 50.91%, with each iteration providing an increase in efficiency. The following results were found using the qemu-arm command to run the program using an ARM simulator.

Table 5: Performance evaluation using an ARM processor simulator

Iteration	Time taken to complete (s)
mu.c	33.980000
mu1_opt.c	32.390000
mu2_opt.c	32.310000
mu3_opt.c	32.270000
mu4_opt.c	31.690000

A similar trend can be seen here, with each iteration of optimization take less time to complete this task. Here, a decrease of 2.29 seconds, or 6.74%, is seen.

6 Conclusion

Audio compression is made possible using a variety of techniques dependent on both the auditory system and software optimization. The human ear responds in a non-linear fashion that closely resembles a logarithmic response. This behaviour can be taken advantage of to create a compression method that uses non-uniform quantization, since humans will more likely detect noise in lower-level signals than in higher-level signals. A common method of quantization is μ-law compression, which was implemented in this project.

This was done by creating a piecewise linear approximation of the base-2 logarithmic function and using this function to encode 14-bit audio samples into 8-bit values. By using a piecewise linear approximation of the logarithmic function, an implementation that only used integer arithmetic was possible, rather than relying on expensive logarithmic operations.

After the initial implementation in C was complete, a variety of optimizations were performed to reduce the operation time of the piecewise base-2 logarithmic function. The first of these optimizations was a reversal of the intervals that the input was being checked to be in. The second of these optimizations was the creation of a lookup table to reduce the computation time of returning the 8-bit output and used a rough approximation for this output based on the midpoint of the interval that the input fell in. The next optimization took this lookup table and embedded the values as constants in the piecewise approximation function. The last optimization approximated the interval boundaries as powers of two; this allowed bitwise shifting to be used instead of checking the input against integer values. Each of these methods introduced a slight increase in error, but ultimately led to an approximately 60% faster program.

7 Bibliography

[1] Sima, M. (2017). *Lesson 109: Audio compression*. [pdf] Available at: https://www.ece.uvic.ca/~msima/TEACHING/COURSES/SENG_440/PROTECTED/SLIDES/SENG_440_slides_Lesson_project_109.pdf [Accessed 1 Aug. 2018].

[2] Young Engineering. *A-Law/Mu-Law Companding*. [ebook] Available at: http://www.young-engineering.com/docs/YoungEngineering_ALaw_and_MuLaw_Companding.pdf [Accessed 1 Aug. 2018].

[3] H.L., S. (2017). *Companding: Logarithmic Laws, Implementation, and Consequences*. [ebook] Available at: https://www.allaboutcircuits.com/technical-articles/companding-logarithmic-laws-implementation-and-consequences/ [Accessed 1 Aug. 2018].

[4] Wang, Y. (n.d.). *Quantization*. [pdf] Available at: http://eeweb.poly.edu/~yao/EE3414/quantization.pdf [Accessed 1 Aug. 2018].